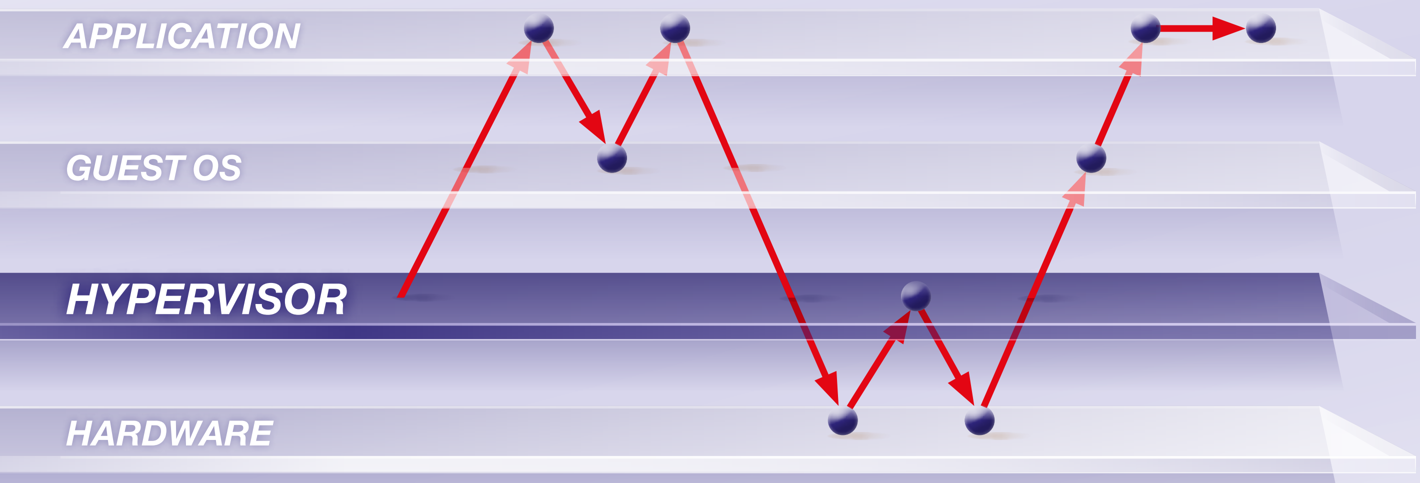
NEWS 2017

English Edition

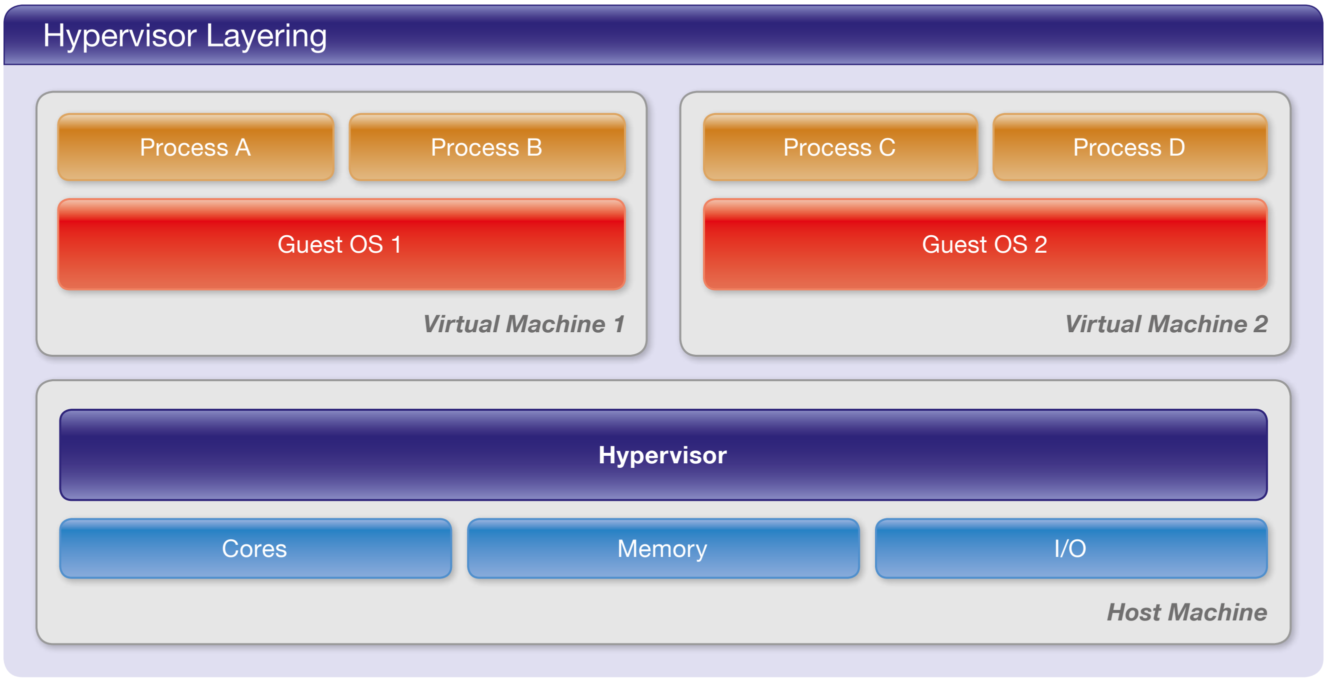


Seamless debugging through all software layers

CONTENTS

LAUTERBACH
DEVELOPMENT TOOLS

Hypervisor Debugging



In April 2017, Lauterbach will provide the high performance capabilities of its new hypervisor support. This article presents a reference implementation on which a Xen hypervisor with two Linux guests is running on a HiKey board from LeMaker (Cortex-A53).

Virtualization in Embedded Systems

The virtualization concept allows multiple operating systems to be run in parallel on a single hardware platform. Currently, virtualization is being used more and more in embedded systems. For example, in the cockpit of a car, real-time applications that are monitored by an AUTOSAR operating system run on the same hardware platform parallel to Android based user interfaces. A hypervisor, which is the core of virtualization, ensures that everything works reliably and efficiently.

The hypervisor, which is also referred to as a virtual machine monitor, is a software layer fulfilling two tasks:

1. Starting and managing the virtual machines (VMs).
2. Virtualizing the physical hardware resources for the VMs.

An operating system running on a VM is referred to as a guest OS. All accesses by the guests to the virtualized hardware resources are mapped to the physical resources by the hypervisor.

CPU virtualization is important for debugging. Every virtual machine is assigned one or more virtual CPUs (vCPUs). The number of vCPUs does not necessarily have to be the same as the number of CPU cores available on the hardware platform.

Memory virtualization is equally important. The VMs do not see the actual physical memory but see the guest physical memory as virtualized memory. The hypervisor manages a separate page table for each VM to control access to physical memory. Since the application processes, at least on operating systems like Linux, work with virtual addresses anyway, the debugger has to deal with a two stage address translation:

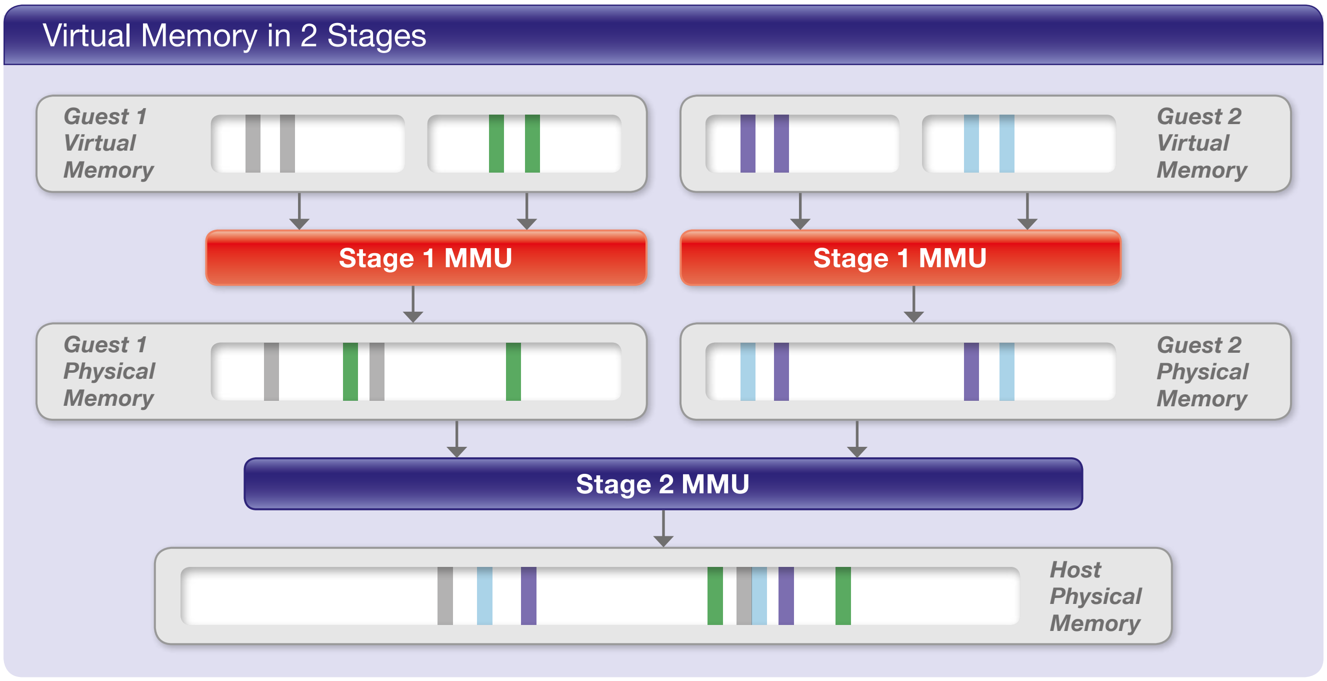
- Guest virtual memory to guest physical memory
- Guest physical memory to host physical memory

See the diagram "Virtual Memory in 2 Stages" on the opposite page:

- The Stage 1 MMUs mapping information is handled by the page table of their guest OS.
- The Stage 2 MMU uses the page tables of the hypervisor.

Extended Debugging Concepts

TRACE32 was systematically extended in 2016 by Lauterbach to provide its customers unlimited debug-



ging capability with a hypervisor. The following extensions were added:

- A machine ID was added to the TRACE32 command syntax. The machine ID allows the debugger to access the context of the active VM as well as the context of all inactive VMs. A virtual machine is considered active when a core has been allocated to it for execution.
- Using the new hypervisor-awareness, the debugger detects and visualizes the VMs of the hypervisor.
- Instead of only being able to debug a single operating system, it is now possible to debug several operating systems at the same time.
- Instead of only being able to access the OS page tables of the active guests as before, the debugger can now also use the page tables of all inactive guests.

The most important objective for all extensions was seamless debugging of the overall system. This means that when the system has stopped at a breakpoint, you can check and change the current state of every single process, all VMs, plus the current state of the hypervisor and of the real hardware platform. In addition, you can set a program breakpoint at any location in the code.

The unlimited debugging capability that Lauterbach has been offering for almost 20 years now, for operating systems like Linux, formed the starting point for all of these implemented extensions. Therefore, what follows is a brief summary of the most important OS debugging concepts:

Processes run on operating systems in a private virtual address space. The TRACE32 OS-awareness and the TRACE32 MMU support allow users to debug seamlessly across process boundaries:

- With the help of the space ID, it is possible to directly access the virtual address space of each process.
- With the help of the TASK option, it is possible to display the current register set and the stack frame for every single process.

Machine ID

How does this concept need to be extended if the operating systems are running on virtual machines?

1. First, it is necessary to uniquely identify each virtual machine. For this purpose, TRACE32 assigns each VM a number, the machine ID. The machine ID of the hypervisor is 0. Just as the space ID is used to identify the virtual address space of a process, the machine ID is used to identify the private address space of a VM.
2. To show the register set and the stack frame of any process, the debugger must know on which VM and on which guest OS the process is running. The MACHINE option was introduced for this purpose.

TRACE32 Commands

Traditional OS-Aware Debugging

```
Data.dump               <space _ id>:<virtual _ address>
Data.LOAD.Elf <file>    <space _ id>:<virtual _ address>
Register.view           /TASK <process _ name>
Frame.view              /TASK <process _ name>
```

< NEW >

Hypervisor Debugging

```
Data.dump               <machine _ id>:::<space _ id>::<virtual _ address>
Data.LOAD.Elf <file>    <machine _ id>:::<space _ id>::<virtual _ address>
Register.view           /MACHINE <machine _ id> /TASK <process _ name>
Frame.view              /MACHINE <machine _ id> /TASK <process _ name>
```

These two extensions are sufficient to allow the debugger to access all information across process boundaries. The "TRACE32 Commands" overview above provides a comparison of the extended TRACE32 command syntax for hypervisor debugging to the traditional syntax used for OS-aware debugging.

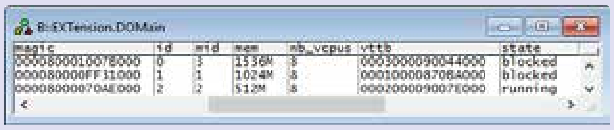
Hypervisor Awareness

Like the OS-awareness functionality, there is now a hypervisor-awareness functionality. This functionality provides the debugger with all information on the hypervisor running on the hardware platform. However, hypervisor-awareness requires the debug symbols for the hypervisor to be loaded. The debugger can then create an overview of all guests. The "Guest List" screenshot for our reference implementation — Xen, Cortex-A53 — shows the following information:

- VM IDs and VM states, number of vCPUs per VM
- Start addresses of the stage 2 page tables (vttb)

The awareness for the particular hypervisor is created by Lauterbach and provided to its customers. An overview of all currently supported hypervisors is shown in the table "Currently Supported Hypervisors" on page 5.

Guest List



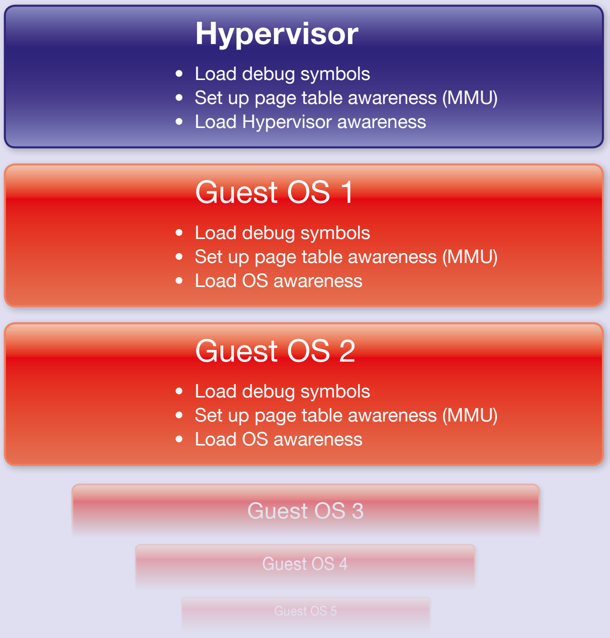
```
magic             id  mid  mem    nb_vcpus  vttb              state
000080001007B000  0   3    1536M  8         0003000090044000  blocked
00008000FF31000   1   1    1024M  8         000100008708A000  blocked
000080000070AE000 2   2    512M   8         00020000900 7E000 running
```

Debugger Configuration

How do the extended debug concepts effect debugging with TRACE32 now? Let‘s first look at the configuration. The following steps are necessary to configure the hypervisor as well as every single guest OS:

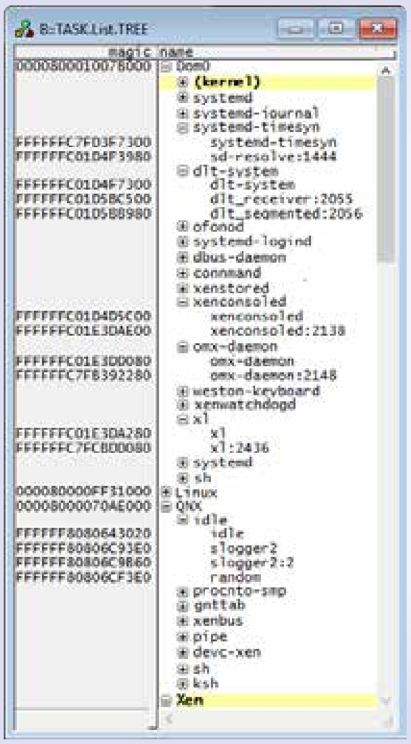
1. Load the debug symbols
2. Set up page table awareness (MMU)
3. Load the TRACE32 hypervisor-awareness respectively the TRACE32 OS-awareness

Debugger Configuration



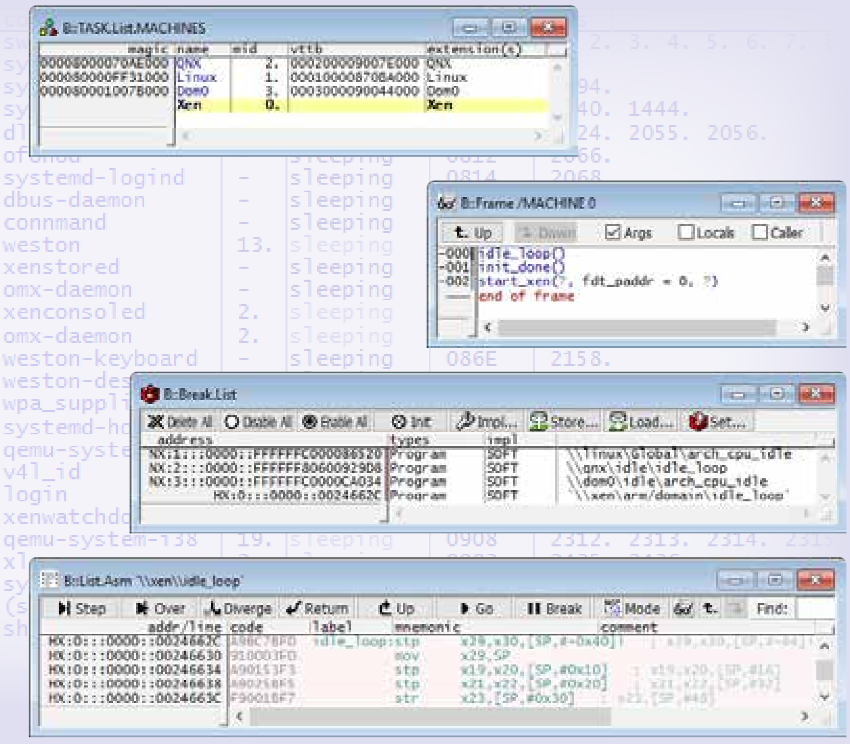
Xen Hypervisor on Cortex-A53

Global Task List



```
B::TASK.List.TREE
           magic  name
000080001007B000  ⊟ Dom0
                     ⊞ (kernel)
                     ⊞ systemd
                     ⊞ systemd-journal
                     ⊟ systemd-timesyn
FFFFFFC7FD3F7300        systemd-timesyn
FFFFFFC0104F3980        sd-resolve:1444
                     ⊟ dlt-system
FFFFFFC0104F7300        dlt-system
FFFFFFC0105BC500        dlt_receiver:2055
FFFFFFC0105B8980        dlt_segmented:2056
                     ⊞ ofonod
                     ⊞ systemd-logind
                     ⊞ dbus-daemon
                     ⊞ connmand
                     ⊞ xenstored
                     ⊟ xenconsoled
FFFFFFC0104D5C00        xenconsoled
FFFFFFC01E3DAE00        xenconsoled:2138
                     ⊟ omx-daemon
FFFFFFC01E3D0080        omx-daemon
FFFFFFC7F8392280        omx-daemon:2148
                     ⊞ weston-keyboard
                     ⊞ xenwatchdogd
                     ⊟ xl
FFFFFFC01E3DA280        xl
FFFFFFC7FCBD0080        xl:2436
                     ⊞ systemd
                     ⊞ sh
000080000FF31000  ⊞ Linux
00008000070AE000  ⊟ QNX
                     ⊟ idle
FFFFFF8080643020        idle
FFFFFF80806C93E0        slogger2
FFFFFF80806C9860        slogger2:2
FFFFFF80806CF3E0        random
                     ⊞ procnto-smp
                     ⊞ gnttab
                     ⊞ xenbus
                     ⊞ pipe
                     ⊞ devc-xen
                     ⊞ sh
                     ⊞ ksh
                  ⊟ Xen
```

Virtual Machine List



```
B::TASK.List.MACHINES
           magic  name   mid  vttb              extension(s)
00008000070AE000  QNX     2.  000200009007E000  QNX
000080000FF31000  Linux   1.  000100008708A000  Linux
000080001007B000  Dom0    3.  0003000090044000  Dom0
                  Xen     0.                    Xen
```

```
B::Frame /MACHINE 0
t. Up   Down   ☑ Args   ☐ Locals   ☐ Caller
-000 idle_loop()
-001 init_done()
-002 start_xen(?, fdt_paddr = 0, ?)
---- end of frame
```

```
B::Break.List
Delete All  Disable All  Enable All  Init  Impl...  Store...  Load...  Set...
address                               types    impl
NX:1:::0000::FFFFFFC000086320         Program  SOFT   \\linux\Global\arch_cpu_idle
NX:2:::0000::FFFFFF80600929D8         Program  SOFT   \\qnx\idle\idle_loop
NX:3:::0000::FFFFFFC0000CA034         Program  SOFT   \\dom0\idle\arch_cpu_idle
          HX:0:::0000::0024662C       Program  SOFT   \\xen\arm/domain\idle_loop
```

```
B::List.Asm '\\xen\\idle_loop'
Step  Over  Diverge  Return  Up  Go  Break  Mode  Find:
        addr/line  code      label           mnemonic                     comment
HX:0:::0000::0024662C  A9BC7BFD  idle_loop:stp  x29,x30,[SP,#-0x40]!   ; x29,x30,[SP,#-64]!
HX:0:::0000::00246630  910003FD                 mov  x29,SP
HX:0:::0000::00246634  A90153F3                 stp  x19,x20,[SP,#0x10]  ; x19,x20,[SP,#16]
HX:0:::0000::00246638  A90258F5                 stp  x21,x22,[SP,#0x20]  ; x21,x22,[SP,#32]
HX:0:::0000::0024663C  F90018F7                 str  x23,[SP,#0x30]      ; x23,[SP,#48]
```

The "Debugger Configuration" diagram shows an overview of the individual configuration steps.

Debug Process

The operation of a debugger must often resolve contradicting requirements. One user group wants simple and intuitive operation while another group demands maximum flexibility and full scripting capabilities. Let's first take a look at the intuitive operation. The basic idea is actually very simple: if the debugger stops at a breakpoint, then the GUI visualizes the application process that triggered the breakpoint.

If you are interested in a different application process, then you simply open the TRACE32 global task list. All tasks executing on the overall system are listed there. You can select the task you want to display in the GUI by double-clicking on the task. The global task list also offers a simple way to set program breakpoints for a specific task. Since the debug symbols are associated with a machine ID and a space ID when the .elf file is loaded, functions and variables can be addressed by name as per usual when debugging.

Maximum flexibility and full scripting capabilities can be obtained using the TRACE32 commands. The extended syntax for these commands are presented above.

Summary

Since Lauterbach has systematically extended the well-known concepts for OS-aware debugging to hypervisor debugging, it will be easy for TRACE32 users to get started with just a little practice.

Currently Supported Hypervisors

KVM	Wind River Hypervisor 2.x
VxWorks 653 3.x	Xen

(more to follow)

Intel® x86/x64 – Tool Update

In January of this year, Lauterbach introduced the new CombiProbe Whisker MIPI60-Cv2. The TRACE32 CombiProbe and TRACE32 QuadProbe now offer the same debug features for the Converged Intel® MIPI60 connector:

- Standard JTAG, Intel® debug hooks with Pmode, and I2C bus
- Merged debug ports (two JTAG chains)
- Intel® Survivability features (threshold, slew rate, ...)

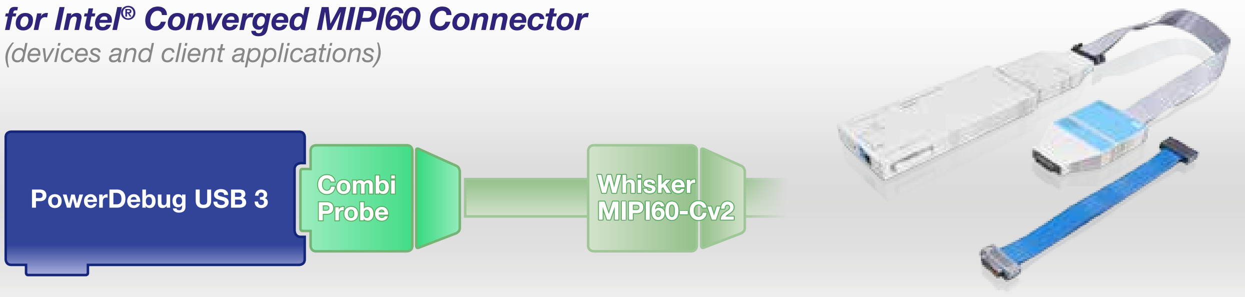
However, these debug tools have different areas of application. The **TRACE32 QuadProbe**, which is expressly designed for server processors, is a dedicated debug tool that enables SMP debugging of hundreds of threads on targets with up to four debug connectors.

The **TRACE32 CombiProbe** with the **MIPI60-Cv2 Whisker**, designed for client as well as mobile device processors, can capture and evaluate system trace data in addition to its enhanced debugging features. Trace capabilities include support of one 4-bit and one 8-bit trace port with nominal bandwidth.

The **TRACE32 CombiProbe** with the **DCI OOB Whisker** is specially designed for debugging and tracing of form factor devices without debug connectors. If the chip contains a DCI Manager, the target and the debugger can exchange debug and trace messages directly via the USB3 interface. The DCI protocol used to exchange messages supports standard JTAG and Intel® debug hooks as well as trace messages for recording system trace information.

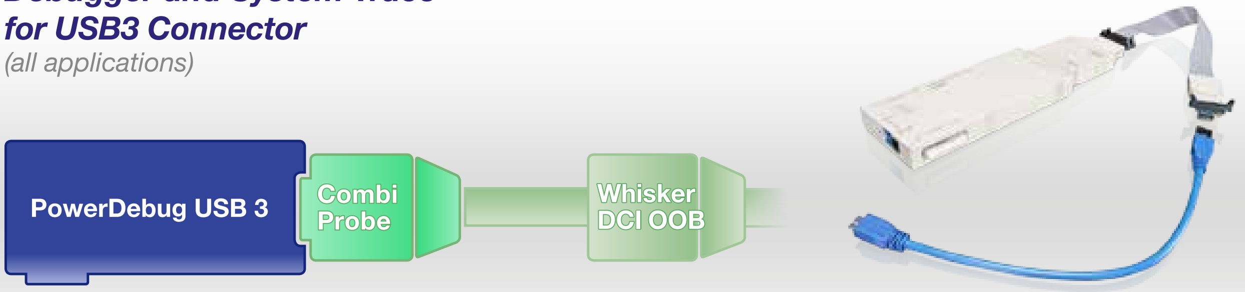
Debugger and System Trace for Intel® Converged MIPI60 Connector

(devices and client applications)



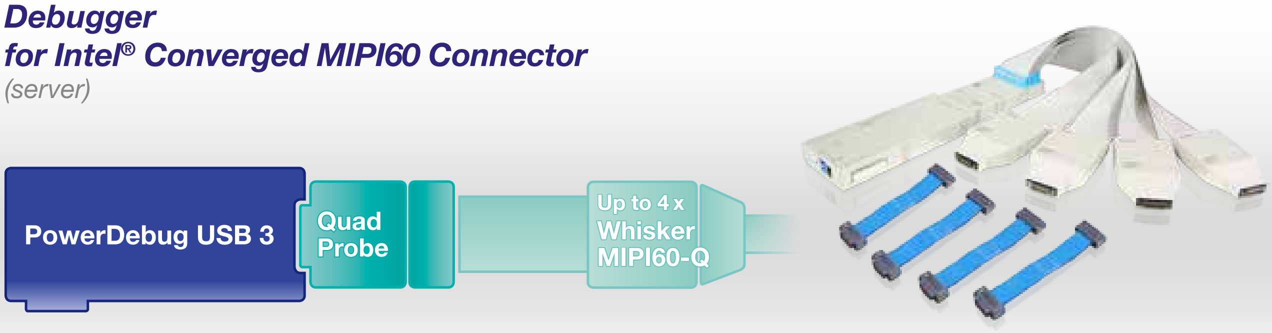
Debugger and System Trace for USB3 Connector

(all applications)



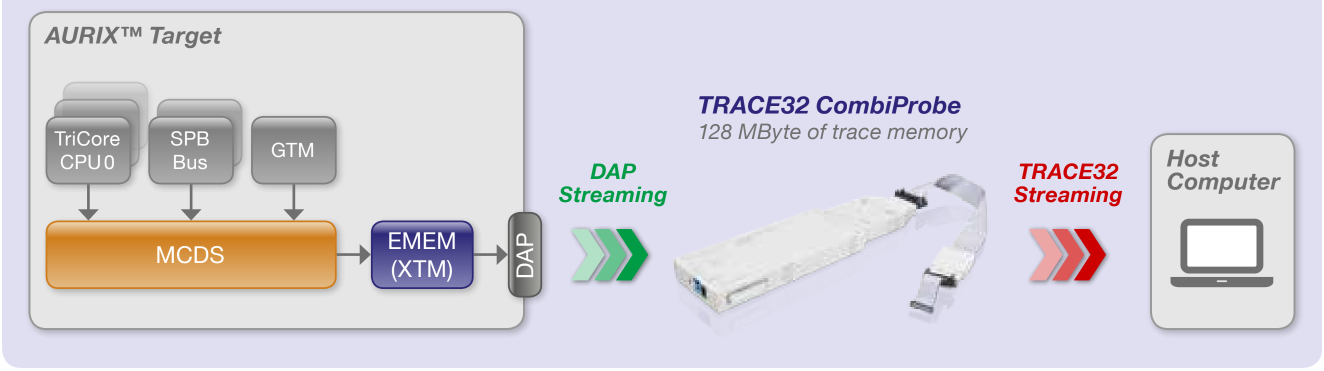
Debugger for Intel® Converged MIPI60 Connector

(server)



TRACE32 CombiProbe TriCore DAP

DAP Streaming and TRACE32 Streaming



Lauterbach has been supplying its new CombiProbe TriCore DAP for the AURIX™ family from Infineon since October 2016. This means TRACE32 now offers comprehensive run-time analyses for all AURIX users whose target hardware does not provide an AGBT interface.

DAP Streaming

The CombiProbe implements a new technology named DAP streaming: The contents of the on-chip trace memory are read while the program is executing and transmitted in full to the 128 MB trace memory of the CombiProbe. For this, the chip must provide a high-speed debug interface. The AURIX™ DAP interface meets the corresponding requirements: DAP frequencies of up to 160 MHz, data rates of up to 30 MB/s. The DAP bandwidth is not large enough to transmit the entire program flow, but extensive analyses can still be performed:

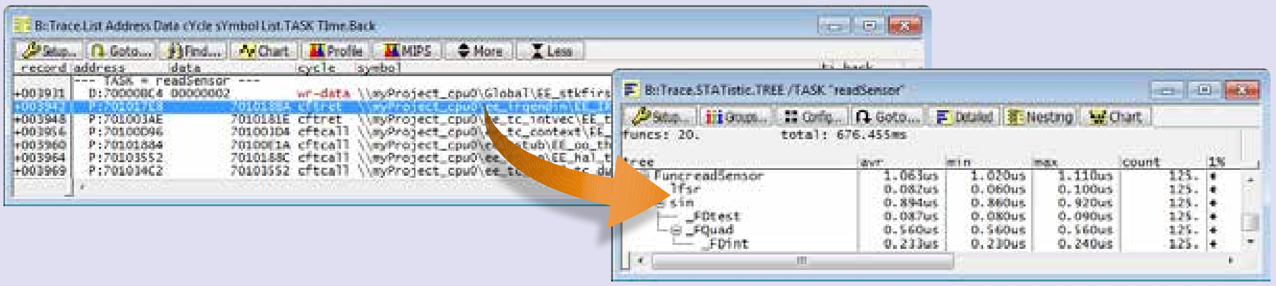
- Function run-time measurements using the Compact Function Trace (CFT). This trace is a special program trace mode in which trace data is only generated for function calls (cftcall) and function returns (cftret). The “Compact Function Trace” diagram below shows an example of the call tree and the run-time details TRACE32 calculated based on this trace data.
- Analysis of the contents of selected variables over time.
- Run-time measurements for tasks, ISRs, and OS services.

TRACE32 Streaming

If the 128 MB trace memory the CombiProbe is not large enough to record all relevant trace data, then it is possible to combine DAP streaming with TRACE32 streaming. TRACE32 streaming transfers the trace data immediately after it is received by the CombiProbe to the host computer and stores it there in a file. This makes it possible to record several TB of contiguous trace data for the purpose of long-term measurements.

You will find more details at:
www.lauterbach.com/8467

Compact Function Trace



```
B::Trace.List Address Data cYcle sYmbol List.TASK TIme.Back
record  address          data          cycle    symbol
        --- TASK = readSensor ---
+003931   D:700008C4 00000002         wr-data  \\myProject_cpu0\Global\EE_stkfirs
+003943   P:70101718           70101884 cftret   \\myProject_cpu0\ee_trgendm\EE_
+003948   P:701003AE           7010181E cftret   \\myProject_cpu0\ee_tc_intvec\EE_t
+003956   P:70100D46           70100ID4 cftcall  \\myProject_cpu0\ee_tc_context\EE_
+003960   P:70101884           70100E1A cftcall  \\myProject_cpu0\ee_stub\EE_oo_th
+003964   P:70103552           7010188C cftcall  \\myProject_cpu0\ee_   \EE_hal_t
+003969   P:701034C2           70103552 cftcall  \\myProject_cpu0\ee_tc_   _dw
```

```
B::Trace.STATistic.TREE /TASK "readSensor"
funcs: 20.          total: 676.455ms

tree                         avr        min        max        count    1%
 FuncreadSensor             1.063us    1.020us    1.110us     125.    •
   lfsr                     0.082us    0.060us    0.100us     125.    •
   sin                      0.894us    0.860us    0.920us     125.    •
    ├── _FDtest             0.087us    0.080us    0.090us     125.    •
    └─⊖ _FQuad              0.560us    0.560us    0.560us     125.    •
        └── _FDint          0.233us    0.230us    0.240us     125.    •
```

ARTI – AUTOSAR Run-Time Interface

The new ARTI standard will be specially designed to meet the current requirements of the automobile industry for OS-aware debugging and tracing. Lauterbach, as an official AUTOSAR development partner, is active in the design of this standard. Publication of the standard is planned for the beginning of 2018.

The ORTI standard, which has been used throughout the automobile industry since 2003, helps thousand of developers to debug and profile their AUTOSAR systems. The standard has aged gracefully over the years and requires updating to meet current needs.

Goals

New methods for software development, multicore and multi-ECU systems, and increasing requirements on the validation of real-time critical systems – the new standard needs to cover all of this. Many of the new debug, trace, and profiling features that will be added to the ARTI standard are already in use today as proprietary solutions. This means the functionality has already proven itself. What is missing are standardized interfaces between the various tools used in the development process. The following is an example.

Exporting Trace Data

Since 2014, Lauterbach has been working closely with various manufacturers of tools used for the validation and optimization of automotive software. TRACE32 exports its real-time trace data recorded, which is then loaded into the external tool and comprehensively analyzed. So what is currently missing?

1. The ORTI file created by the build tool only contains information on the tasks, the OS services, and the ISRs, but no information on when the tasks were started or terminated, as well as no information on the runnables. Before exporting, this missing information must be added manually in TRACE32.
2. There is no standardized format for trace data exports. This means the external tools must be able to read the proprietary TRACE32 format.

The new ARTI standard will close both of these gaps. The ARTI file provided by the build tool will contain information on all AUTOSAR objects while simultaneously standardizing the export of trace data.

Summary

Since all important tool manufacturers as well as tool users work together on drafting the new ARTI standard, it will surely be just as successful and long-lived as its predecessor.