NEWS 2010

LAUTERBACH DEVELOPMENT TOOLS

DEBUGGER, REAL-TIME TRACE, LOGIC ANALYZER

Stephan Lauterbach during a panel discussion at the IP/ESC in Grenoble, France

Communication: The Key to Success

Our success is due to building long term, solid relationships with our customers and partners. By keeping up-to-date with new technologies from our partners, and being proactive to the requirements of our customers, we are able to provide the right development tools at the right time. Successful communication requires active involvement, which will continue to be our philosophy in 2010.

Expert Seminars

To help expand our customers' level of product use and understanding, we hosted our first, industry specific, TRACE32 expert forum, October 2009, at our new headquarters in Germany. The essential outline of this forum was to exchange information between the system users and our TRACE32 developers. Positive feedback has impelled us to plan more like events during 2010.

Committed to Standards

In addition to the ongoing dialogue with our embedded industry partners, our participation in standards committees represents a critical means of exchanging information and fostering contacts. Over the years, we have incorporated many of the results from these committees into our products. In this newsletter, we'd like to provide you with more details regarding our involvement in this area.

Panel Discussions

Trade show panel discussions provide further opportunities to discuss current and future market requirements with customers and partners. In November 2009, for example, Stephan Lauterbach took part in a discussion at the IP/ESC in Grenoble, France. The discussion, initiated by ARM, focused on the future of debugging and trace technology.

We are looking forward to talking with you at the upcoming ESC Silicon Valley in San Jose (booth# 1910).

CONTENTS

New TRACE32 Installer for Windows 7

With the new TRACE32 DVD from December 2009 (Build 20817/Release), Lauterbach provides official support for Windows 7.

The TRACE32 installer has been enhanced to simplify the installation of TRACE32 USB drivers on systems running Windows 7. This was necessary because Windows 7 only installs drivers automatically using the Windows Update Server or a pre-installed driver package.

If you do not have a new DVD available, you can download the new *TRACE32 USB driver installer for Windows XP/Vista/7 (32-bit and 64-bit)* under the following link:

http://www.lauterbach.com/faq/t32usb_setup.exe

The handling as well as the look and feel of TRACE32 remains the same on systems running Windows 7. To provide even better support for the Windows security model and to make automatic installation easier, from December 2009, the executable TRACE32 files on the DVD are also signed. A signature has been used on the USB driver since 2007.

Programming Serial Flash Devices

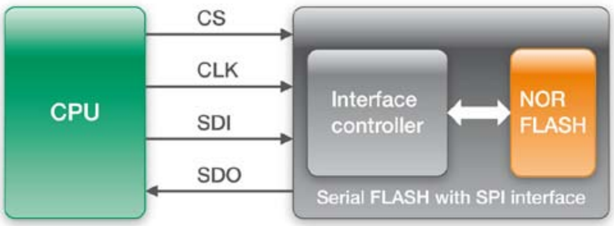


Fig. 1: Serial flash with SPI interface

Embedded systems are increasingly being designed with serial flash memory devices. Lauterbach recognized this trend early and since mid-2009, TRACE32 debuggers have supported the programming of serial flash devices. We will be increasing this support significantly during 2010.

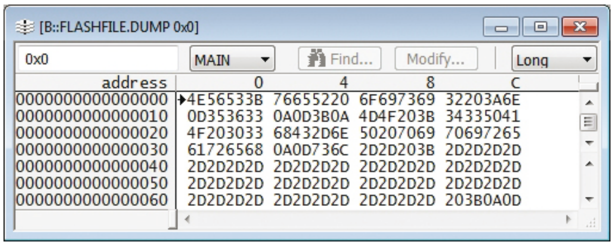
Low pin count, compact format, and reduced energy consumption make serial flash devices a cost-effective alternative to NOR/NAND flash devices. The concept behind serial flash design is simple: An interface controller is connected upstream of the NOR/NAND flash device, enabling you to program and read out the component over an SPI bus or MMC bus (see Figure 1).

TRACE32 support for serial flash devices includes programming as well as reading and displaying contents. The flash contents are displayed in a conventional hex dump, which enables you to check the program data quickly (see Figure 2).

To find out if TRACE32 supports your serial flash, please check the following lists:

Supported NAND/SERIAL FLASH Controllers
http://www.lauterbach.com/ylistnand.html

Supported NOR/NAND/SERIAL FLASH Devices
http://www.lauterbach.com/ylist.html



```
         address   0        4        8        C
0000000000000000 ▸4E56533B 76655220 6F697369 32203A6E
0000000000000010  0D353633 0A0D3B0A 4D4F203B 34335041
0000000000000020  4F203033 68432D6E 50207069 70697265
0000000000000030  61726568 0A0D736C 2D2D203B 2D2D2D2D
0000000000000040  2D2D2D2D 2D2D2D2D 2D2D2D2D 2D2D2D2D
0000000000000050  2D2D2D2D 2D2D2D2D 2D2D2D2D 2D2D2D2D
0000000000000060  2D2D2D2D 2D2D2D2D 2D2D2D2D 203B0A0D
```

Fig. 2: A visual representation of serial flash content

Debuggers for Intel® Atom™ Microarchitecture

Since October 2009, Lauterbach has been offering development tools for Intel® Atom™. Linux debugging is already fully supported, and Windows CE support is planned for early 2010.

Supported Derivatives

Intel®	**LA-3776 (Atom)**	
	• 230	• N270
	• 330	• N280
	• D410	• N450
	• D510	• Z5XX
	Other derivatives are planned.	

Differential Loading

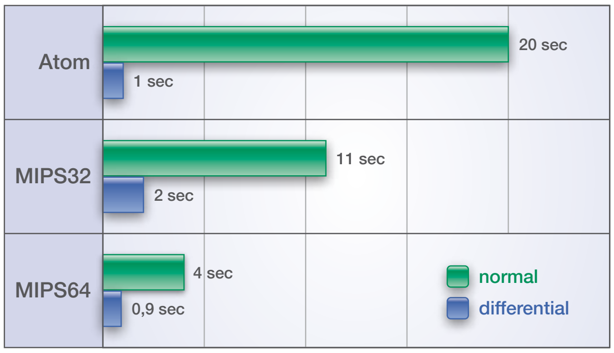


Fig. 3: Differential loading makes it significantly faster to load a 4-MByte file.

A typical debug cycle consists of the following steps: debug the program – locate errors – correct errors – compile the program – reload the program. Your tool chain has to be able to perform each individual step quickly and without any delays.

Long waiting times will occur when downloading a large program to the target RAM via a slow JTAG interface. Differential loading can provide a remedy for this problem. If the newly compiled program differs only slightly from the previously loaded program, the loading times can be cut significantly.

The basic concept behind differential loading is that the debugger holds a copy of the program that is already loaded. When loading the newly compiled version a differential file is created. The differential file contains all of the information, in compressed form, that is required to update the original file to the newly compiled version. The debugger then downloads only the differential file to the target system. Because the differential file is usually thirty to one hundred times smaller than the complete newly compiled program, the download is much faster. A load agent on the target system then decompresses the differential file and makes sure that the new compilation is stored in the target memory.

The measurements in Figure 3 were performed for a test file in which one percent of the program had changed.

1. Atom architecture
CPU: Z530P from Intel
Processor frequency: 1.6 GHz
JTAG frequency: 20 MHz
Normal download: 204 KByte/s

2. MIPS32 architecture
CPU: BCM7325 from Broadcom
Processor frequency: 167 MHz
JTAG frequency: 20 MHz
Normal download in Turbo Mode: 370 KByte/s

3. MIPS64 architecture
CPU: OCTEON Plus CN58XX from Cavium
Processor frequency: 950 MHz
JTAG frequency: 50 MHz
Normal download in Turbo Mode: 1 MByte/s

New Supported Processors

New Derivatives	
ARC	**LA-3750 (ARC)** • ARC 601/ARC 630
ARM	**LA-7843 (Cortex-A/R)** • Cortex-A5/Cortex-A5MPCore • Cortex-A9/Cortex-A9MPCore **LA-7844 (Cortex-M)** • Cortex-M0/Cortex-M1
ATMEL	**LA-3779 (AVR32)** • AVR32 (Q2/2010) **LA-7844 (Cortex-M)** • AT91SAM3U
Broadcom	**LA-7760 (MIPS32)** • BCM3380 • BCM56xxx/5836 • BCM6362/6368/6550 • BCM7401/7402/7403
CEVA	**LA-3711 (CEVA-X)** • CEVA-X1641 • CEVA-XC (Q2/2010) **LA-3774 (TeakLite-III)** • TeakLite-III
Cavium	**LA-7761 (MIPS64)** • Octeon CN54xx/CN56xx • Octeon CN63xx **LA-7765 (ARM11)** • ECONA CNS3XXX
Cortus	**LA-3778 (APS)** • APS-IP (Q2/2010)
Energy Micro	**LA-7844 (Cortex-M)** • EFM32
Freescale	**LA-7736 (MCS12X)** • MC9S12G **LA-7742 (ARM9)** • i.MX23/i.MX25 **LA-7732 (ColdFire)** • V1 ColdFire Core **LA-7753 (MPC55xx)/ LA-7630 (NEXUS MPC55xx)** • MPC5643L **LA-7764 (PowerQUICC III)** • QorIQ P1013/P1022/P4080
Infineon	**LA-7756 (TriCore)** • TC1167/TC1197/TC1337 • TC1367/TC1387/TC1387ED • TC1782/TC1782ED
Infineon (Cont.)	**LA-7759 (XC2000/C166S V2)** • XC2000ED • XC2200/XC2300/XC2700 • XE166/XGOLD110
LSI	**LA-7834 (StarCore)** • StarPro25xx/26xx
Marvell	**LA-7742 (ARM9)** • 88AP128/162/166/168 • MV76100/78100/78200 **LA-7765 (ARM11)** • 88SV581X-V6 **LA-7843 (Cortex-A/R)** • 88SV581X-V7 **LA-7762 (XScale)** • PXA93x/PXA950
MIPS	**LA-7760 (MIPS32)** • MIPS32 1004K/1004KF • MIPS32 1004K CPS • MIPS32 M14K/M14Kc
NEC	**LA-3777 (78K0R)** • 78K0R/Fx3, 78K0R/Kx3 **LA-7835 (V850)** • V850E2/Px4 • 70F3502/70F3504/70F3506
NXP	**LA-7844 (Cortex-M)** • LPC13xx **LA-7742 (ARM9)** • LPC29xx
ST Microelectronics	**LA-7753 (MPC55xx)/ LA-7630 (NEXUS MPC55xx)** • SPC56EL60 **LA-7844 (Cortex-M)** • STM32F105/STM32F107
Tensilica	**LA-3760 (Xtensa)** • Xtensa 8
Texas Instruments	**LA-7847 (TMS320C28X)** • TMS320F28232 • TMS320F28234/F28235 **LA-7838 (TMS320C6400)** • TMS320TC6424 • TMS320TCI6482/I6488 **LA-7843 (Cortex-A/R)/ LA-7838 (TMS320C6400)** • AM3505/AM3517 (Sitara) • OMAP4430/OMAP4440 **LA-7742 (ARM9)/ LA-7841 (TMS320C6700)** • OMAP-L137/OMAP-L138

Debugging AMP and SMP Systems

Many multi-core processors can be used as either AMP or SMP systems. Depending on the mode of operation, different debug and trace concepts are applicable. In the following article, we describe how these concepts can be applied using TRACE32 to debug an ARM Cortex-A9 MPCore.

Debug Concepts

In discussion with our customers, we realize again and again that there are many varying interpretations of the two terms:

- **AMP – asymmetrical multiprocessing**
- **SMP – symmetrical multiprocessing**

Therefore we think it is worth taking the time to explain how these terms are used at Lauterbach and what effect they have on the configuration and usage of a TRACE32 debugger.

As the term "multiprocessing" implies, multiple cores are working together in an embedded system. What is crucial for debugging is how the system tasks are distributed to the individual cores.

Debug Concept for AMP Systems

In AMP systems, each core is assigned a specific task. How the tasks are distributed is determined in the design phase of the system. In addition to a standard controller (usually RISC architecture) specialized accelerators (DSPs or customized cores) are frequently chosen.

When debugging AMP systems, an individual TRACE32 instance is started for each core (see Figure 4). There are two reasons for this:

1. An AMP system can contain different core architectures.
2. Each core processes a separate part of the application. This means that the majority of the symbol and debug information is assigned exclusively to the corresponding core.

However, because the cores do not work independently, but perform the application task together and in parallel, it must be possible to start and stop all cores simultaneously. This is the only way to test the interaction between the cores and to monitor and control the entire application. There are different ways to start and stop all cores simultaneously. Ideally, the multi-core processor will support this through internal synchronization logic. If this logic is missing, TRACE32 takes over the synchronization process. A special algorithm calculates JTAG command sequences to control all cores as promptly as possible.

»

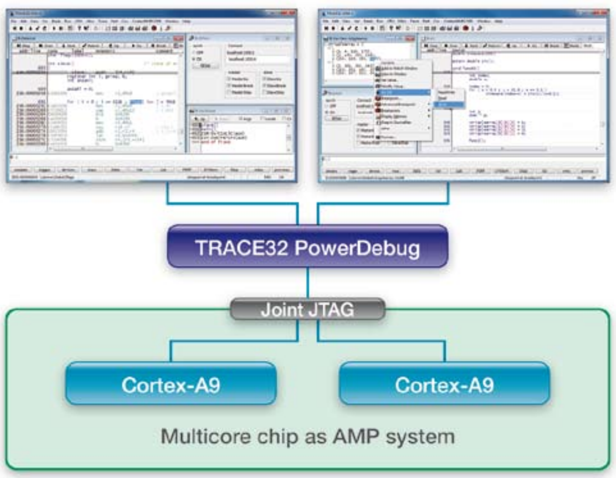
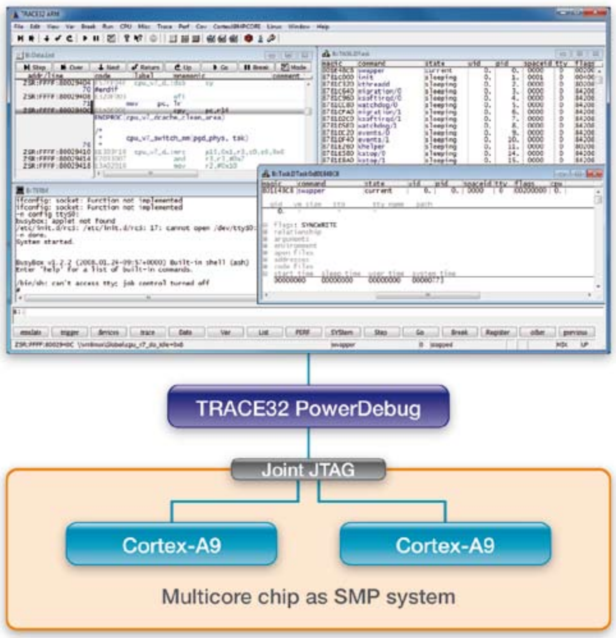


Fig. 4: When debugging AMP systems, an individual TRACE32 instance is started for each core.



```
ifconfig: socket: Function not implemented
ifconfig: socket: Function not implemented
-n config ttyS0:
busybox: applet not found
/etc/init.d/rcS: /etc/init.d/rcS: 17: cannot open /dev/ttyS0:
-n done.
System started.


BusyBox v1.2.2 (2008.01.24-09:57+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

/bin/sh: can't access tty; job control turned off
#
```

Fig. 5: When debugging SMP systems, a single TRACE32 instance is started for all cores.

Debug Concept for SMP Systems

In contrast to AMP systems, where the tasks assigned to each core are predefined, the assignment in SMP systems is flexible. In an SMP system, the system designer no longer assigns tasks to cores. An SMP operating system does this instead. All cores must be the same type to enable tasks to be assigned freely to each core as required.

Task assignment is performed dynamically, meaning that the assignment depends on the current system status. A task unit that can be assigned by an operating system is called a "task" or "thread". In simple terms, a task that needs to be processed is assigned to a core that is currently free.

For debugging SMP systems, only **one** TRACE32 instance is opened and all cores are controlled from this one point (see Figure 5 on the previous page). Because the developer is primarily concerned with debugging a single task, the TRACE32 user interface displays the state of the entire SMP system from the perspective of this single task or from the perspective of the core where the task is running. Of course the visualization can be switched to other tasks or cores if required.

TRACE32 assumes a function that is similar to an SMP operating system. It organizes the debugging of all cores so that developers do not need to look into the details of the SMP system. For example, if a breakpoint is set, TRACE32 makes sure that the breakpoint is entered in all cores. This is necessary because at the time when the breakpoint is set, it's not yet clear which core will execute the program section with the breakpoint. If a core stops at a breakpoint, all other cores are also stopped automatically. The display in TRACE32 switches to the task or core that hits the breakpoint. If the program is restarted, all cores start running together.

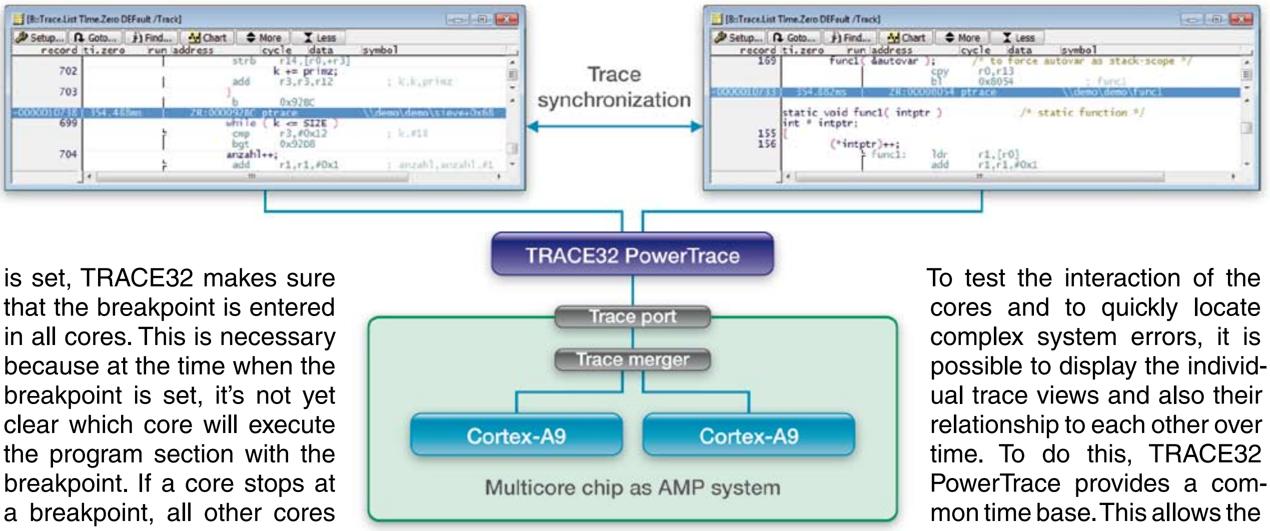
Debugging SMP systems with TRACE32 is easy. After a TRACE32 instance is started and configured for the SMP system, the developer essentially use it as if he were debugging only one core.

Trace Concepts

TRACE32 analyzes and displays trace information in different ways, depending on whether the trace data was generated by an AMP system or an SMP system. For AMP systems, trace analysis is largely performed on each core independently. The trace information for an SMP system, however, can be analyzed for a single task, a single core, or for the entire system, depending on the type of query.

Trace Concept for AMP Systems

Because debugging individual cores of an AMP system is performed over separate TRACE32 instances, trace information is also displayed on these individual user interfaces. AMP systems can consist of different types of cores, so different trace protocols might be used. As the individual trace streams are displayed in the separate user interfaces, they can be individually decoded and analyzed.



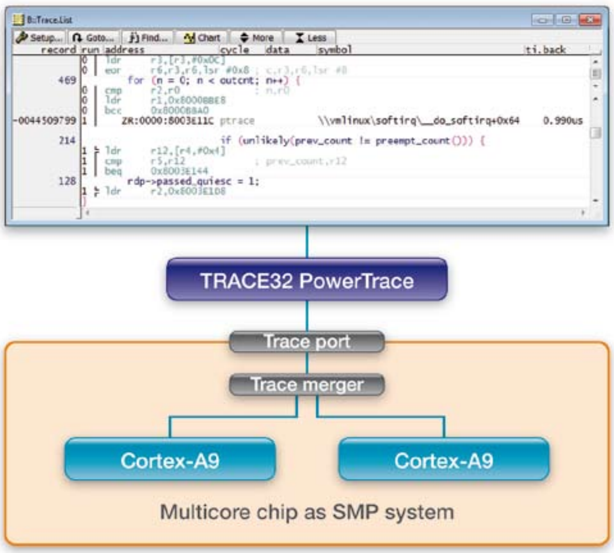
```
record ti.zero   run address       cycle   data     symbol
                          strb     r14,[r0,+r3]
     702                       k += primz;
                          add      r3,r3,r12          ; k,k,primz
     703                  }
                          b        0x928C
-000010738 354.488ms   ZR:0000928C ptrace              \\demo\demo\sieve+0x68
     699                while ( k <= SIZE )
                          cmp      r3,#0x12           ; k,#18
                          bgt      0x92D8
     704                anzahl++;
                          add      r1,r1,#0x1         ; anzahl,anzahl,#1
```

```
record ti.zero   run address       cycle   data     symbol
     169          func1( &autovar );          /* to force autovar as stack-scope */
                          cpy      r0,r13
                          bl       0x8054             ; func1
-000010733 354.882ms   ZR:00008054 ptrace              \\demo\demo\func1

          static void func1( intptr )                 /* static function */
          int * intptr;
     155  {
     156          (*intptr)++;
                  func1:  ldr      r1,[r0]
                          add      r1,r1,#0x1
```

Fig. 6: When tracing AMP systems, the trace information for each core is displayed on a separate user interface. Time synchronization of the user interfaces is possible.

To test the interaction of the cores and to quickly locate complex system errors, it is possible to display the individual trace views and also their relationship to each other over time. To do this, TRACE32 PowerTrace provides a common time base. This allows the developer to select a point in time in the trace view on one user interface and see exactly »



```
     record run address            cycle   data   symbol                              ti.back
               0  ldr    r3,[r3,#0x0C]
               0  eor    r6,r3,r6,lsr #0x8 ; c,r3,r6,lsr #8
        469          for (n = 0; n < outcnt; n++) {
               0  cmp    r2,r0             ; n,r0
               0  ldr    r1,0x80008BE8
               0  bcc    0x80008BA0
-0044509799 1     ZR:0000:8003E11C ptrace        \\vmlinux\softirq\__do_softirq+0x64    0.990us
        214                     if (unlikely(prev_count != preempt_count())) {
               1  ldr    r12,[r4,#0x4]
               1  cmp    r5,r12            ; prev_count,r12
               1  beq    0x8003E144
        128          rdp->passed_quiesc = 1;
               1  ldr    r2,0x8003E108
```

Fig. 7: When tracing SMP systems, the information for all cores is stored in a shared trace memory.

which command was being executed by another core at approximately the same time (see Figure 6).

Trace Concept for SMP Systems

All information about the programs processed on an SMP system is stored in a shared trace memory for all cores (see Figure 7). One of the advantages of TRACE32 is that it provides different views of this information.

To locate errors in a task or for task-specific runtime measurements, trace information can be displayed specifically for an individual task.

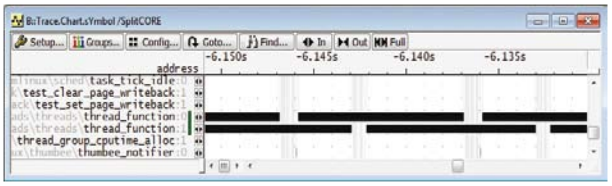


Fig. 8: The trace analysis shows which cores processed the individual program sections.

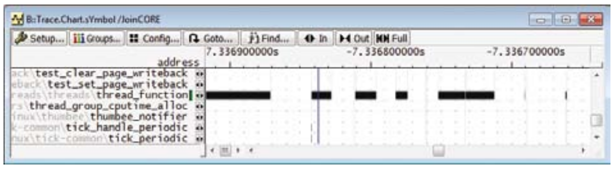


Fig. 9: The trace analysis analyses the SMP system as a whole; which core processed which program section is unimportant.

If you want to know information such as "Which cores processed my task?" or "What is the run-time load of my cores?", it can be useful to view the trace information for all cores at the same time. Figure 8 shows an example of this view. The core number (0 or 1) indicates the cores on which the individual program sections ran.

In order to examine the SMP system as a whole, it is not necessarily to know which core processed which task or program section. TRACE32 also provides display options for this type of view of the SMP system (see Figure 9).

During 2010, Lauterbach will continue to enhance the preparation and display of trace information from SMP systems. This will include new analysis functions based upon feedback from existing users and also new concepts currently in development.

The Latest on RTOS Debugging

New Supported RTOS	
FAMOS for ARM	available
Linux for Atom	available
SMP Linux for MIPS64	available
OKL4 for ARM	available
OS21 for ARM	available
SMP Symbian OS for ARM	available
Windows CE for Atom	Q1/2010

Updating for new RTOS versions
- LynxOS 5.0 for PowerPC
- MQX 3.x for ColdFire
- OSE 5.4
- QNX 6.4
- VxWorks 6.4
- µC/OS-III

Enhancements
- Partitions and MPU/MMU support for µC/OS-II
- Paged breakpoints for Symbian OS
- RTP support for VxWorks

Standards Committees

Many customers would like to see a higher level of standardization for debug and trace technologies. In order to take an active role in the development of suitable standards, Lauterbach has been participating in a number of international committees over many years. Our active participation makes it possible for us to include full support within TRACE32 for all new standards as soon as they are approved.

AUTomotive Open System ARchitecture

AUTOSAR

www.autosar.org

Working group
OSEK/VDX Debug Interface Working Group

Standard
OSEK Run Time Interface Version 2.2, Nov. 2005
http://portal.osek-vdx.org/files/pdf/specs/orti-a-22.pdf
http://portal.osek-vdx.org/files/pdf/specs/orti-b-22.pdf

ORTI Standard

One of the first standards that Lauterbach actively helped to develop was the ORTI standard. This standard deals with a descriptive language that describes the structure and memory mapping of OSEK operating system objects for RTOS-aware debugging and stores it in a file. This standard was developed within the AUTOSAR consortium and has been applied since 2003 by all providers of OSEK operating systems, including ETAS Group and Vector.

Working with an ORTI file can be described as follows: The OSEK System Builder creates the ORTI file from the user configuration of an OSEK operating system. The ORTI file is then loaded in the TRACE32 debugger in order to provide OSEK-aware debugging (see Figure 10).

TRACE32 can then offer the developer the following functions (presuming the OSEK System Builder has saved the required information in the ORTI file):

- Intuitive display of OSEK resources (Figure 11 shows the alarm list as an example)
- Task-specific breakpoints
- A task stack analysis
- A task context view
- Analysis of task runtimes (see Figure 13)
- Analysis of service runtimes

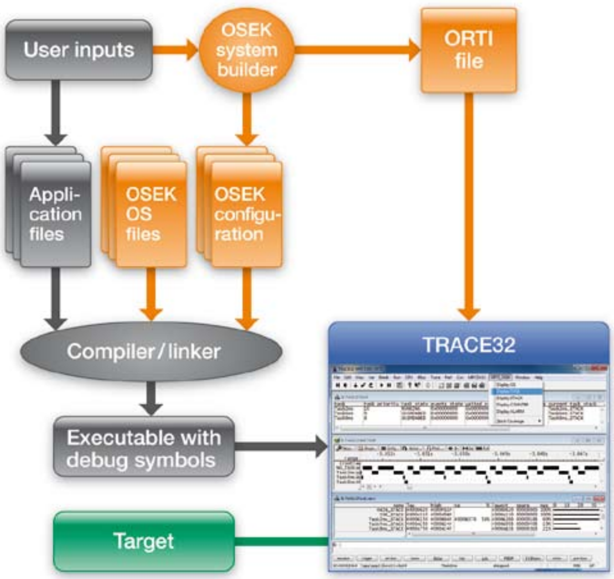
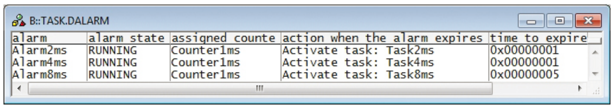


Fig. 10: The OSEK System Builder creates an ORTI file that enables OSEK-aware debugging after the file is loaded in TRACE32.



B::TASK.DALARM

alarm	alarm state	assigned counte	action when the alarm expires	time to expire
Alarm2ms	RUNNING	Counter1ms	Activate task: Task2ms	0x00000001
Alarm4ms	RUNNING	Counter1ms	Activate task: Task4ms	0x00000001
Alarm8ms	RUNNING	Counter1ms	Activate task: Task8ms	0x00000005

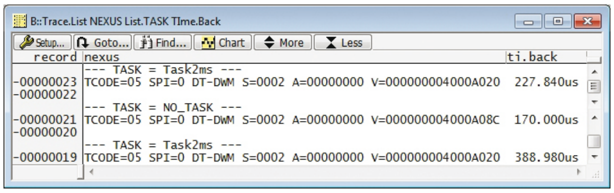
Fig. 11: The alarm list of an OSEK operating system in TRACE32

NEXUS Standard

Back in 1998, at the NEXUS 5001 Forum, a first step was taken toward standardizing a debug interface and a trace interface for embedded processors. The NEXUS standard was then approved in 2003. This includes:

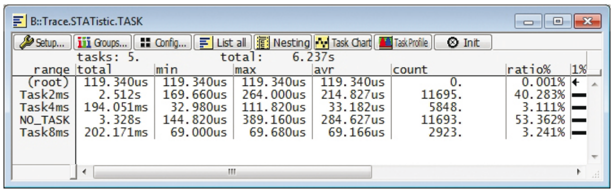
- A JTAG interface, usually IEEE1149.1
- Mechanisms that enable a debugger to read and write to memory, while the program execution is running
- A message-based trace protocol (program flow as well as data trace)
- Debugging and tracing for multi-core processors
- A hardware layer
- Standard NEXUS connectors

»



```
   record nexus                                                     ti.back
          --- TASK = Task2ms ---
-00000023 TCODE=05 SPI=0 DT-DWM S=0002 A=00000000 V=000000004000A020  227.840us
-00000022
          --- TASK = NO_TASK ---
-00000021 TCODE=05 SPI=0 DT-DWM S=0002 A=00000000 V=000000004000A08C  170.000us
-00000020
          --- TASK = Task2ms ---
-00000019 TCODE=05 SPI=0 DT-DWM S=0002 A=00000000 V=000000004000A020  388.980us
```

Fig. 12: NEXUS messages for recording task switching



```
          tasks: 5.           total:    6.237s
   range total      min        max        avr        count           ratio%   1%
  (root)  119.340us  119.340us  119.340us  119.340us           0.     0.001%
 Task2ms     2.512s  169.660us  264.000us  214.827us       11695.    40.283%
 Task4ms  194.051ms   32.980us  111.820us   33.182us        5848.     3.111%
 NO_TASK     3.328s  144.820us  389.160us  284.627us       11693.    53.362%
 Task8ms  202.171ms   69.000us   69.680us   69.166us        2923.     3.241%
```

Fig. 13: Analysis of task runtimes for an OSEK operating system

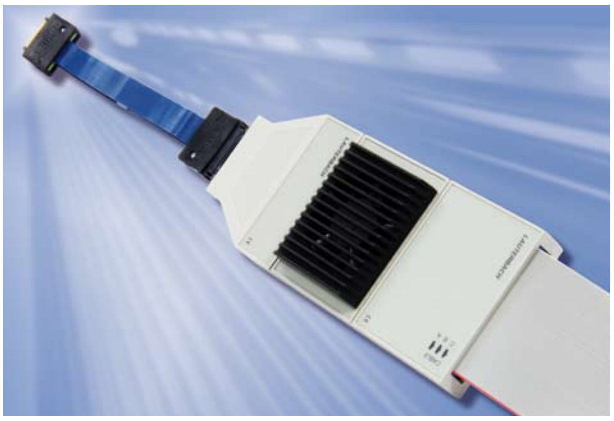


Fig. 14: TRACE32 High-Speed Serial Trace for QorIQ

Lauterbach has supported the NEXUS standard for various processor architectures since 2001. The latest and most important one is the MPC55xx/ MPC56xx family from Freescale and ST Microelectronics, which is widely used in the automotive industry. Figures 12 and 13 show how the NEXUS trace log is used to measure the runtimes of OSEK tasks.

Because the NEXUS standard gives a high degree of freedom in the implementation, Lauterbach completely redesigned its NEXUS hardware in 2008. The new FPGA-based hardware is designed to be so flexible, that it can be adapted to all NEXUS implementations. New features include support for an optimized sampling point for trace signals as well as the new JTAG protocol IEEE 1149.7. More information about IEEE 1149.7 can be found on page 10.

The 2003 NEXUS standard is currently being revised. Although the new version of the standard has not yet been approved, some processors from the MPC56xx family already implement this new standard. The new protocol is supported by TRACE32 by means of a software update available now.

Nexus 5001™

www.nexus5001.org

Standard
Standard for a Global Embedded Processor Debug Interface Version 2, December 2003
http://www.nexus5001.org/st/ieee_isto_5001_2003.pdf

Serial Trace for QorIQ

Lauterbach will present its new QorIQ High-Speed Serial Trace (see Figure 14) at the ESC Silicon Valley April 2010. The trace technology of QorIQ P4xxx is based on a NEXUS protocol, an AURORA-based hardware layer, and a connector approved by a working group of the Power.org organization.

The first processor to be supported will be the QorIQ P4080 from Freescale. The P4080 SoC features eight Power Architecture e500mc cores that can run in SMP or AMP configurations. Configurations that combine SMP and AMP groups are also possible. Detailed information about "Debugging AMP and SMP Systems" can be found on page 5 of this newsletter.

Compared to parallel trace interfaces, serial trace interfaces have the following advantages: pin reduction through serial transfer and high data throughput using differential signals to transfer the data.

The large volume of produced trace data naturally requires a correspondingly large trace memory buffer size. This is provided by the Lauterbach product PowerTrace II, which has a memory of up to 4 GByte.

High-Speed Serial Trace for the QorIQ is designed for a maximum of four high-speed channels. The following transfer rates are supported:

- 6.25 GBit/s max. per channel for up to 3 channels
- 3.125 GBit/s max. per channel for up to 4 channels

Trace data is captured over a connector system supplied by Samtec. Lauterbach supplies adapters for different variants of the connector.

»

Power.org

www.power.org

Working group
Common Debug Interface Technical Subcommittee

Standard
Power.org™ Standard for Physical Connection for High-Speed Serial Trace, July 2008
http://www.power.org/resources/downloads/Power_CDI_Physical_Connection_for_HSST_APPROVED_v1.0.pdf

IEEE Standards Association

www.standards.ieee.org

Working group: 1149.7

Standard
Official approval of the standard is planned for Q1/2010

IEEE 1149.7

Also known as cJTAG (compact JTAG), IEEE 1149.7 updates the familiar JTAG standard IEEE 1149.1 to meet the latest technical requirements. As the leading manufacturer of debuggers, Lauterbach has been heavily involved in the definition of this new 2-pin interface.

According to IEEE 1149.7, the entire communication between debugger and core can occur over the TCK and TMS pins. Compared to the standard debug communication, the new features can be described as follows:

- IEEE 1149.7 has only a two-pin connector on the outside. On the chip, the 1149.7 controller converts the communication back to standard JTAG (see Figure 15).
- Simply stated, the IEEE 1149.7 is serialized compared to the standard JTAG interface (see Figure 16).

Because the TMS signal is a unidirectional signal in standard JTAG, Lauterbach had to adapt its debug cables to support bidirectional TMS:

- All debug cables for ARM/Cortex architectures were converted at the beginning of 2008 (Debug Cable V4).
- An updated version of the debug cable for the MPC55xx/MPC56xx architecture has been available since September 2009 (OnCE Debug Cable with serial JTAG).

The new debug cables will of course continue to support the conventional standard.

TRACE32 was successfully tested with cJTAG in October 2009. However, it can only be released after the 1149.7 standard has been officially approved.

»

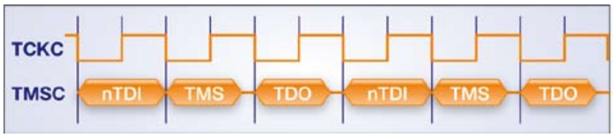


Fig. 16: Simplified diagram of the 1149.7 protocol

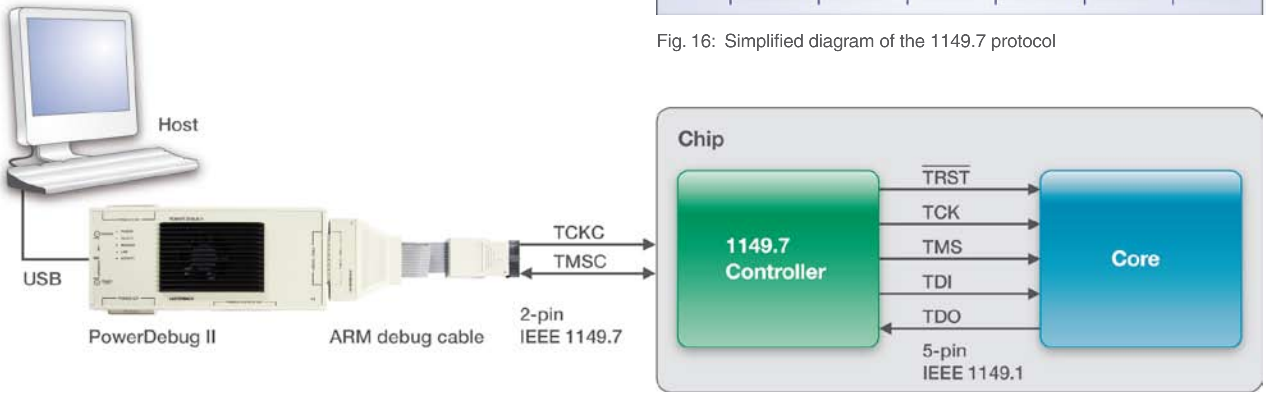


Fig. 15: The IEEE 1149.7 protocol is converted back to the standard JTAG protocol on the chip.

Open SoC Design Platform for Reuse and Integration of IPs

SPRINT

www.sprint-project.net

Working group: Debug and Analysis

Standard: Multi-Core Debug API v1.0, April 2009
http://www.lauterbach.com/sprint_mcd_api_v1_0.zip

MCD API

The MCD API defines a C interface for debugging multi-core systems. It makes no difference whether the multi-core system consists of real hardware or a software simulation. Lauterbach currently sees two applications for the MCD API:

1. MCD API as the standard interface for TRACE32, as the new TRACE32 MCD Remote API

2. MCD API as the standard interface for virtual prototypes

1. Updated TRACE32 Remote API

The Lauterbach Remote API allows an external application to control TRACE32. It enables for example the automation of regression tests.

Over the years, our remote API has been subject to ongoing development driven by the many new requests from our customers. Currently, the increasing number of multi-core systems would have required many further comprehensive modifications.

Instead of revising the current remote API, Lauterbach has decided to rebuild its remote API based on the MCD standard. It is highly probable that the MCD API does not cover all TRACE32 requirements, which means that functions specific to TRACE32 will be included (see Figure 17).

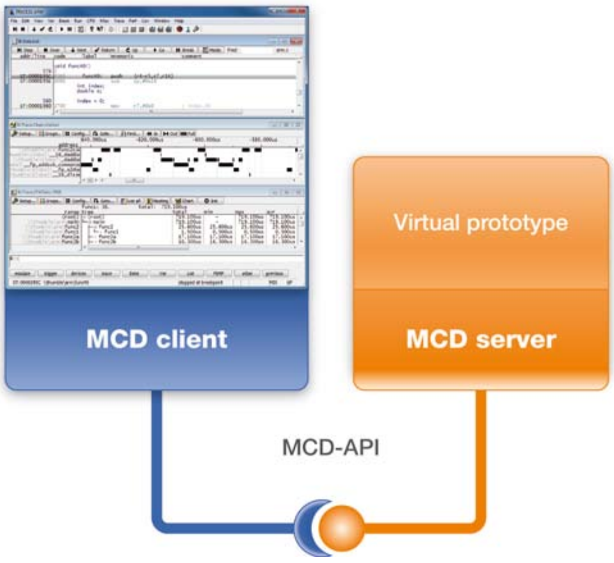


Fig. 18: Debugging a virtual prototype takes place over the MCD-API.

The release of the first version of the new TRACE32 MCD Remote API is scheduled for mid-2010. At this point in time, all development on the current Legacy Remote API has been "put on ice". For our customers, this means that although we will continue to support the Legacy Remote API, we will not be implementing any new functionality.

2. Standard Interface for Virtual Prototypes

For the past five years, TRACE32 has been capable of debugging virtual prototypes. Manufacturers of virtual prototypes typically provide their own debugging APIs for this purpose. The MCD API provides a standardized interface between debugger and virtual prototype (see Figure 18). This has the following advantages:

- Fast adaptation to new virtual prototypes
- Larger range of well-tested functions with higher performance

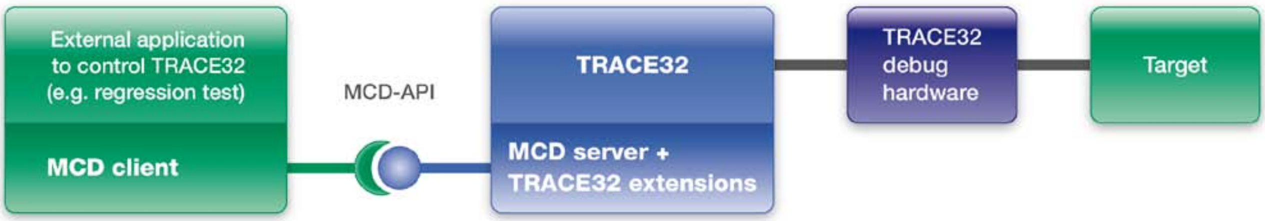

External application to control TRACE32 (e.g. regression test)
MCD client
MCD-API
TRACE32
MCD server + TRACE32 extensions
TRACE32 debug hardware
Target

Fig. 17: TRACE32 can be controlled by an external application over its Remote API.

Tip – More Read and Write Breakpoints

On-chip breakpoints are highly valuable debug resources that, on most processors, are only available in very limited numbers. For a long time it has been possible to set up two additional read/write breakpoints on an ARM core, provided the core included an Embedded Trace Macrocell (ETM). Using these breakpoints, there are extra options to define the break conditions with great accuracy. Unfortunately, not many developers are aware of this useful option.

Additional Read/Write Breakpoints	
ARM7/9/10/11 Cortex-A5/-A8/-R4	2 extra read/write breakpoints with data values
Ceva-X/TeakLite	2 extra read/write breakpoints with data values
TMS320C6400	Up to 4 extra read/write breakpoints with data values

Table 1: List of architectures for which additional breakpoints can be provided over the trace logic.

For efficient debugging, it is often important to be able to halt the program when a given data value is written to a variable. It is a disadvantage that not all ARM or Cortex cores provide the required read/write breakpoints. For example, the breakpoints in the Processor Debug Logic for the Cortex-A8 do not support this capability.

However, if the Cortex-A8 has an ETM logic, TRACE32 can provide this functionality by using two of the address and data comparators provided in the ETM. These comparators are then no longer available for their original function of filtering trace information and generating triggers. However, this is normally not critical because this functionality remains largely unused during debugging.

By setting the option “ETM.ReadWriteBreak”, the resource management of TRACE32 is reconfigured so that two address/data comparators of the ETM can be used as standard read/write breakpoints.

These breakpoints provide superior functionality when compared to the standard breakpoints provided in the Processor Debug Logic of the Cortex-A8. As an example, the read/write breakpoints of the Processor Debug Logic only offer bit masks for marking address areas, with the ETM breakpoints the address areas can be defined exactly. Because of this, TRACE32 gives priority to using ETM-based breakpoints. Table 2 compares the functionality of both breakpoint types.

Processor Debug Logic of Cortex-A8
1 to 16 read/write breakpoints Address area as bit mask No data values
ETM-based Breakpoints of Cortex-A8
2 read/write breakpoints Exact address area Both breakpoints with exact data values

Table 2: The additional ETM-based breakpoints offer enhanced functionality.

Note that ETM breakpoints can also be used even if the debugger in use is only the TRACE32 JTAG debugger. This is because the ETM comparators can be configured over the standard JTAG interface.

Similarly, as described for the Cortex-A8, trace logic features can be used in other architectures to provide additional breakpoints with enhanced functionality (see table 1).

BRANCHES AROUND THE WORLD

- Germany
- France
- UK
- Italy
- USA
- 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**