Hypervisor Debugging User Guide

Release 09.2024

MANUAL

Hypervisor Debugging User Guide

Hypervisor Debugging User Guide

Introduction

The most important objective of the TRACE32 Hypervisor Awareness is a 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 virtual machines, plus the current state of the hypervisor and of the real hardware platform.

NOTE:	**This manual is still under construction.**

Features

The TRACE32 Hypervisor Awareness supports the following debugging features:

- It provides the debugger with all of the hypervisor's information running on the hardware platform. This allows debugging of all components at the same time, be it the hypervisor itself, the OS of a guest or a process/library within a guest OS.

- The TRACE32 Hypervisor Awareness will provide a list of all VMs started and assigns each VM a number, the machine ID. The machine ID is a unique identifier that is used by TRACE32 and appears as an address extension; a concept similar to the space ID which is already familiar to many TRACE32 users. Using this machine ID, virtual addresses are bound to a specific machine.

- Symbols are loaded to a specific machine. That is, whenever you access a symbol, the debugger automatically knows which VM to address and how to access the function/variable. When debugging in High Level (e.g. C), you simply access a function of variable by its name, the debugger calculates internally the correct access method.

- In addition to the Hypervisor Awareness, you can load an OS Awareness for each guest OS individually. This allows to display the task list of each guest at the same time. A tree view including all tasks of all VMs provides an overview of the overall system.

- The TRACE32 main menu bar is extended to allow quick access to the most important features of the hypervisor support.

- To get access to the whole system, the Hypervisor Awareness makes use of the built-in TRACE32 MMU awareness. That means, the debugger knows how to resolve a virtual address to a physical address. By doing the MMU table walk itself, it can access data and code of the

hypervisor, any guest OS and the processes running, even if they are currently not being executed on a real core. You can inspect the MMU translations of an OS and the hypervisor to debug translation problems.

Intended Audience

Developers who want to:

• Debug the boot phase of a system including the hypervisor start-up

• Debug the hypervisor itself, its second-stage MMU translation and/or the exception routines

• Debug bare-metal or OS applications running in a VM within a virtualized system

• Debug the kernel or drivers of an OS running in a VM within a virtualized system

• Debug processes that run within a virtualized OS

• Debug communication channels between VMs

This document assumes as your background:

• You should be familiar with TRACE32 and its basic concepts

• If an OS is used as a guest, you should be familiar with the OS Awareness

• You should be familiar with the concepts of the hypervisor in use

• You should be familiar with the target hardware (CPU, cores, periphery)

• You should be familiar with PRACTICE scripting (*.cmm)

Prerequisites

The following is a checklist summarizing technical prerequisites that must be fulfilled *before* you can continue with this document:

OK	Technical Prerequisites
	TRACE32 is correctly configured for the target.
	Debug symbols are available for the hypervisor.
	Debug symbols are available for all guests and applications that should be debugged.
	A Hypervisor Awareness for the hypervisor in use is available.
	An OS Awareness for the guest OS in use (if to be debugged) is available.
	All watchdogs in hardware, hypervisor, and guests are disabled.
	The components to be debugged are compiled with minimum optimization (if possible).
	External protocols are insensitive to big time gaps.

Contacting Support

Use the Lauterbach Support Center: https://support.lauterbach.com

- To contact your local TRACE32 support team directly.

- To register and submit a support ticket to the TRACE32 global center.

- To log in and manage your support tickets.

- To benefit from the TRACE32 knowledgebase (FAQs, technical articles, tutorial videos) and our tips & tricks around debugging.

Or send an email in the traditional way to support@lauterbach.com.

Be sure to include detailed system information about your TRACE32 configuration.

1. To generate a system information report, choose **TRACE32** > **Help** > **Support** > **Systeminfo**.

| NOTE: | Please help to speed up processing of your support request. By filling out the system information form completely and with correct data, you minimize the number of additional questions and clarification request e-mails we need to resolve your problem. |

2. Preferred: click **Save to File**, and send the system information as an attachment to your e-mail.

3. Click **Save to Clipboard**, and then paste the system information into your e-mail.

Related Documents

- **Processor Architecture Manuals** (debugger_*<arch>*.pdf)

- **OS Awareness Manuals** (rtos_*<os>*.pdf)

- Hypervisor Awareness Manuals (hv_*<hypervisor>*.pdf)

Restrictions

- Up to 30 guest machines are supported.

- On systems with hardware timers that continue running although the CPU is stopped, it is not always possible to enter and leave debug mode without side effects on the system's behavior.

Terms, Abbreviations and Definitions

Awareness

An awareness for TRACE32 is an OS-specific or hypervisor-specific extension which can be loaded into TRACE32 at run time. With the help of a loaded awareness, TRACE32 can determine the current state of an OS or hypervisor and extract all necessary information about tasks or guest machines.

An awareness is an extra software module written in C and built to a loadable binary using the TRACE32 Extension Development Kit (EDK). LAUTERBACH provides awareness files for a large number of hypervisors and operating systems. Users can also write their own awareness with the EDK, which is provided to customers upon request.

```
                        ┌──────────────────┐
                        │    Awareness     │
                        └──────────────────┘
                                 │
              ┌──────────────────┴──────────────────┐
   ┌──────────────────────┐            ┌──────────────────────┐
   │  Hypervisor Awareness │            │     OS Awareness     │
   └──────────────────────┘            └──────────────────────┘
```
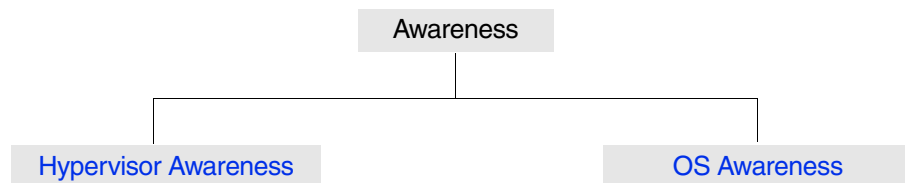
Context

The context refers to the execution context of the code which is currently being executed on a physical core or a virtual VPU (VCPU). The execution context comprises the state of all CPU and MMU registers which determine the previous and the next execution steps. This includes, but is not limited to, the CPU's general purpose registers, the state registers, the stack frame registers and the MMU registers which define the configuration of the currently applied address translation.

Core

TRACE32 defines a core as an execution unit within a CPU, with a dedicated register set.

CPU

A CPU is a physical unit that contains one or more cores, and may contain further logic (e.g. peripheral blocks).

Extension

An extension is an external module provided by Lauterbach or written by users who want to add custom features to the TRACE32 software. The custom features can be new commands, windows, PRACTICE functions, TRACE32 OS Awarenesses, and TRACE32 Hypervisor Awarenesses.

Extensions in TRACE32 are controlled with the command group **EXTension**.

Hypervisor

A piece of software or hardware that is able to run virtual machines.

Hypervisor Awareness

An extension loaded to TRACE32 that allows specific operations on the hypervisor, such as displaying and working with guest machines.

Extensions in TRACE32 are controlled with the command group **EXTension**.

Machine

A machine is a TRACE32 term for a physical or virtual environment for an operating system (OS).
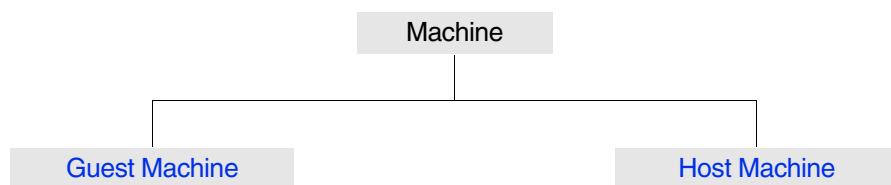
Host Machine

The host machine is the physical computer environment which runs the hypervisor software. A host machine comprises the computer hardware and all software which is not running under control of the hypervisor. Term is used for systems which involve virtualization.

The counterpart to host machine is guest machine.

Guest Machine (synonym: virtual machine, VM)

A guest machine is a runtime environment which is running under the control of a hypervisor. A guest machine consists of one or more VCPUs and can run a complete operating system.

Term is used for systems which involve virtualization.

The counterpart to guest machine is host machine.

NOTE:	The synonym *virtual machine* is used in contexts where it is necessary to implicitly or explicitly distinguish between virtual machines and physical machines.
	For an example, see hypervisor.

Machine ID

A *machine ID* is a numeric identifier which extends a logical address and intermediate address in TRACE32 or can be used together with the option **MACHINE** in some TRACE32 commands. The purpose of a machine ID is to identify guest machines within a system that is using a hypervisor to run multiple virtual machines.

<machine_id>	**Parameter Type**: Decimal or hex value. **Range**: `0x0` <= machine ID < `0x1F` Machine IDs are displayed, for example, in the **mid** column of the **TASK.List.MACHINES** window as decimal values (`1.`, `2.`, etc.)

In TRACE32, the machine ID clearly specifies which virtual machine (a guest machine or the host machine) an address belongs to:

- The machine ID 0 (zero) is always associated with the host machine running the hypervisor.

- All the other machine IDs >= 1 are associated with the guest machines.

Format of addresses with machine IDs:

In the TRACE32 address format, the machine ID is always in the leading position, directly after the access class specifier. The machine ID is followed a triple colon (`:::`) to separate the machine ID from the remaining parts of an address. The format of a TRACE32 address containing a machine ID looks like this:

- Without space ID:

  <access_class>:<machine_id>:::<address_offset>

- With space ID:

  <access_class>:<machine_id>:::<space_id>::<address_offset>

Examples:

- Without space ID:

  - G:0x1:::0x80000000

  - 0x2:::0xA0000000

- With space ID:

  - G:0x3:::0x020A::0x80000000

  - G:0x0:::0x0::0x4000C000

  - 0x2:::0x170::0x1F000000

Notes:

- Machine IDs can only be used if a TRACE32 Hypervisor Awareness is loaded with the command **EXTension.LOAD**.

- Use command **SYStem.Option.MACHINESPACES ON** to enable machine IDs in TRACE32.

Memory Management Unit (MMU)

The MMU is a unit inside the CPU core that translates logical addresses to physical addresses. You can access and view the MMU in TRACE32 with the commands of the **MMU** command group.

MMU Space

MMU space is a TRACE32 term for an MMU-mapped memory space (aka "address space"). In operating systems (OSs) usually called "process".

TRACE32 uses the term *MMU spaces* if a target system with an MMU uses multiple equivalent page tables **within the same zone (= CPU mode)** to run independent software parts (such as processes) on virtual addresses. Each page table defines an individual address translation for the software part running on the page table. The page tables are usually maintained by an owner system - usually an operating system. TRACE32 labels the address space which is defined by the page table as one MMU space.
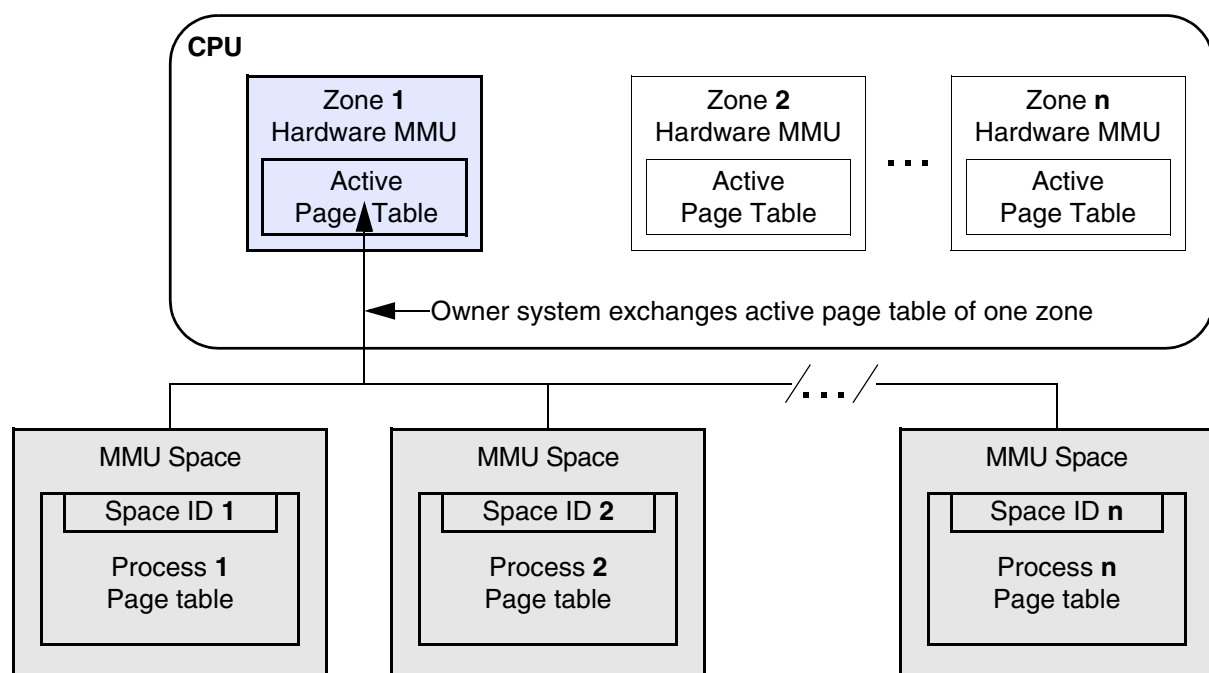
Standard use cases for MMU spaces are operating systems, such as Linux, where processes run on identical virtual addresses and the kernel configures an individual page table for each process.
Often, the number of valid MMU spaces is dynamic: during runtime new MMU spaces may be created or removed by the controlling software - for example if an OS creates or terminates processes.

In TRACE32, the MMU spaces and their identifiers, the space IDs, are enabled with the command **SYStem.Option.MMUSPACES**.

For more information, see space ID.

OS

An "Operating System" (OS) is a piece of software that is able to schedule and dispatch several threads of code to the cores. If each thread has its own execution environment (especially its own register set), TRACE32 calls them tasks. An OS may also perform a memory management providing virtualized memory to the tasks. Several tasks that share the same virtualized memory space form a TRACE32 MMU space.

OS Awareness

An extension loaded to TRACE32 that allows specific operations on the RTOS, such as displaying and measuring the tasks.

Extensions in TRACE32 are controlled with the command group **EXTension**.

Process

In OSes usually a collection of threads that share the same virtualized memory space. In this sense, the TRACE32 term "MMU space" maps to a process.

Note: In some RTOS (esp. ARINC based OSs) a "process" is defined as an execution context. In this sense, it maps to the TRACE32 term "task".

RTOS

Real Time Operation System - equivalent to kernel.

Task

TRACE32 term for an execution unit with its own register context. In OSes usually called "thread" or "task". Some RTOS (esp. ARINC based OSs) also call this "process".

Thread

In OSes usually an execution unit. If it has its own register context, it maps to the TRACE32 term "task".

VCPU

A virtual core used by a virtual machine. If the virtual machine is currently running, it maps to a physical core.

Virtual Machine, VM

See Guest machine.

In a Nutshell

This chapter describes the basic concepts of Hypervisor Debugging in TRACE32 in a very short and condensed way. It is intended to give a quick overview and serve as a reference. The following chapters will explore the Hypervisor Debugging in much more detail.

TRACE32 features for hypervisor debugging

Basics

TRACE32 uses a *machine ID* to identify virtual machines:

Host:	machine ID = 0
Guest 1:	machine ID = 1
Guest 2:	machine ID = 2
...	

In TRACE32, *machine IDs* are enabled with the command:

SYStem.Option.MACHINESPACES ON

TRACE32 addresses are extended by a machine ID:

```
              D :           0x2 :::     0x03A5 : 0x00401EC0
  <access class> : <machine_id> ::: <space_id> : <offset>
```

Load a Hypervisor Awareness:

Example: Loading XEN Awareness:

```
  EXTenstion.LOAD xen.t32 /MACHINE 0 /NAME xen /ACCESS H:
```

The Hypervisor Awareness reads out the current state of the hypervisor and guests.

Load a Hypervisor Menu:

Example: Loading XEN menu:

```
  MENU.ReProgram xen.men
```

The TRACE32 PowerView user interface will be extended by a Hypervisor specific menu.

Load an individual OS Awareness for each guest machine.

Example: guest 1 running Linux and guest 2 running FreeRTOS:

```
EXTension.LOAD linux.t32    /MACHINE 1 /NAME Linux
EXTension.LOAD freertos.t32 /MACHINE 2 /NAME FreeRTOS
```

TRACE32 gets all information about the system from the combination of Hypervisor Awareness plus OS Awareness.

NOTE:	The awarenesses usually need the symbols of the RTOS/kernel. Please ensure that the symbols are loaded correctly as described in the individual hypervisor / OS Awareness manuals.

When loading an awareness for a guest OS, the TRACE32 PowerView user interface will be extended by guest specific menus corresponding to the selected guest names.

PowerView layout with Hypervisor Awareness:

Virtualization of physical memory

If the hypervisor uses a hardware-based virtualization then there is a two-stage memory management.

- The application uses the "guest logical address",
  e.g. "D:1:::", "N:1:::" (ARM), "G:1:::" (PowerPC / x86)

    - The "first stage MMU" (managed by the guest OS) translates this to a "guest physical address."

- The guest OS uses the "guest physical address", aka "intermediate address", i.e. "I:1:::"

    - The "second stage MMU" (managed by the hypervisor) translates this to absolute physical.

- The host machine uses "absolute physical address, i.e. "A:"
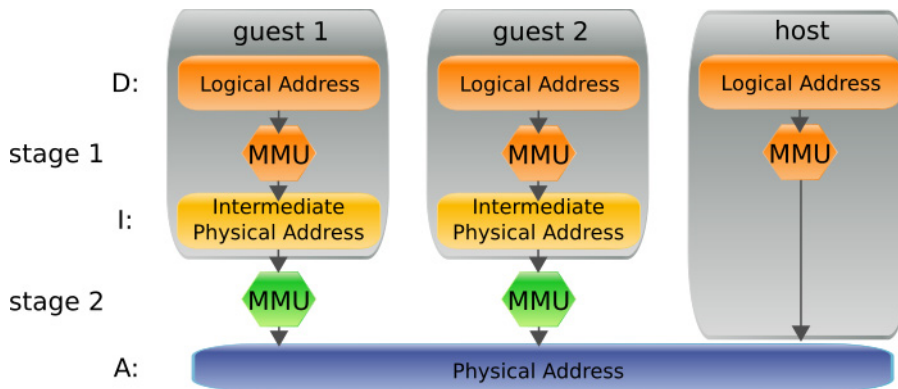
The guest "thinks" it has physical memory with physical addresses, but this is *virtualized* physical memory with guest physical address (aka intermediate address).

Each process has its own stage 1 translation, owned by the guest OS.
Each guest has its own stage 2 translation, owned by the hypervisor.

The "current view" is the memory seen by the core with the current MMU settings, which is only the "current process". But TRACE32 can do a *full* translation for *any process and any machine*:

- Virtual address: `D:<machine_id>:::<space_id>:<offset>`

    - MMU stage 1 translation by OS Awareness

- Intermediate address: `I:<machine_id>:::<offset>`

    - MMU stage 2 translation by Hypervisor Awareness

- Physical address: `A:<offset>`

Symbols

The debug symbols for each machine are kept separate and contain the machine ID. Hence, it is very convenient to work with symbols, as they "know" to which machine they belong and you don't need to specify the machine explicitly.
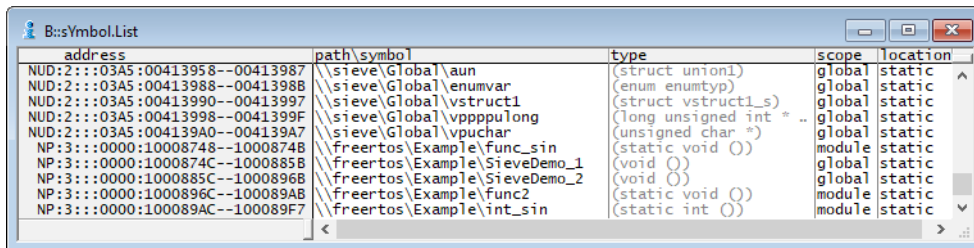
Examples: Loading symbols to machines

```
; load symbols for hypervisor:
Data.LOAD.Elf xen-syms.elf H:0:::0 /NoCODE

; load kernel symbols for guest 2 running Linux:
Data.LOAD.Elf vmlinux N:2:::0 /NoCODE /NoClear

; load application symbols for process running in linux:
; (usually done by symbol autoloader)
Data.LOAD.Elf sieve N:2:::0x03A5:0 /NoCODE /NoClear

; load symbols for guest 3 running FreeRTOS:
Data.LOAD.Elf freertos.elf N:3:::0 /NoCODE /NoClear
```


Machine specific commands

Example: accessing memory

```
; Use symbol name, contains machine ID and space ID:
List func0

; Extend with current machine ID and space ID:
List 0x401EC0

; Use space ID 0x3A5, current machine ID will be extended:
List 0x3A5:0x401EC0

; Specify machine ID and space ID:
List NUX:2:::0x3A5:0x00401EC0
Data.dump NUD:2:::0x3A5:0x401EC0
```

Example: setting breakpoints

```
Break.Set func0
Break.Set P:2:::0x03A5:0x00401EC0
Break.Set myVar /Write
Break.Set D:3:::0x1044 /Read
```

Example: viewing stack frame

```
; current stack frame
Frame
; frame of first found task named "sieve" in any machine
Frams /TASK "sieve"
; frame of task "sieve" in machine "FreeRTOS":
Frame /TASK "FreeRTOS:::sieve"
```
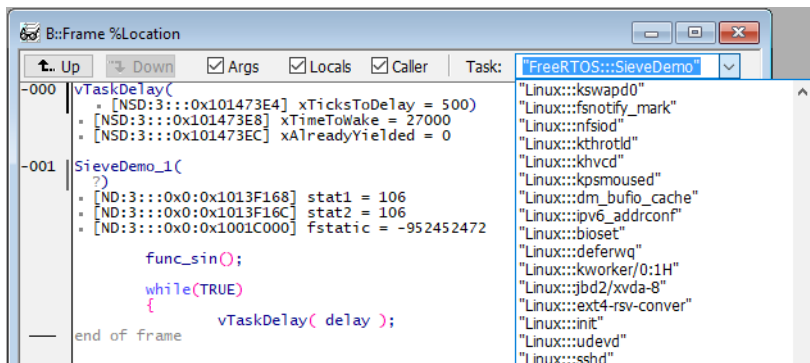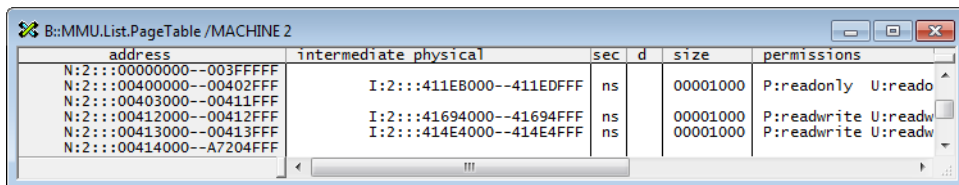

Example: viewing register set

```
Register.view              ; current register set
Register.view /MACHINE 2   ; register set of machine ID 2
```

Example: MMU page table analysis

```
MMU.List PageTable          /MACHINE 2
MMU.List IntermedPageTable  /MACHINE 3
MMU.List TaskPageTable      "FreeRTOS:::sieve"
MMU.DUMP PageTable          /MACHINE 2
MMU.SCAN PageTable          /MACHINE 2
```


Example: Address translation configuration
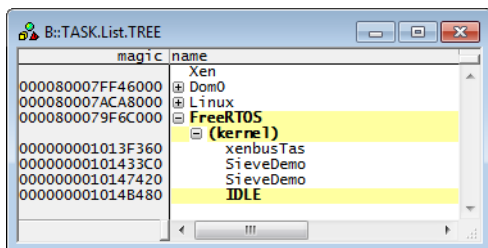
```
TRANSlation.Create D:2:::0x10000++0xffff I:2:::0x530000
TRANSlation.Create I:2:::0x500000++0x1fffff A:0x20000000
TRANSlation.COMMON D:2:::0xC0000000--0xFFFFFFFF
MMU.FORMAT STD /MACHINE 2
```

Example: Hypervisor specific commands

```
; list all tasks
TASK.List.TREE
TASK.List.TASKS

; list all processes
TASK.List.SPACES

; list all machines
TASK.List.MACHINES
```

Example: listing all cores with current task

```
CORE.List
```
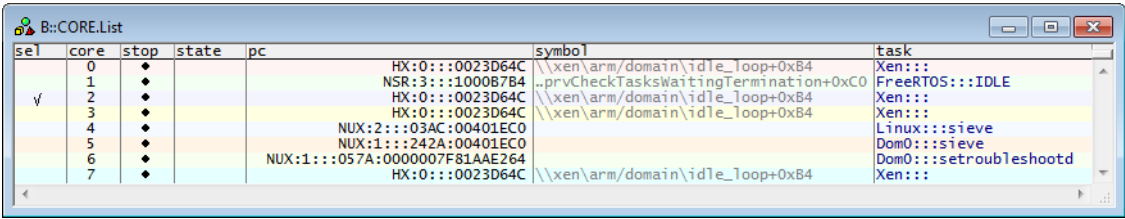

Example: using guest OS specific commands

```
; schema: TASK.<awareness>.<command>
TASK.Linux.Process
TASK.FreeRTOS.TaskList
```
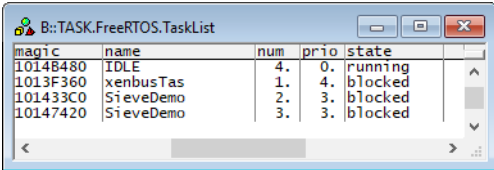
About Hypervisors and Virtualization

This chapter gives general information about hypervisors and virtualization, as far as this is necessary to understand its impact to debugging. It will not go into debugging yet, this is subject to the next chapter.

Types of Hypervisors

A "hypervisor" is a piece of software, firmware or hardware that creates and runs virtual machines. The real piece of hardware (could be a desktop PC or an embedded system) that runs a hypervisor and virtual machines is referred to as a "host machine", whereas a virtual machine is referred to as "guest machine". Each guest machine can run its own OS, which means, a hypervisor can run different OSs on a single host machine in parallel.
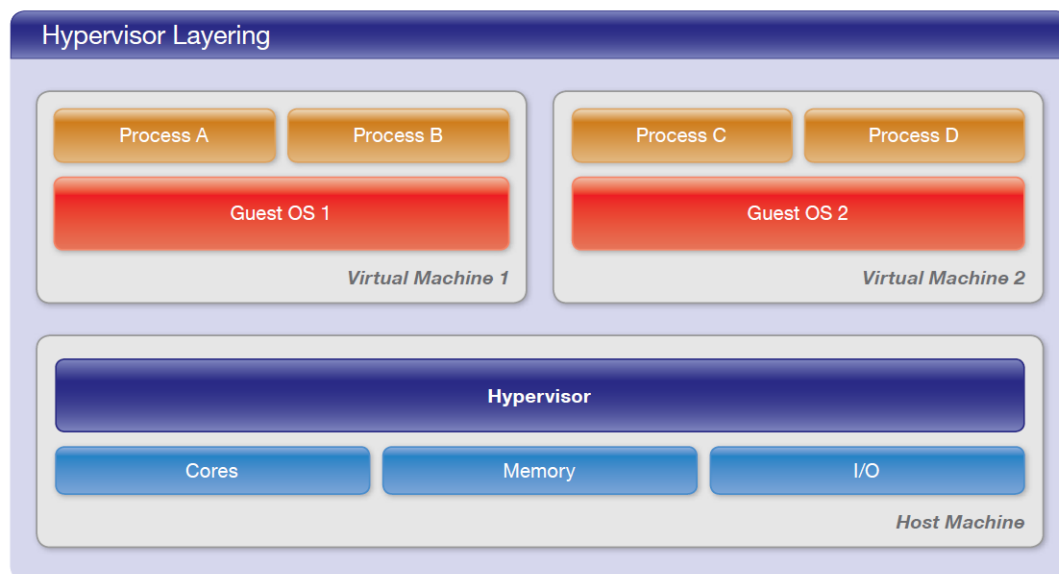
There are two sorts of hypervisors:

- Software-only hypervisors use a so-called "paravirtualization".
  They do not need any hardware support for virtualization, instead all protection parts (MMU, hardware access) is moved from the guest into the hypervisor. This means that guest OSs using such functionality need to be adapted for paravirtualization.

- Hypervisors with hardware support.
  These hypervisors utilize a dedicated hardware part of the CPU for virtual machine management. The guest OS usually does not know that it runs within a virtualized environment, and does not need to be adjusted to run as a guest. In fact, the OS "thinks" it runs on its own hardware.

Hypervisors come in two types:

- **Type 1 Hypervisor**, also known as "bare metal hypervisor"

  These hypervisors run as their own piece of software independently of any guest OS with a pure "guest" scheduler. There may be a dedicated guest for managing the hypervisor, but in fact it could run without it. Xen is a typical example, with the so-called "DomU" as managing guest.

- **Type 2 Hypervisor**

  A type 2 hypervisor is running within an OS. Typically such a hypervisor is a special process or module within the host OS managing and scheduling the guests. It can not run standalone. Examples are KVM (as module) or QEMU (as process).

Virtualization of Resources

The so-called host machine runs the hypervisor and controls the "real" physical memory, interrupts and periphery. It is responsible to schedule these resources to the guests and monitor the guest behavior.

A guest machine runs a bare metal application or an operating system. Each guest only sees virtualized memory, even if it "thinks" it has physical memory and uses it as a such. That's why this is also often referred to as "guest physical" addressing. Also all interrupts and periphery will be shown virtualized to the guest. A hypervisor may give exclusive and direct access of interrupts and/or periphery to a dedicated guest. This is managed again by the hypervisor and transparent to the guest, the guest does not know if it "owns" the hardware, or if it was given exclusive access, of if it accesses only virtual hardware.

A guest OS such as Linux has its own memory management, including virtual addressing. It translates a logical address via the MMU to a physical address. In the case, where this OS runs on top of a hypervisor, these are then called "guest logical addresses" and "guest physical addresses", where the guest physical address is no longer a real physical address. This address translation is then the "first stage". The hypervisor then does a "second stage" address translation, utilizing a (hardware based) second stage MMU. This MMU

then translates the "guest physical address" to an "absolute physical address", which in now really physically mapped to the memory. This means we have two full translation stages, one controlled by the guest and one controlled by the hypervisor.
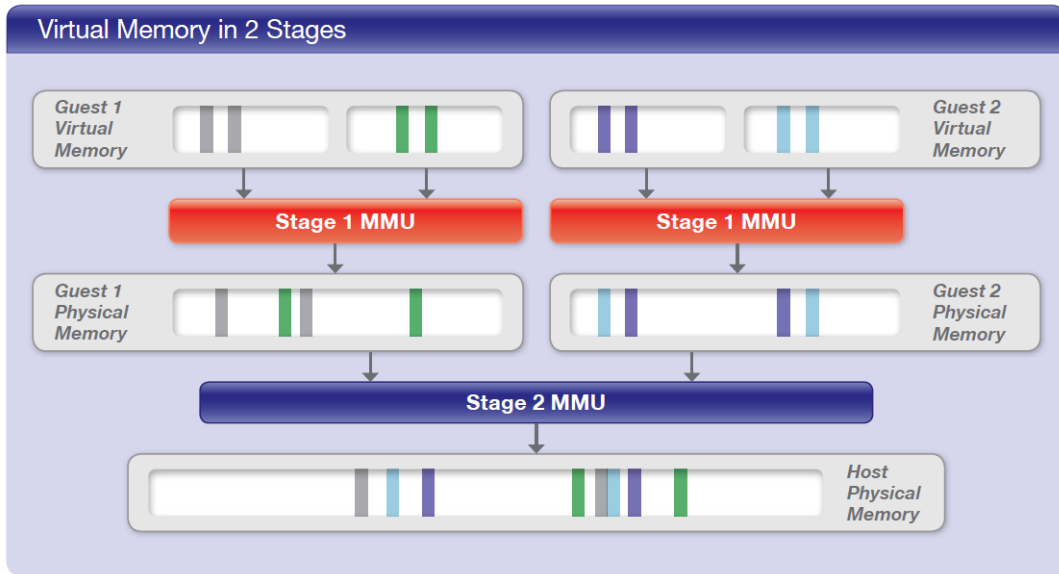
Physical Cores and Virtual CPUs

Hypervisors typically run on a multi-core system. The hypervisor is responsible to schedule the cores to the various guests. This can be done in two ways: either by a strict core binding, or by virtual CPUs (VCPUs).

If a guest is bound to a fixed core (or a set of cores), then these cores are exclusively used by this guest, and the hypervisor does not need to do any further virtualization of these cores. If you a looking on a bound physical core, you know from the binding, which guest is running thereon. No guest switch happens, the registers will not be swapped an therefore there's no need to store its contents in a VCPU memory area. Such a system looks like (and may be debugged as) a traditional AMP system, just with the exception that a hypervisor monitors the run. To give an example, "Jailhouse" is a hypervisor using this mechanism.

On a full-blown hypervisor, however, there is usually no core binding. Instead, the cores of the guests are virtualized to "virtual CPUs" (VCPUs). The number of VCPUs within a guest does not correlate to the number of physical cores, indeed a guest may have more VCPUs than available physical cores. The hypervisor is responsible to schedule the VCPUs of all guests onto physical cores. This means that VCPUs may suspend or migrate to another physical core. If a VCPU is suspended, its register content is stored within the hypervisor in a dedicated memory area, to be restored as soon as the VCPU continues on any other core. This is an SMP architecture, and must be debugged in SMP mode. You never know, on which physical core your guest or application will run.

What to know about Hypervisors in TRACE32

This chapter describes how hypervisors are managed in TRACE32. It will show the GUI elements when debugging hypervised systems, the management of virtual addresses and the handling of debug symbols. At the end of this chapter, you should know all elements that are relevant to debug a system using a hypervisor with TRACE32.

Remember that there are several schedulers within a hypervisor system. First, the hypervisor schedules the various guests (actually their VCPUs) onto the physical cores. Second, *each* guest OS schedules its processes and threads onto the VCPUs. This means a specific process may run on any VCPU of the guest OS and on any physical core, and even may be migrate between VCPUs and physical cores. Indeed it may also be "running" on a VCPU but suspended in the hypervisor, i.e. having no physical core. Therefore, when debugging, you never know "where", and if, the process runs that you want to debug at a specific time.

Run-Mode vs Stop-Mode Debugging

Basically, there are two modes of operation for debugging: "run-mode" debugging and "stop-mode" debugging. While run-mode debugging in hypervisor systems is fairly easy, stop-mode debugging faces some extra challenges.

Run-Mode Debugging

When using run-mode debugging, the debugger software on the debug host connects to a piece of software, the "debug agent", on the target. The debug agent (e.g. the gdbserver or gdbstub) does the actual debugging, like setting breakpoints, reading variables, single stepping etc. The debug host (e.g. gdb or TRACE32 GDB front-end) controls the debug agent and visualizes the results. Both communicate over an arbitrary communication channel, usually either a serial or a TCP/IP connection is used. As the target continues to run, at least to serve the debug agent, this is called "run-mode".

Let's assume you want to debug process number 3 in guest number 2. The debug agent will run in the environment of process 3, i.e. in guest 2. This means that the debug control already has full access to the software to be debugged, as it sees the same environment. However, it can *only* be attached to software running in this environment, i.e. it is bound to guest 2 and does not have access to other guests or the hypervisor.

However, to debug, all you have to do is to establish a connection between the debug host and the debug agent. E.g. set up a serial connection to the guest by relaying it through the hypervisor, or create a virtual Ethernet connection to the guest, and you're able to do run-mode debugging. Both, the debug host and the debug agent even don't recognize that there is a hypervisor in between.

To sum up: Run-mode debugging is relatively easy to set up. But this is limited to a single process (or a set of dedicated processes). You won't see other guests or the hypervisor. And you won't be able to monitor the context switching of processes and guests. Debugging into interrupts or parts of the kernel is also not possible, or at least very limited - the debug agent needs to keep running and communicating.

Stop-Mode Debugging

In the case of stop-mode debugging, e.g. when debugging with JTAG capabilities, there is no special software running on the target. Instead, the debugger takes control over the hardware, i.e. the physical CPU. When halting the system, e.g. by running onto a breakpoint, the *complete* system is halted, i.e. all physical cores that are under control of the debugger will stop executing any code. The core is "frozen", even interrupts will not be served.

The big advantage of this mode is that you gain control over *everything* that is running on this system. As the debugger has access to the full memory, it can inspect, analyze and visualize the complete state. With the help of a Hypervisor Awareness and OS Awareness, the debugger can evaluate, which guests and which processes are available on the target. It can show any function of any process, OS or even the hypervisor, and can set breakpoints therein at any time. You're able to view global variables of any process, the stack frame of any thread, the register set of any thread or VCPU and the MMU page tables for active and inactive processes or guests. If necessary, the stop-mode allows to single step through all layers, even through interrupts, observing the changes of the CPU modes. E.g. if a message is sent from process A in guest 1 to process B in guest 2, it is possible to single step all the way through process A --> kernel guest 1 --> hypervisor --> kernel guest 2 --> process B.

As already mentioned, the debugger needs to be "aware" of the hypervisor and the guest OSs. This means that an awareness for those must be available and configured. Additionally, some settings about the MMU layout must be given to enable a full address translation of all elements. This leads to a somewhat complex setup that needs to be adjusted to match to the specific target application.

To sum up: Stop-mode debugging gives you access to everything that runs in your target, be it hypervisor, guest OS kernel or processes, all at the same time. As a drawback, the configuration of the debugger becomes a little bit complex to suit each specific system.

Machines in TRACE32

TRACE32 handles both, the physical host system running the hypervisor and the virtual guests, as "machines". A machine is identified by its machine ID. Machine ID 0 is always referring to the hypervisor, guest machines get an ID not equal to zero. The ID for a specific guest is usually provided by the Hypervisor Awareness and is closely related to the guest ID used by the hypervisor itself. The machine IDs do not need to be consecutive.

NOTE:	Some hypervisors number their guests starting with 0 for the first guest (e.g. Dom0 in Xen). However, in TRACE32 the machine ID 0 is reserved for the hypervisor itself. In these systems, the machine ID is (typically) off by one to the guest ID in the hypervisor (e.g. Dom0 has machine ID 1).

Machine IDs are enabled with the command **SYStem.Option.MACHINESPACES ON**, virtual addresses are then extended by the machine ID. This is similar to TRACE32's concept of handling user processes (**SYStem.Option.MMUSPACES ON**).

TRACE32 also allows to give names to machines, either provided by the Hypervisor Awareness, or provided manually by commands. For better visibility, some windows will then show the machine name rather than the machine ID.

As an identifier for a specific machine, the machine ID serves in virtual addresses to uniquely identify the addresses (and their translation). Some commands also accept the machine ID or machine name as an option, to direct the functionality of this command to a specific machine. See also **"Menus and Commands"**, page 28.

To get knowledge about the machines in the system, the debugger needs the Hypervisor Awareness, loaded with the command **EXTension.LOAD**, e.g.:

```
EXTension.LOAD xen.t32 /MACHINE 0 /NAME Xen
```

With the Hypervisor Awareness, the debugger evaluates the available virtual machines, their IDs, possibly their names and especially their MMU translation settings. Symbol information of the hypervisor is necessary for the awareness to work.

Guests in TRACE32

Each guest (or "virtual machine") in a hypervisor system is handled in TRACE32 as a separate "machine" with its own machine ID. The machine IDs of guests are provided by the Hypervisor Awareness (usually related to the guest ID in the hypervisor) and are always greater than zero. See also the previous paragraph. Each guest can hold its own set of symbols, mapped to the according machine ID This way a guest can be debugged as it would run on a real machine.

There are several kinds of guests:

Bare Metal

A "bare metal" guest contains an application written to run on the physical system without any OS support. When a Hypervisor Awareness is active, you can simply load the symbols of the bare metal application to the machine ID of the guest. When accessing these symbols, the debugger takes care of resolving these to the correct guest. See the chapter "Symbol Management" later in this document.

RTOS (without memory spaces)

If the guest contains an RTOS that does not utilize the MMU for virtual memory management, you can add an OS Awareness to this guest. This means, you can inspect the OS resource (e.g. tasks) of this guest, even if it is not active. This OS Awareness is loaded *additionally* to the Hypervisor Awareness, and works only on this guest. If you have several guests with several, same or different RTOS running, you can load an awareness for each guest separately. E.g. if you run FreeRTOS within the guest with machine ID 2:

```
EXTension.LOAD freertos.t32 /MACHINE 2 /NAME FreeRTOS
```

Rich OS (with memory spaces)

A guest can contain a full blown operating system like e.g. Linux. In this case, the guest OS uses its own memory management, providing memory spaces (usually called "processes") and tasks (aka threads). In this case, the debugger adds a so-called space ID to the virtual address that specifies the process to which it belongs. As the space ID is part of the guest, it is only valid within the appropriate machine ID.

In order to correctly decode the MMU translation of the guest, it is mandatory to declare the MMU layout to the debugger and to load the appropriate OS Awareness. Again, TRACE32 is able to handle different guest OSs simultaneously.

CPU Modes

CPUs that provide hardware virtualization also provide different privilege levels, sometimes called "modes" or "zones":

ARMv7

In ARMv7, you've got "TrustZones", typically divided into "Monitor", "Hypervisor", "Secure", "Non-Secure".

ARMv8

ARMv8 provides different "Exception Levels", EL0 to EL3, where typically EL0 is user space, EL1 OS space, EL2 hypervisor space and EL3 monitor mode.

PowerPC

On PowerPC systems, the access is divided in "hypervisor supervisor", "hypervisor user", "guest supervisor" and "guest user".

Intel® x64

The Intel® x64 architecture knows a "hypervisor" and "guest" address space.

These modes provide the possibility to create their own MMU translation. By default (**SYStem.Option.ZoneSPACES OFF**), the debugger does not care about the current CPU mode and correlates the symbols of an application only by its address. In OS and hypervisor environments, the symbol may be bound to an additional space ID and/or machine ID, but this only depends on the current running process and/or guest, not on the CPU mode. This setting should be used whenever this is enough to distinguish the different applications. E.g. in an ARMv7 system, if the "Secure" code is running in a different address range than the "Non-Secure" code, or if the hypervisor and OS code is always uniquely identifiable by the machine ID or space ID.

If a system sets up the MMU in a way that the code of the different modes overlaps virtually, the debugger needs to distinguish the accesses and symbols by the CPU mode. E.g. if the "Secure" application runs on the same virtual address range as the "Non-Secure" application, or if the hypervisor contains both, code running on EL2 and EL1 with overlapping address ranges. To bind the symbols to a specific CPU mode, set **SYStem.Option.ZoneSPACES** ON and load the symbols to the according access class. In hypervisor systems, combine the access class with machine ID. E.g.

```
SYStem.Option.ZoneSPACES ON
Data.LOAD.Elf hypervisor_drv.elf  H:0:::0 /NoCODE
Data.LOAD.Elf scheduler_app.elf   N:0:::0 /NoCODE /NoClear
Data.LOAD.Elf guest-app.elf       N:1:::0 /NoCODE /NoClear
```

Symbol Management

The goal of the Hypervisor Awareness in TRACE32 is to have access to everything that runs on the target. This means you will be able to debug the hypervisor, all guests and processes, all at once. To access a specific function or variable within a part of the software (e.g. Linux process or bare metal guest), you can simply use the symbols to e.g. view the variable or set a breakpoint on the function.

By loading the symbol information (ELF file) of a part of the software to a specific machine ID and space ID, the debugger "knows" to which machine and process the symbol belongs. E.g. loading the symbols of a process called "sieve" to guest 1 an process ID 0x12:

```
Data.LOAD.Elf sieve.elf 1:::0x12:0 /NoCODE /NoClear
```

Note that processes of supported OSs are usually loaded by the symbol autoloader, which does this internally.

This means, a symbol is *always* bound to a specific machine and space. The debugger will then automatically decode the virtual address and access the correct virtual or physical memory location.

When using the symbol alone, the debugger will check, if the symbols is globally unique, or if it is unique in the current context. It will then use the associated machine and space:

```
Break.Set function1
Variable.View var1
```

Symbols in TRACE32 follow a "symbol path". The full path is identified by *<machine_name>**<program>**<module>**<symbol>*, where *<machine_name>* is the machine name, *<program>* is usually the file name (without extension), *<module>* is usually the function name, and *<symbol>* the C symbol name. By using the full path, you can identify symbols that are not unique over the platform. You can omit parts of the path if the path remains unique. Some examples:

```
Break.Set \\\guest1\\sieve\main\func1
Break.Set \\sieve\\func2
Variable.View \\vmlinux_guest\\linux_banner
Variable.View \\vmlinux_host\\linux_banner
```

Menus and Commands

This chapter gives an overview about commands and menus to be used when a Hypervisor Awareness is active. See also a detailed description in chapter **Using Hypervisor Features**.

Several commands accept machine specific parameters.

If the parameter is an address, you can specify the machine by adding the machine ID to the address.See **"Machine ID"** in TRACE32 Concepts, page 26 (trace32_concepts.pdf) for details. When specifying a symbol as parameter, the machine ID is included in the symbol information, see "**Symbol Management**".

Some commands allow an explicit declaration of the machine. In this case the command accepts the parameter "/MACHINE", where you can specify a machine magic (if available), machine ID or machine name. The command will then act on this machine.

Extension files (awareness files) can be loaded to a specific machine with the /MACHINE option. The extension will then read its data from the specified machine. This way you can load OS Awarenesses to guests. To be able to execute an awareness specific command for a specific guest, you can specify the awareness name after the "TASK." or "EXT." command (infix). **Example**:

```
; load Linux awareness to machine 1
EXTension.LOAD ~~/demo/arm/kernel/linux/awareness/linux.t32 /MACHINE 1 \
    /NAME vm1

; load another Linux awareness to machine 2
EXTension.LOAD ~~/demo/arm/kernel/linux/awareness/linux.t32 /MACHINE 1\
    /NAME vm2

; display process list of both guests
TASK.vm1.Process
TASK.vm2.Process
```

The awareness files of the hypervisor create an additional menu for the hypervisor itself. A script, also provided by the hypervisor support, reorganizes menus of guest OSs to be able to access each guest by its own menu entry. At the end, you'll get a menu with dedicated entries for the hypervisor and each guest.

Functional Details of the TRACE32 Hypervisor Support

TRACE32 collects a lot of information from the target to provide a hypervisor support. It then prepares this information to be shown in a convenient way. See also chapter **Using Hypervisor Features**.

Hypervisor Awareness

An integral part of the hypervisor support is the so-called "Hypervisor Awareness". Each hypervisor is different and handles the guests in a different way. That's why Lauterbach creates an external extension for each hypervisor. This extension, which forms the "awareness", reads out all the necessary information from the target. To be able to find this information, in most cases the awareness needs the symbol information of the hypervisor.

The Awareness mainly reads out the list of guests and their attributes (ID, name, etc.). It also reads out the current state of the guest virtual CPUs (VCPUs), to reconstruct the execution context of the guest. This is also very important to provide the guest MMU translation on debugger side.

MMU Translation

As mentioned before, there is a two-stage MMU translation active, one for the guest OS (first stage), and one for the hypervisor (second stage). To be able to access the hypervisor and each guest at any time, regardless at which context the current CPU is operating, the debugger must do the MMU translation by itself. It has to know the characteristics and current settings of the translation and the MMU registers. The static characteristics are provided by TRACE32 **TRANSlation** commands, while the current MMU settings are read out of the target with the help of the Hypervisor Awareness.

Machines, Spaces and Tasks

To operate the hypervisor support, the debugger needs the lists of machines, MMU spaces and tasks that are currently active on the target. It uses the Hypervisor Awareness to read the list of machines with its attributes and it uses the guest OS Awareness to read out the space and task list of each guest. Out of this, the debugger creates an internal database upon the state of the system. You can view this information with the **TASK.List** commands. More detailed information is given in the chapter "**Viewing the System**".

Cores VCPUs and Contexts

Obviously, the real system operates with the physical register set of the CPU cores. This is used by TRACE32 to determine the current state of the system. In case you want to see information of a currently not active guest, this is not sufficient. In this case, the debugger uses the Hypervisor Awareness to read the VCPU register contents out of the target memory. It is then able to reconstruct the register set at the time when the guest was preempted and can display the system as if the guest was active. Similar to this, the debugger can show the state of a task at the time when it was preempted. In this case, the debugger uses the OS Awareness to read the register set of the task and reconstructs the environment.

You can view the register sets of VCPUs and task contexts, see "**Viewing Registers**".

State Line - Cursor Field

The **Cursor** field provides the following information about the hypervisor environment:

*   Boot information of TRACE32 (Booting …, Initializing … etc.).

*   Directly at debug mode entry (i.e. when a breakpoint has been hit or a **Break** command has been executed), the fully qualified address (e.g. `NUX:2:::03A5:00401EC0`) of the current PC is shown and, if available, the associated symbol (e.g. `\\sieve\sieve\fill_buffer+0x64`).

Address information available in the **Cursor** field:
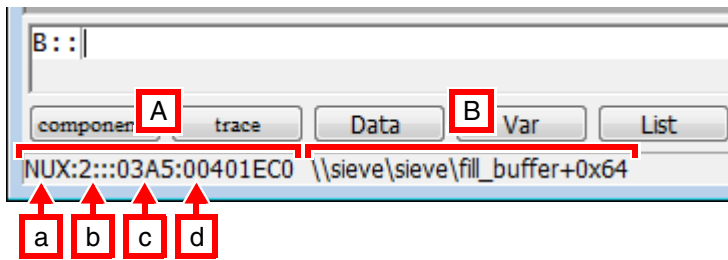
A Example of a fully qualified address in a hypervisor environment:

- **a** Access class (`NUX:`)
- **b** Machine ID (`2:::`)
- **c** Space ID (`03A5:`)
- **d** Logical address (`00401EC0`)

B Symbol (`\\sieve\sieve\fill_buffer+0x64`)

The machine ID [**b**] is displayed only if you set **SYStem.Option.MACHINESPACES** to **ON**, and the space ID [**c**] is displayed only if you set **SYStem.Option.MMUSPACES** to **ON**.

State Line - Task Field

The **Task** field in the state line displays the name of the currently running task preceded by the name of the machine which runs the task (Example: `Linux:::sieve`). The machine name [**A**] and task name [**B**] are separated with three colons `:::`

The task name can only be displayed if a suitable OS Awareness has been loaded for the machine which is being executed on the current core.
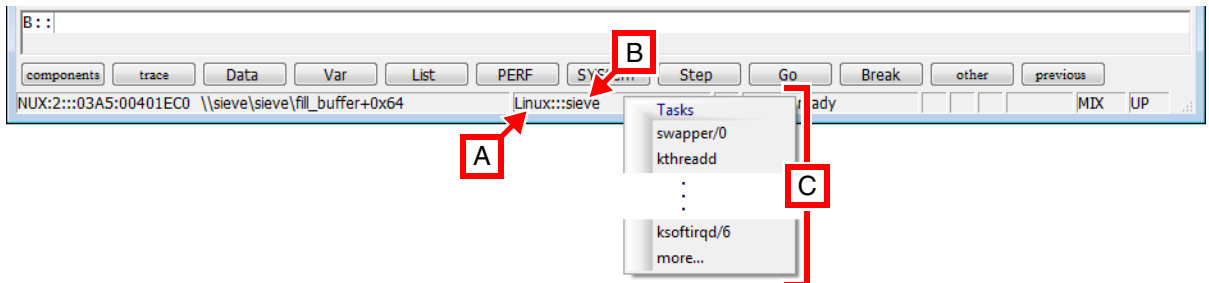
Double-clicking the **Task** field [**A**] opens the **TASK.List** window. Alternatively, right-click and then select more from the **Tasks** popup menu.

Use the Task Field to Select the Default Viewing Context

Right-clicking the **Task** field opens the **Task** drop-down list [**C**] where all task running in the system are shown. You can temporarily switch the default viewing context to any task by selecting one of the tasks. The windows **Register.view**, **Frame.view**, **Datal.List** etc. are now showing data belonging to the selected default viewing context. It is not possible to modify data of a task which is not currently running on a physical core.

State Line - Cores Field and CORE.List Window

For each physical core assigned to TRACE32 with the **CORE.ASSIGN** command, TRACE32 uses a logical core number, which serves as an alias for the physical core. For more information, please refer to the **CORE.ASSIGN** command.

The **Cores** field [**A**] shows the currently selected logical core (short: selected core). TRACE32 PowerView visualizes system information from the perspective of the selected core, such as the currently executed machine, task, and PC of each logical core.
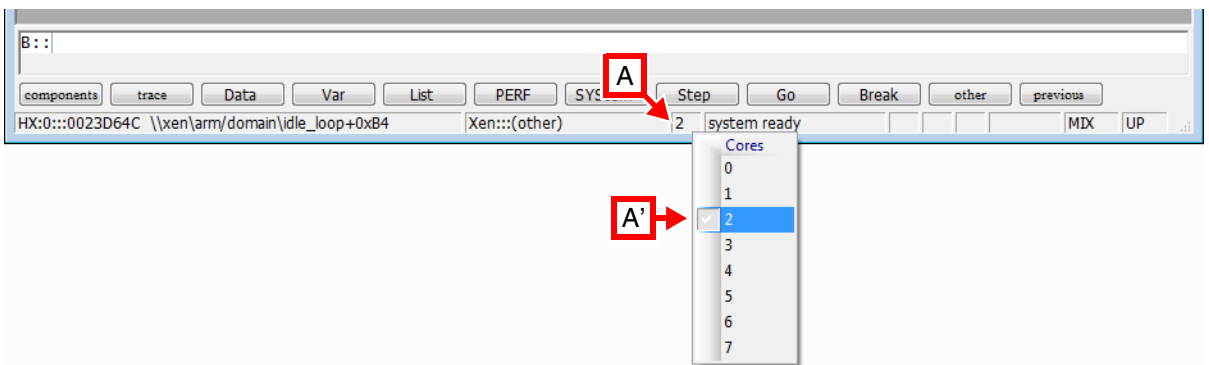
Double-clicking the **Cores** field [**A**] opens the **CORE.List** window:
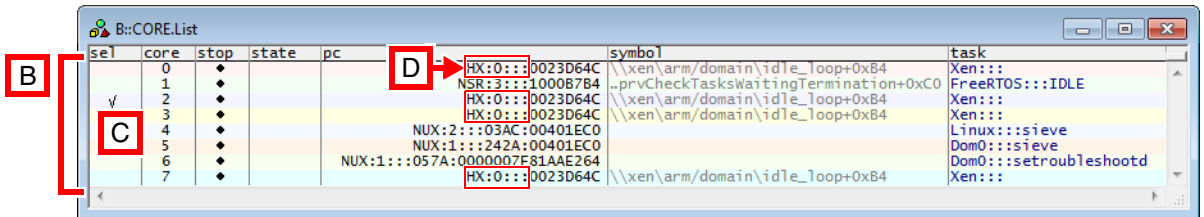
B Each row stands for one logical core, and each logical core has its own color. The same color of a core is re-used in other TRACE32 windows that display information from that particular core.

C The check mark indicates the selected core in both the **CORE.List** window and the **Cores** drop-down list.

D In the **pc** column, all rows containing `0:::` (= machine ID 0 followed by three colons) tell you the cores on which the host machine is running.

To select a core, do one of the following:

•   Right-click the **Cores** field, and then select the core you want from the **Cores** drop-down list.

•   Double-click the core you want in the **CORE.List** window.

   As a result, the viewing context of the debugger switches to the newly selected core. The background color and the information displayed in the affected windows changes accordingly.

   The background color of non-affected windows remains white as usual.

To pin a window to a particular core:

•   Append **/CORE** *<number>* to the TRACE32 window command.

   As a result, the window no longer follows your core selection in the **CORE.List** window or the **Cores** drop-down list but displays only information from the specified core.

Example:

```
Register.view /CORE 3.
```

Configuration of TRACE32

In this section:

- SMP or AMP configuration

- System settings

- Configuring the hypervisor

- Configuring the guests

- Example script for Xen

- Special architectural considerations

- Configuring the MMU

SMP or AMP Configuration

A hypervisor usually provides "virtual CPUs" (VCPUs) to the guests. These VCPUs are then mapped dynamically to the existing physical cores. The number of VCPUs is not related to the number of physical cores, in fact, you can configure more VCPUs (even within one guest) than physical cores. The hypervisor schedules all VCPUs to the physical cores in way that each VCPU gets some computing time. This scheduling of VCPUs is similar to scheduling tasks in an OS. An application, or even a complete guest OS, therefore can migrate from one physical core to another, making it almost impossible to predicate, at which core a specific code portion will run. TRACE32 takes care of this by watching all cores in the hypervisor system and by attaching always the current running guest to each core. In this case, TRACE32 *must* run in SMP mode, to control all affected cores at the same time.

To start TRACE32 in SMP mode, simply set **SYStem.CPU** to a chip that has several cores. **SYStem.Mode Up** or **SYStem.Mode Attach** will then automatically run the debugger in SMP mode to all available cores. If not all available cores are used for the hypervisor system, you can manually select the affected cores with the **CORE.ASSIGN** command. TRACE32 will then only control these cores, keeping all others untouched and free running.

Example 1: SMP configuration on R-Car H3

```
SYStem.CPU R8A77950
CORE.ASSIGN 1. 3. 5. 7.    ;/only the A57 cores used by hypervisor
SYStem.Mode Attach
Break
```

If the VCPUs of a guest are fixed bound to dedicated cores, and if these cores are running one guest exclusively, you may also run TRACE32 in an AMP configuration. That means that you create a TRACE32 PowerView GUI that only controls the VCPUs (and therefore the physical cores) of this guest. Use the **CORE.ASSIGN** command to attach the GUI to all physical cores used by a specific guest. You may do this in parallel with several guests, if those have a fixed mapping of VCPUs to physical cores, too. In this configuration, one TRACE32 PowerView GUI only controls one specific guest, and thus does not need to have a Hypervisor Awareness. These debug instances can be configured as if these cores were running the

guest OS without a hypervisor. Because you don't need a Hypervisor Awareness in these configurations, they will not be discussed in this document. We'll keep the SMP configuration assumed in the following chapters.

Example 2: AMP configuration with Jailhouse

```
InterCom.NAME RootCell
TargetSystem.NewInstance Inmate
InterCom.WAIT Inmate

InterCom ALL SYStem.CPU ZYNQ-ULTRASCALE+-APU
InterCom RootCell SYStem.CONFIG CORE 1. 1.
InterCom Inmate   SYStem.CONFIG CORE 2. 1.

InterCom RootCell CORE.ASSIGN 1. 2. 3.
InterCom Inmate   CORE.ASSIGN 4.

InterCom ALL SYStem.Mode Attach
InterCom ALL Break
```

System Settings

Several system settings configure, how the debugger works with the Hypervisor Awareness.

SYStem.Option.MACHINESPACES is the most important option, as it actually enables the hyervisor awareness. When set to ON, the debugger enables the machine ID. If a Hypervisor Awareness is loaded, the debugger uses this awareness to read the characteristics of the hypervisor and to get the current active machine.

SYStem.Option.MMUSPACES enables MMU space handling. This is necessary, if one of the guests or the hypervisor itself uses MMU spaces (processes). The debugger uses the OS Awareness of the guest to determine the existent spaces and to get the current active MMU space.

SYStem.Option.ZoneSPACES enables the distinction of symbols on different CPU zones. This is necessary, if e.g. hypervisor level code overlaps with user level code on the same machine. See detailed description in "Configuring the MMU" later in this chapter.

Several architectures allow to read memory while the target is running. Use **SYStem.MEMACCESS** to define, how the debugger can access memory in run time. It will enable viewing the machines and guests even if the target is not halted. Please note that some access methods bypass the caches. This is usually not suitable for hypervised systems, because the debugger will read outdated memory contents and the MMU table walk will fail.

At last, you have mainly two options to connect the debugger to the target:

SYStem.Mode Up will connect the debugger to the CPU and execute a reset on this CPU. This is useful, if you want to start debugging right from the beginning. However, note that neither the hypervisor nor any guests are active at this time. It's recommended to wait for the configuration of the machines until the hypervisor started.

SYStem.Mode Attach will connect the debugger to an already running target. Please note that the target keeps running until you perform a break. This allows you to immediately read out all the characteristics and to configure the hypervisor and guests according to the target state.

Example: attaching to an already running system

```
SYStem.Option.MACHINESPACES ON    ; enable machine IDs
SYStem.Option.MMUSPACES ON        ; enable space IDs
SYStem.Attach                     ; attach to running target
Break                             ; halt target for inspection
```

Configuring the Hypervisor

The Hypervisor Awareness relies on the symbol information of the hypervisor. Therefore it is essential to load the symbol file of the hypervisor. As the hypervisor always runs on machine ID zero, load the symbols to this machine ID and specify the access class at which the hypervisor runs. See example below.

Regardless if the Hypervisor uses a stage 1 MMU translation for itself, you have to declare the MMU characteristics to the debugger. If the hypervisor doesn't use MMU spaces, declare a common area over the whole address range.

Load the Hypervisor Awareness and menu that fits to your system. Specify machine ID zero and the access class that is used to access the hypervisor internals.

Example: configuring Xen hypervisor

```
; load the hypervisor symbols to access class H: and machine ID 0
Data.LOAD.Elf xen-syms H:0:::0::0 /NoCODE /NAME xen

; set up debugger address translation for the hypervisor
MMU.FORMAT STD
TRANSlation.COMMON H:0:::0x0--0xffffffffffffffff
TRANSlation.TableWalk ON
TRANSlation.ON

; load the Hypervisor Awareness
EXTension.LOAD ~~/demo/arm/kernel/xen/xen.t32
     /MACHINE 0 /ACCESS H: /NAME "Xen"
MENU.ReProgram ~~/demo/arm/kernel/xen/xen.men
```

Configuring the Guests

To be able to work with the guest applications, the debugger needs to get knowledge about the guest characteristics. At first you need to load the symbol information of the guest to the machine ID of the guest. If the guest contains an OS, load the symbol information of the OS to be able to work with a guest OS Awareness. See example below.

If the guest uses its own MMU translation, you need to declare the layout to the debugger. Always specifiy the according machine ID. Note that guest MMU translates to intermediate (I:) addresses, not physical (A:).

At last, load the guest OS Awareness, if there is any. Again, specify the machine ID of the guest. If you have several guests with the same OS, you need to modify the menu entries to be unique for each guest. If the guest OS Awareness needs any additional settings, remember to use the infix method of the TASK command to identify the guest.

Example: configuring a Linux guest in Xen

```
; load the symbols of the guest OS
Data.LOAD.Elf vmlinux N:1:::0::0 /NoCODE /NoClear /NAME dom0

; set up debugger address translation for the guest
MMU.FORMAT LINUXSWAP3 \\dom0\\swapper_pg_dir /MACHINE 1
TRANSlation.Create N:1:::0xFFFF000008080000--0xFFFF000008FFFFFF \
    I:1:::0x50080000
TRANSlation.COMMON N:1:::0xFFFF000000000000--0xFFFFFFFFFFFFFFFF

; load the guest OS Awareness
EXTension.LOAD ~~/demo/arm/kernel/linux/linux-3.x/linux3.t32 \
    /MACHINE 1 /NAME "Dom0"
TASK.Dom0.Option MMUCHECK OFF
```

Example Script

This is an example script for a hypervisor system using Xen on a HiKey 960 board.
The hypervisor runs 3 guests:

• guest 1: Dom0, a privileged Linux guest

• guest 2: DomU, running another Linux and

• guest 3 running FreeRTOS.

Example: Xen configuration on HiKey 960 with three guests

```
; Reset debugger (not target!)
RESet

; ===== Configure Debugger =====

SYStem.CPU KIRIN960                    ; set CPU type
SYStem.Option.MMUSPACES ON            ; enable space IDs
SYStem.Option.ZONESPACES ON           ; enable zone distinction
SYStem.Option.MACHINESPACES ON        ; enable machine IDs

; Attach to all cores, correct core sequence is important
CORE.ASSIGN 2. 4. 6. 8. 1. 3. 5. 7.  ; little cores first
SYStem.Mode Attach                     ; attach to running target
Break                                  ; halt target


; ===== Configure Hypervisor support =====

; Load hypervisor symbols
Data.LOAD.Elf xen-syms H:0:::0::0 /NoCODE /NAME xen

; Declare Hypervisor MMU layout and enable debugger translation
MMU.FORMAT STD
TRANSlation.COMMON H:0:::0x0--0xffffffff
TRANSlation.TableWalk ON
TRANSlation.ON

; Load Hypervisor Awareness
EXTension.LOAD ~~/demo/arm/kernel/xen/xen.t32 \
    /MACHINE 0 /ACCESS H: /NAME Xen
MENU.ReProgram ~~/demo/arm/kernel/xen/xen.men
```

```
; ===== Configure guest 1 (Dom0, N:1:::) =====

; Load guest 1 symbols
Data.LOAD.Elf dom0/vmlinux N:1:::0::0 /NoCODE /NoClear /NAME dom0

; Declare guest 1 MMU layout
MMU.FORMAT LINUXSWAP3 \\dom0\\swapper_pg_dir /MACHINE 1
TRANSlation.Create N:1:::0xFFFF000008080000--0xFFFF000009020FFF \
    I:1:::0x20080000
TRANSlation.COMMON N:1:::0xFFFF000000000000--0xFFFFFFFFFFFFFFFF

; Load guest 1 OS Awareness
EXTension.LOAD ~~/demo/arm/kernel/linux/linux-3.x/linux3.t32 \
    /MACHINE 1 /Name Dom0


; ===== Configure guest 2 (DomU, N:2:::) =====

; Load guest 2 symbols
Data.LOAD.Elf domu/vmlinux N:2:::0::0 /NoCODE /NoClear /NAME linux

; Declare guest 2 MMU layout
MMU.FORMAT LINUX \\linux\\swapper_pg_dir /MACHINE 2
TRANSlation.Create N:2:::0xFFFF000008080000--0xFFFF000009020FFF \
    I:2:::0x40080000
TRANSlation.COMMON N:2:::0xFFFF000000000000--0xFFFFFFFFFFFFFFFF

; Load guest 2 OS Awareness
EXTension.LOAD ~~/demo/arm/kernel/linux/linux-3.x/linux3.t32 \
    /MACHINE 2 /NAME Linux


; ===== Configure guest 3 (FreeRTOS, N:3:::) =====

; Load guest 3 symbols
Data.LOAD.Elf FreeRTOS.elf N:3:::0 /NoCODE /NoClear /NAME freertos

; Load guest 3 OS Awareness
EXTension.LOAD ~~/demo/arm/kernel/freertos/freertos.t32 \
    /MACHINE 2 /NAME "FreeRTOS"

; ===== Done =====

ENDDO
```

Special Architectural Considerations

ARMv7 architectures:

The hypervisor usually runs in "hypervisor" mode of the CPU. This mode is mapped to the TRACE32 access class "H:". If some code of the hypervisor runs in a different mode (e.g. "N:"), and if these code fragments overlap, you may need to set **SYStem.Option.ZoneSPACES** ON, to have a clear distinction of the symbol information for the code running in hypervisor and user/supervisor mode (see also next chapter).

ARMv8 architectures:

The hypervisor usually runs in "EL1" mode of the CPU. This mode is mapped to the TRACE32 access class "H:". If some code of the hypervisor runs in EL2, and if these code fragments overlap, you may need to set **SYStem.Option.ZoneSPACES** ON, to have a clear distinction of the symbol information for the code running in EL1 and EL2 (see also next chapter).

PowerPC architectures:

PowerPC provides two ways of separating user space from supervisor space. Please ensure to set the **SYStem.Option.TranslationSPACE** accordingly. Refer to **"QorIQ Debugger and NEXUS Trace"** (debugger_ppcqoriq.pdf) for more information.

The MMU tables in PowerPC are separeted on Hypervisor/Guest, as well as Supervisor/User. If you need a distinction of Supervisor/User *within* a machine, set **SYStem.Option.ZoneSPACES** to ON.

Configuring the MMU

Address translation is one of the key functionalities in a hypervisor system. For addresses from guest machines, there are two independent stages of translation:

- the translation of guest logical addresses to guest physical addresses (intermediate addresses) in each guest machine. The translation tables involved are controlled by the guest OS. We refer to this translation step as **stage 1 translation**

- On the host machine, the intermediate addresses of each guest are translated to absolute physical addresses. The translation tables used are controlled by the hypervisor. We refer to this translation step as **stage 2 translation**.

The built-in debugger address translation of TRACE32 can mimic both translation stages. So, the debugger can retrieve the full translation of all guest and host machine data at any time and read the data from physical memory.

Further, addresses of the host machine itself also need a translation. This is usually only a single translation stage, a page table maintained by the hypervisor itself.
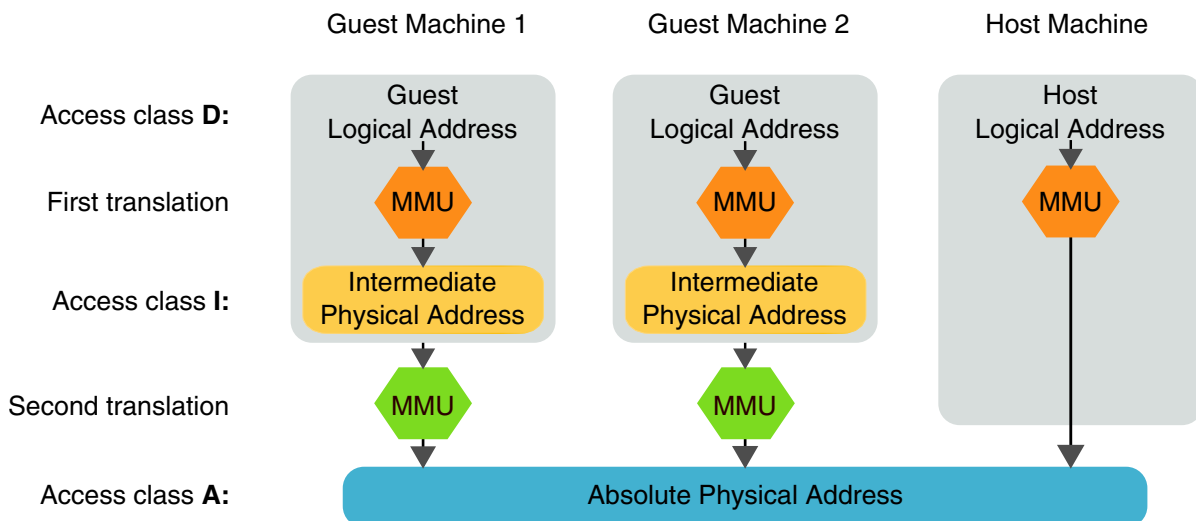
A proper configuration of TRACE32 for the hypervisor MMU is essential for the debugger to access any data in the system. A proper configuration of TRACE32 for a guest machine MMU is needed to debug the guest OS or applications running under its control.

The configuration splits up into three major blocks:

1. The SYStem configuration appropriate for the hypervisor system you want to debug

2. The MMU configuration for the host machine running the hypervisor

3. The MMU configuration for all guest machines you want to debug

Selecting the System Configuration

Command **SYStem.Option.MACHINESPACES ON** is mandatory in the head of a TRACE32 debugger script. This command configures TRACE32 to use machine IDs and to keep symbols, MMU configurations and breakpoints separate for all machines. Additionally, a separate OS Awareness can be loaded for each machine.

Enabling or disabling ZoneSPACES

Use the following check list to decide whether you should set **SYStem.Option.ZoneSPACES** to **ON** or **OFF**.

Set **SYStem.Option.ZoneSPACES** to **OFF** if any of the following statements is true:

- if your CPU has only one zone, i.e. if it has not more than one CPU mode with individual MMU and register set.

- if your CPU has more than one zone but in your system the machine ID alone is sufficient to determine clearly in which zone or CPU mode a machine is executing.

Set **SYStem.Option.ZoneSPACES** to **ON** if you are using a CPU which has more than one zone, i.e. more than one CPU mode with own MMU and register set and

- the hypervisor executes part of its code in more than one CPU zone and symbol addresses used in the zones may overlap

- or any guest machine executes part of its code in more than one CPU zone and symbol addresses used in the zones may overlap

- or you need to load symbols and debug code which runs in a CPU zone which is neither used by the hypervisor nor the guest machines.

If you are unsure which category your system falls into, you can try it out. Start with the simpler configuration and set **SYStem.Option.ZoneSPACES** to **OFF**.
In this configuration, MMU settings made with the **TRANSlation** and **MMU** command groups and symbols sets loaded with **Data.LOAD** are only separated by machine ID and not by CPU zone (i.e. by access class).
If machines in your system are using overlapping address ranges of code or data, executed in more than one CPU zone, you might experience problems. Such problems could be missing information from the hypervisor or guest OS Awareness windows, you might see MMU translation errors or get warning messages about overlapping stack segments. If such problems occur, try to set **SYStem.Option.ZoneSPACES** to **ON**.

Examples for CPUs with more than one zone are ARM CPUs with TrustZone and/or VirtualizationExtension, Intel® CPUs with VMX virtualization technology or PowerPC QORIQ CPUs with hypervisor support.

- An example for a hypervisor system where **SYStem.Option.ZoneSPACES OFF** is suitable is Xen on ARMv8 where the Hypervisor runs exclusively in EL2 mode (access class H:0:::) and all guests run exclusively in EL0 and EL1 mode (access class NU:0::: and NS:0:::).

- An example for a hypervisor system where **SYStem.Option.ZoneSPACES ON** is required is the QNX hypervisor running on ARMv8. In this system, most hypervisor code executes in EL1 nonsecure mode (access class N:0:::), but some priviledged driver code executes in EL2 mode (access class H:0:::).

Hypervisors running on CPUs without hardware virtualization support always belong to this category. For hypervisors running on CPUs with hardware support such as ARM TrustZone, Intel® VMX or PowerPC QORIQ different machines may execute in more than one CPU mode.

Examples:

- Xen on ARM64 where the Hypervisor runs exclusively in EL2 mode and all guests run exclusively in EL0 and EL1 mode (access class NU:0::: and NS:0:::)

- paravirtualized hypervisors running on CPUs without virtualization extension

Configuring the MMU for the Host Machine Running the Hypervisor

To evaluate the correct address translation, the debugger needs to know the layout of the MMU translation for each machine. Especially the MMU layout for the host machine is essential, as all virtual to physical translations will go through this. The MMU layout depends on several items:

- The CPU architecture

- The layout of the PTEs (MMU Page Table Entries)

- The content of the MMU registers

- Hypervisor / OS specific MMU settings

The actual configuration can vary from system to system, as a good start look for example scripts for your hypervisor / architecture combination.

The debugger knows about the CPU architecture by the **SYStem.CPU** setting. No special setting necessary for this one.

The PTE layout may vary. Use the command **MMU.FORMAT** to inform the debugger about the particular layout. On some architectures (like most Arm CPUs), this is a fixed format, where you can use "STD" as argument. If the PTE layout is not fixed by the architecture, each hypervisor / OS can use a different format. In this case, you need to declare the particular format to the debugger.

The actual used MMU layout depends on the current content of the MMU registers in the CPU. E.g. the MMU table base address is stored in a special register. To do the address translation correctly, the debugger needs to know all the MMU related register settings. If not otherwise specified, the debugger reads out the registers and uses its values. However, if the registers are altered during run time, e.g. because of a guest switch, the debugger still needs to know the register settings of the according machine. The most important part, the table base address, can be applied to the **MMU.FORMAT** command. The Hypervisor Awareness may provide further information about the MMU register contents of each machine (including the hypervisor).

If **SYStem.Option.MMUSPACES** is **ON**, also the hypervisor address range is divided into MMU spaces. If the hypervisor does not use MMU spaces (e.g. Xen), then simply tell the debugger to ignore the space ID, by setting **TRANSlation.COMMON** on the complete address range of the hypervisor. If the hypervisor contains an OS using MMU spaces (e.g. QNX), set the TRANSlation.COMMON range as advised in the OS Awareness manual.

Starting up the System

The easiest way to debug a hypervisor based system is to let the target start without any debugger intervention, then attach the debugger to the running system. This way all target settings are already done and the debugger doesn't need to take care about the boot sequence. A script to do this could be as simple as:

```
SYStem.CPU <cpu>
SYStem.Mode Attach
Break
```

However, in some cases you might want to start the target under the control of the debugger. E.g. to debug the startup sequence, to download the application image via the debugger, or to do modifications during the startup phase. This chapter gives an overview how a debugger controlled startup can be accomplished.

Attaching to the Boot Core

Single-Core, halt on Reset:

In a single core environment, you need to select the used CPU type, set all CPU specific options, and issue a **SYStem.Mode Up**.

```
SYStem.CPU <cpu>
SYStem.Up
```

The SYStem.Up command attaches the debugger to the CPU, performs a reset of the CPU and tries to halt on the reset vector.

NOTE:	In complex SOCs, it may not be possible or require further actions to reset the CPU via JTAG and/or to halt the CPU on the reset vector

Multi-Core, halt on Reset:

In an SMP environment, often only the first core comes out of reset, while the others are still not accessible. In this case, use **CORE.ASSIGN** to assign the debugger only to the first core, then issue a **SYStem.Mode Up**.

```
SYStem.CPU <cpu>
CORE.ASSIGN 1.
SYStem.Up
```

CORE.ASSIGN causes the debugger to talk only to the first core, while all other cores remain untouched. The SYStem.Up command then attaches to the first core, performs a reset of this core and tries to halt on the reset vector.

Halt after Initialization:

On complex systems, it is often not possible to halt the CPU on the reset vector. In this case, you may try **SYStem.RESetTarget** to reset the board, or reset it manually, and attach to the CPU after the first initializations are done. A good time to attach to it is after the bootloader started into the boot prompt.

```
; wait for bootloader prompt, then:
SYStem.CPU <cpu>
CORE.ASSIGN 1.      ; in case of SMP
SYStem.Attach       ; attach to first core
Break               ; halt it
```

Loading the Images

There are various ways to load an application image into the RAM of the system. Typically the bootloader copies it from internal storage (Flash, SD-card, ...) to RAM and starts it. During development phase you may want to load the image directly from your development PC for faster turnaround. This chapter shows three different ways how to do this, based on u-boot. Which one to choose depends on the capabilities of the target system and the development cycle (there's no "best way"). Other bootloaders and other download strategies work in a similar way.

Loading and Starting the Image via U-Boot

You can simply use the bootloader to load and run your application, just as usual. You may let it start and attach later to the running application. If you want to debug the startup sequence, you may halt at the entry point of the loaded application.

Example: We're loading and starting a Linux image and DTB from SD-card to physical addresses 0x48080000 rsp. 0x48000000 using u-boot. The debugger is set up to halt at the (physical) entry point of the image:

```
; Example TRACE32 commands to halt when the loaded image is started.
; Execute this before starting the image in u-boot

; Gain access to the CPU:
Break

; Set an onchip breakpoint onto the entry point of the image.
; Onchip because the loading of the image overwrites software BPs.
Break.Set 0x48080000 /Onchip
```

```
; Continue
Go

; Wait until breakpoint is hit. Start the image in u-boot terminal
; (see block below).
WAIT !STATE.RUN()

; After this, the target halted at the image entry point.
; Remove any breakpoints
Break.Delete
```

```
; Example u-boot commands to load and start a Linux image:

=> ext2load mmc 0:1 0x48080000 /boot/Image
=> ext2load mmc 0:1 0x48000000 /boot/board.dtb
=> booti 0x48080000 - 0x48000000
```

Loading the Image via Debugger and Starting via U-Boot

If you don't want or cannot load the binary image of your application with u-boot, you may use the debugger to do so. Let u-boot start the target into the u-boot prompt, then load the image via debugger and start it via u-boot as usual.

Example: After starting u-boot into the u-boot prompt, we're loading a Linux image and DTB via JTAG to physical addresses 0x48080000 rsp. 0x48000000 and then start the image using u-boot. The debugger is set up to halt at the (physical) entry point of the image:

```
; Example TRACE32 commands to halt when the loaded image is started.
; Execute this when u-boot shows the prompt.

; Gain access to the CPU:
Break

; Load the Linux image and DTB to the physical addresses
Data.LOAD.Binary /host/path/to/boot/Image AD:0x48080000
Data.LOAD.Binary /host/path/to/boot/board.dtb AD:0x48000000

; Set a breakpoint onto the entry point of the image.
Break.Set 0x48080000
```

```
  ; Continue
  Go


  ; Wait until breakpoint is hit. Start the image in u-boot terminal
  ; (see block below).
  WAIT !RUN()

  ; After this, the target halted at the image entry point.
  ; Remove any breakpoints
  Break.Delete
```

```
  ; Example u-boot commands start a Linux image:

  => booti 0x48080000 - 0x48000000
```

Loading ELF Files via Debugger

When loading binary files via the debugger, you need to know the addresses, to which the file has to be loaded. ELF files, in turn, include the load addresses and the program start address. This means, when loading an ELF file with the debugger, you don't need to specify any address. On the other hand, it often does not include the DTB, and it is quite complicated to provide a DTB then (setting the DTB address in a register). Loading and starting an application by an ELF file usually makes sense, if no external DTP is needed. Again, the easiest way is to let the bootloader initialize the system, then download the application. As the start address is provided by the ELF file, we can directly start the application after downloading, without bootloader intervention.

Example: After starting u-boot into the u-boot prompt, we're loading an application ELF file via JTAG and then start the image using the debugger.:

```
  ; Example TRACE32 commands to load and start an ELF image.
  ; Execute this when u-boot shows the prompt.

  ; Gain access to the CPU:
  Break

  ; Load the application's ELF file into RAM.
  Data.LOAD.Elf myApplication.elf

  ; After this, the PC is already set to the entry point.
  ; Simply continue to start the application
  Go
```

Debugging Startup Sequence

If you want to debug the very early bootup stages of your application, you have to care about several issues.

First, some multi-core systems just start with one core awaken. In this case, ensure that the debugger is only attached to the first core by executing

```
CORE.ASSIGN 1.
```

before attaching the debugger to the target (SYStem.Attach or SYStem.Up).

Second, usually the image is started with MMU disabled. That means, the system is running in physical address space. Symbols are mapped to virtual addresses, so the symbols must be shifted from virtual to physical address. E.g.

```
Data.LOAD.Elf myApplication.elf <phys_addr>-<virt_addr> /NoCODE
```

Now you can debug the physical startup, until the MMU is switched on. As soon as the MMU is active, you have to reload the symbols to the virtual address space:

```
Data.LOAD.Elf myApplication.elf /NoCODE
```

Attaching to SMP Cores

Often, multi-core systems start with just one core enables. In this case, start the debugger attached just to the first core (see chapter above).

As soon as the hypervisor enabled all cores of an SMP system, the debugger should be attached to all cores, too, to manage breakpoints correctly.

Example: Reattach the debugger to 8 cores

```
SYStem.Mode Down
CORE.ASSIGN 1. 2. 3. 4. 5. 6. 7. 8.
SYStem.Mode Attach
```

NOTE:	Ensure that the core ordering is the same as the one used by the hypervisor, especially in big-little systems!

NOTE:	Do *not* use SYStem.Mode Up, as this would reset the cores.

Using Hypervisor Features

In this section:

Viewing the System

A complete hypervisor based system consists of the following major components:

- Host machine
- Guest machines
- Physical cores (used by the host machine)
- Virtual cores (aka "vCPUs", used by the guest machines)
- MMU spaces (in host and/or guest machines)
- Tasks (in host and/or guest machines)

Host Machine

There are several ways to view information about the host machine (running the hypervisor).

Whenever an address is given, you can simply use machine ID 0 to access the host machine. In systems, where the host machine is running in hypervisor mode of the CPU, you can also use the H: access class.

Examples: showing code and data of the host machine

```
List.Asm 0:::0x1234      ; show code of machine 0 at address 0x1234
Data.dump 0:::0x5678     ; show data of machine 0 at address 0x5678
Data.dump H:0x5678       ; show data at address 0x5678 using H: access
```

Loaded symbols are bound to a given machine. That means, the debugger knows that a specific symbols belongs to the host machine. When accessing symbols (display code or variables), simply use the symbol; no special action is necessary to specify the host machine.

Some commands take the option /MACHINE to display information. In this case include the option **/MACHINE 0** in a *<command>* to address the host machine.

<command> **/MACHINE 0**	The host machine (= **0**).

The *<commands>* can, for example, be:

• **Register.view** and **Frame.view**

• **MMU.DUMP**, **MMU.List** and **MMU.SCAN** with the table options **PageTable** and **KernelPageTable**

The Hypervisor Awareness usually adds commands that display additional information about the hypervisor. E.g. new windows that show the internal representation of the guests, internal states and register information. The actual layout of the commands and windows depends on the used hypervisor and awareness features. Please see the manual dedicated to the hypervisor for more information.

In order to access the contents of the host machine, the debugger needs to know the "context" that is the register set (both core registers and MMU registers) of the host machine. TRACE32 will proceed as follows to retrieve the register set of the host machine:

1.   If the selected core is running **MACHINE 0**, then the register set is fetched from the selected core. Remember that the selected core is displayed in the **Cores** field of the TRACE32 state line.

2.   If the selected core is not running MACHINE 0, but another core runs MACHINE 0, TRACE32 uses the register set of this core for MACHINE 0.

3.   If no core is running **MACHINE 0**, then TRACE32 will resort to the rescued register set of the host machine on the selected core.

NOTE:	The exact source of the rescued register set depends on the type of hypervisor and the CPU architecture. The Hypervisor Awareness is responsible to retrieve the contents of the register set. Usually, not all CPU registers of the host machine are rescued.

Guest Machines

When viewing information about a guest machine, you should know the machine ID of the desired guest. machine IDs of guests are always bigger than zero.

Use **TASK.List.MACHINES** to show a table of all created machines, including the hypervisor and all guests.

Whenever an address is given, you can simply use machine ID of the guest to access this guest.

Examples: showing code and data of the guest machine with ID "3"

```
  List.Asm 3:::0x1234     ; show code of machine 3 at address 0x1234
  Data.dump 3:::0x5678    ; show data of machine 3 at address 0x5678
```

Loaded symbols are bound to a given machine. That means, the debugger knows that a specific symbols belongs to a specific machine. When accessing symbols (display code or variables), simply use the symbol; no special action is necessary to specify the guest.

Some commands take the option /MACHINE to display information. in this case include the option **/MACHINE** in a *<command>* to address the guest machine.

<command> **/MACHINE** *<machine_id>*	any guest machine (>= **1**).

The *<commands>* can, for example, be:

• **Register.view** and **Frame.view**

• **MMU.DUMP**, **MMU.List** and **MMU.SCAN** with the table options **PageTable**, **KernelPageTable**, and **IntermedPageTable**

If you loaded a guest OS Awareness, this awareness usually adds commands that display additional information about the guest. E.g. new windows that show the tasks of the guests, internal states and register information. The actual layout of the commands and windows depends on the used guest and awareness features. Please see the manual dedicated to the guest OS for more information.

In order to access the contents of the guest machine, the debugger needs to know the "context" that is the register set (both core registers and MMU registers) of the guest. TRACE32 will proceed as follows to retrieve the register set of the guest machine:

1. If the selected core is running any VCPU of machine *<machine_id>*, then the register set is fetched from the selected core. Remember that the selected core is displayed in the **Cores** field of the TRACE32 state line.

2. If the selected core is not running machine *<machine_id>*, but another core is running any VCPU of machine *<machine_id>,* TRACE32 uses the register set of this core for machine *<machine_id>*. Check the **CORE.List** window to see a list of cores and which machine they're running.

3. If no core is running any VCPU of machine *<machine_id>*, then TRACE32 will resort to the rescued register set of VCPU 0 of machine *<machine_id>*.

NOTE:	The exact source of the rescued register set depends on the type of hypervisor and the CPU architecture. The Hypervisor Awareness is responsible to retrieve the contents of the register set. Usually, not all CPU registers of the host machine are rescued.

Physical Cores

In a hypervisor environment, the "physical" core is the core (and register set) at which the hypervisor is operating on. Guests are working on "virtual" cores (see next chapter). The Register window always shows the current register settings, being it hypervisor or guest. If you explicitly want to see the register set of the hypvervisor, add the /MACHINE 0 option.

CORE.List shows an overview of all physical cores, including their PC and the current running guest/task. The core numbers match the core numbers used by the hypervisor. Double click on one of the cores to change the view of TRACE32 to this core.

See also "Viewing registers" below.

NOTE:	TRACE32 uses a different naming for cores: • "physical core" (1 to n) is the core number in hardware. • "logical core" (0 to n) is the core number used by the software and equals the "physical core" of the hypervisor. See note below. • "virtual core" matches the vCPU of a guest
NOTE:	Use **CORE.ASSIGN** to match the physical cores controlled by TRACE32 to the physical cores used by the hypervisor.

Virtual Cores

The hypervisor assigns "virtual CPUs" (vCPUs) to its guests. This way, a guest can manage its own multi-core environment with less or even more cores than the physical system provides. The hypervisor then schedules the vCPUs to the physical cores. The assignment of a vCPU to a physical core is not fixed, i.e. it may change (migrate) in time.

If a guest is running on a physical core, the Register window of the core shows the actual content of the vCPU currently assigned to this core. If a specific vCPU is currently not running on any core, its register contents are stored in the hypervisor structures. The dedicated Hypervisor Awareness usually allows displaying the contents of preempted vCPUs; the actual command and display for this depends on the Hypervisor Awareness.

Some register related commands, e.g. Register, Frame or MMU, allow to specify the vCPU to address a specific register set.

<command> **/MACHINE** *<machine_id>* **/VCPU** *<vcpu_id>*	a VCPU in a guest machine, *<machine_id>* >= 1.

See also "Viewing registers" below.

MMU Spaces

An "MMU Space", or in short "space", is a collection of code and data that share the same address translation. In OSes, such an MMU space is usually called a "process" (but note that there are exceptions, e.g. ARINC processes). To identify the spaces, the debugger uses the OS Awarenesses. That means, for each OS running in the system where you want to see the spaces (i.e. hypervisor and guests), an appropriate awareness must be loaded.

Each awareness provides a command to display the MMU spaces / processes of this OS. The command and the layout depend on the awareness. Please see the appropriate manual for more information.

For an overview of all spaces known to the debugger, you can use the command **TASK.List.SPACES** .

Tasks

A "Task" in TRACE32 refers to an execution unit with its own register set. Often called "thread" in OSes. To identify the tasks, the debugger uses the OS Awarenesses. That means, for each OS running in the system where you want to see the tasks (i.e. hypervisor and guests), an appropriate awareness must be loaded.

Each awareness provides a command to display the tasks of this OS. The command and the layout depend on the awareness. Please see the appropriate manual for more information.

For an overview of all spaces known to the debugger, you can use the command **TASK.List.tasks** .

Some register related commands, e.g. Register, Frame or MMU, and the Breakpoint commands allow to specify the task to address a specific register set.

<command> **/TASK** *<task>*	a task in a host or guest machine

See also "Viewing registers" below.

Task names (if available) may be a a composed by <machine_name>:::<space_name>::<task_name> to uniquely identify a task amongst the system, If a task name is in itself unique, you can also use only the task name.

Summary

To get an overview of the machines, MMU spaces and tasks, use these commands:

TASK.List.MACHINES	display all machines
TASK.List.SPACES	display all MMU spaces
TASK.List.tasks	display all tasks
TASK.List.TREE	display machines, spaces and tasks in a tree view

The following table summarizes how you can display hypervisor system information about cores, tasks, the host machine, as well as the guest machines and their VCPUs.

Execute a command together with these arguments...	To display information about...
<command> (no further arguments)	the currently selected core.
<command> **/CORE** *<core>*	any core you want.
<command> **/MACHINE 0**	the host machine (= 0).
<command> **/MACHINE** *<machine_id>* **/VCPU** *<vcpu_id>*	a VCPU in a guest machine, *<machine_id>* >= 1.
<command> **/TASK "***<machine_name>***:::***<task_name>***"**	any task in the machine *you* want.

| **NOTE:** | *<command>* **/TASK "***<task_name>***"** displays information about the specified task in the current machine. However, if the specified *<task_name>* is not found on the current machine, an incremental search is performed on all other machines, and the first matching task is displayed.

Therefore it is recommended that you precede a *<task_name>* with the *<machine_name>*.
Result: *<command>* **/TASK "***<machine_name>***:::***<task_name>***"** |

Viewing Registers

You can view the following CPU registers of a hypervisor system in TRACE32:

- The registers of a core

- The registers of a VCPU of a guest machine

- The registers of a task running on a guest machine

Viewing the Registers of a Core

The register sets of cores are read directly from the hardware. You can display the register set of a core with the **Register.view** command:

|	
Register.view	View the register set of the default viewing context. By default, the default viewing context is based on the selected core. The selected core is flagged by a check mark in the **CORE.List** window.
Register.view /CORE *<number>*	View the register set of any core with the **/CORE** *<number>* option.

To view the register set of a core:

- Execute one of the above **Register** commands to display the register set you want to view.

  TRACE32 opens a **Register** window, displaying the requested CPU register set.

  - The window title is the same as the executed command.

  - The background of the window is highlighted in the same color as the respective core in the **CORE.List** window. Thus the background color gives you a visual cue to which core the register set belongs.
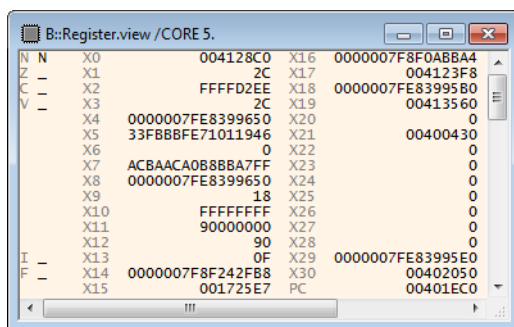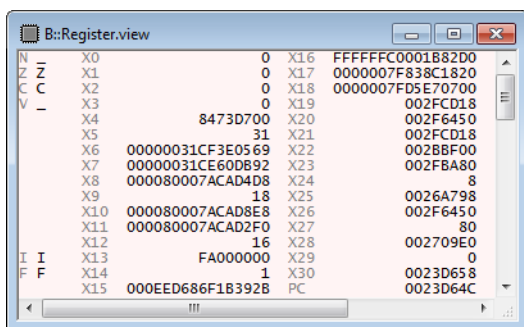
Example:

```
Register.view /CORE 5.
```

Viewing the Registers of a VCPU of a Guest Machine

You can display the register set of a specific VCPU of a guest machine by passing the options **MACHINE** and **VCPU** to the **Register.view** command:

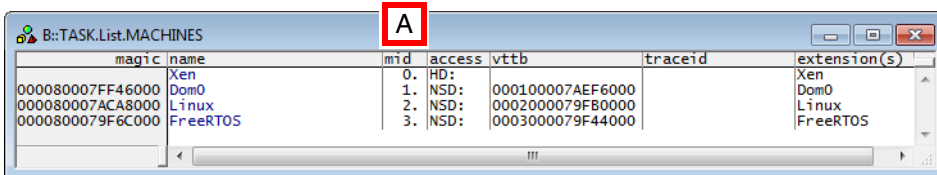|	
Register.view /MACHINE *<machine_id>* **/VCPU** *<vcpu_id>*	View the registers of a VCPU of a guest machine

You can look up the parameters you need in the following places:

- In your PRACTICE script (*.cmm) that loads the TRACE32 Hypervisor Awareness.

- Via the TRACE32 PowerView GUI by taking the steps below.

- In your documentation about the hypervisor system you have designed.

- Some, but not all, hypervisor menus on the TRACE32 main menu bar provide a menu item that is called **Display VCPUs**.

- The TRACE32 Hypervisor Awareness manuals (hv_*<hypervisor>*.pdf)

The VCPU IDs range from 0 to n-1, where n is the number of VCPUs of the guests machine.

To view the registers of a VCPU of a guest machine:

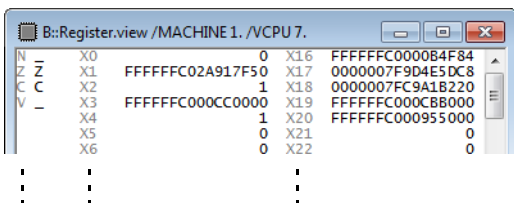1. Open the **TASK.List.MACHINES** window (via the TRACE32 command line).

2. Get the *<machine_id>* you want from the **mid** column [**A**].

3. Look up the *<vcpu_id>* as suggested above.

4. Enter the **Register.view /MACHINE** *<machine_id>* **/VCPU** *<vcpu_id>* command.

   TRACE32 opens a **Register** window, displaying the current register set of the VCPU.

   - The window title is the same as the executed command.

   - The background of the window is white as usual because the window displays a VCPU register set, which is not bound to a specific core.

Example:

```
Register.view /MACHINE 1. /VCPU 7.
```

Viewing the Registers of a Task Running on a Machine

A task which is run by an OS is not always being executed on a VCPU or a physical core. You can view task registers independently of the current execution status of a task. To do that, you need the parameters *<machine_name>* and *<task_name>* of the task running on a guest machine. Pass them to the **Register** commands below:

Register.view /TASK "*<machine_name>***:::***<task_name>*" See step-by-step procedure (A) below.	View the register set of a task WITHOUT switching the default viewing context.

|	
Register.TASK "*<machine_name>***:::***<task>*" **Register.view** See step-by-step procedure (B) below.	Set the specified task as the new default viewing context. View the register set of the default viewing context.

You can look up the parameters you need in the following places:

- In your PRACTICE script (*.cmm) that loads the TRACE32 Hypervisor Awareness.

- Via the TRACE32 PowerView GUI by taking the steps of procedure below.

(A) To view the registers of a task - WITHOUT switching the default viewing context:
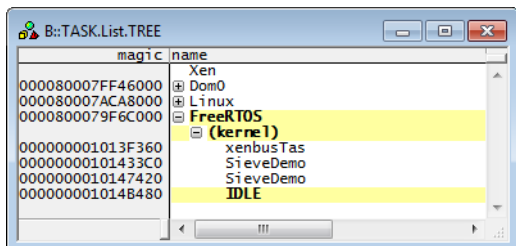
1.   Open the **TASK.List.TREE** window.

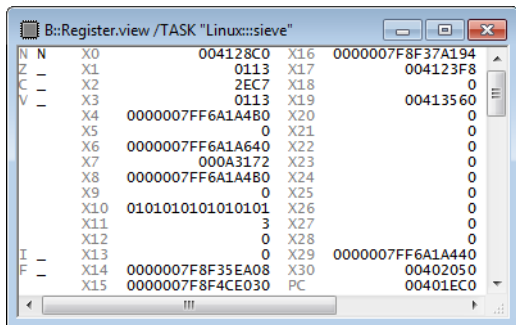2.   Get the *<machine_name>* you want from the first level of the tree.

3.   Get the *<task_name>* you want from the third level of the tree.

4.   Enter the **Register.view /TASK** "*<machine_name>***:::***<task_name>*" command.

    TRACE32 opens a **Register** window, displaying the current CPU register set of the selected task.

    -    The window title is the same as the executed command.

- The background of the window is white as usual because it displays information which is not bound to a specific core.

Example for (A):

```
Register.view /TASK "Linux:::sieve"
```

(B) To view the registers of a task - here we'll switch the default viewing context:

1.   Open the **TASK.List.TREE** window.

2.   Get the *<machine_name>* you want from the first level of the tree.

3.   Get the *<task_name>* you want from the third level of the tree.

4.   Enter the command **Frame.TASK** "*<machine_name>*:::*<task_name>*"

All windows that are related to the current register set will switch to this task.

Viewing Variables

Viewing variables is straigh-forward, as variables are bound to a specific machine and MMU space.

When loading ELF files for a machine, you (or the symbol autoloader, if available) assign the correct machine ID and space ID to the symbol files therein. E.g.

```
Data.LOAD.Elf myApplication 0x1:::0x2:0x0      ; machine 1, space ID 2
```

All variables defined in the ELF file will then be bound to this machine ID and space ID. See **sYmbol.name** for the exact location of the variable symbol. Whenever you're using a variable name, the debugger automatically uses the machine ID and space ID of the associated symbol.

If a symbol is unambiguous, the debugger will try to use symbol of the current context (current machine ID and current space ID), if the symbol is available in this context. You can also identify a specific variable by applying the complete symbol path.

```
; load symbol of process "p1" running on space ID 5 of machine 2:
Data.LOAD.Elf p1.elf 0x2:::0x5:0x0

; load symbol of process "p2" running on space ID 7 of machine 3:
Data.LOAD.Elf p2.elf 0x3:::0x7:0x0

; view variable "myVar" in the current context
Var.view myVar

; view variables "myVar" of p1 and p2
Var.view \\p1\\myVar \\p2\\myVar
```

Viewing the Call Stack

In TRACE32, you can view the stack frames of the cores, the host machine, the guest machines and the VCPUs of a guest machine, as well as the tasks of the various machines. An analysis of the call stacks is useful because it shows you where, and possibly why a specific task is preempted, or at which state a specific guest VCPU was switched.

Frame.view (no further arguments)	View the call stack of the default viewing context. The default viewing context is based on the selected core. The currently selected core is flagged by a check mark in the **CORE.List** window.
Frame.view /CORE *<core>*	View the call stack of any core you want.
Frame.view /MACHINE *<machine_id>* **/VCPU** *<vcpu_id>*	View the call stack of a VCPU in a guest machine. (*<machine_id>* >= **1**).
Frame.view /TASK "*<machine_name>***:::***<task_name>*"	View the call stack of any task in the machine you want.

Frame.TASK "*<machine_name>***:::***<task_name>*"	Set the specified task as the new default viewing context.
Frame.view (no further arguments)	View the call stack of the default viewing context.

You can look up the *<parameters>* in one of following places:

• In your PRACTICE script (*.cmm) that loads the TRACE32 Hypervisor Awareness.

• The **CORE.List** window displays the *<cores>*.

• **TASK.List.MACHINES** window lists the *<machine_ids>* in the **mid** column.

• The **Frame.view** window provides the available combinations of "*<machine_names>***:::***<task_names>*" in the **Task** drop-down list.

• In your documentation about the hypervisor system you have designed.

• Some, but not all, hypervisor menus on the TRACE32 main menu bar provide a menu item that is called **Display VCPUs**.

To view the call stack you want:

• Execute a **Frame.view** command together with the arguments listed in the table above.

    TRACE32 opens a **Frame** window, displaying the requested call stack.

    - The window title is the same as the executed command.

    - The background of the window has the same color as the respective core in the **CORE.List** window. Thus the background color gives you a visual cue to which core the call stack

belongs.

- The background of the window is white as usual if the displayed call stack is not bound to a specific core.

Example:

```
Frame.view /TASK "Linux:::sieve"
```

Next:

# Interpreting/Understanding the Call Stack

About the Background Colors of the Call Stacks

The background of the **Frame.view** window is highlighted in one of the colors from the **CORE.List** window or is white as usual. As you will see from the examples below, the call stack color is governed by the use or non-use of the options **CORE** *<number>* and **MACHINE** *<number>*.

A color from the CORE.List window	The background color gives you a visual cue to which core the call stack belongs.
White	The background is white as usual if the displayed call stack is not bound to a specific core.

The call stack in the **Frame.view** window [**A**] refers to the core you have selected in the **CORE.List** window. As soon as you select a different core in the **CORE.List** window with a double-click, the call stack display and the core color in [**A**] update accordingly.

The call stack in window [**B**] refers to core 5 because you have pinned the window to core 5 with the option **CORE** *<number>*. The background color [**B**] also emphasizes the fact that the call stack displayed refers to core 5 in the **CORE.List** window.
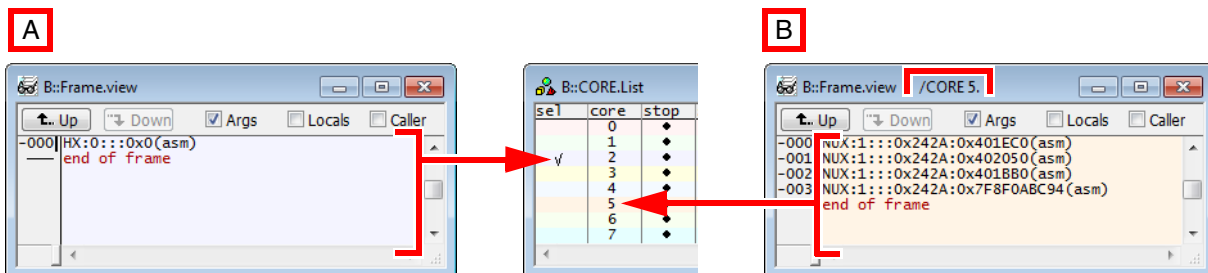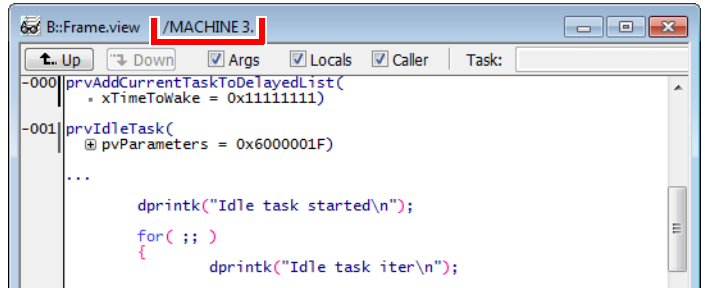

The call stacks in the **Frame.view** windows [**C**] and [**D**] are displayed against a white background because the call stacks are not bound to a specific core. In addition, the windows have been pinned to specific machines with the option **MACHINE** *<number>*.
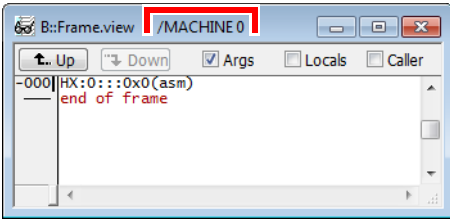
C	Host Machine = 0
D	Guest Machines >= 1

For a general description of the window, see **Frame.view**.

Call Stack of the Tasks

In the **Frame.view** window, you can dynamically set the viewing context to a running or a non-running task, provided you open the window *without* the options **CORE** or **TASK**.

| Running task	A task that is being executed on the selected core is referred to as a running task.
Linux:::sieve	The state line remains light gray - as usual.
Non-running task	A task that is currently **not** being executed on any core is referred to as a non-running task.
Dom0:::ksoftirqd/2	The **Cursor** and the **Task** field in the state line are highlighted in pale brown red to indicate that the viewing context is set to a non-running task.

In either case, all stack-related information that is displayed in a **Frame.view** window will refer to the task. Be aware that this is only for displaying information. When you continue debugging the hypervisor system (with **Step** or **Go**), the debugger will switch back to the viewing context of the running task on the selected core.

However, by opening a **Frame.view** window with the **TASK** option, you can pin the window to the call stack of a specific task on a specific machine. With the **CORE** option, the viewing context is pinned to the specified core - regardless of which task is being executed on that core.

Running Tasks

In the state line, the color gray [**A**] is used to indicate that the viewing context is set to the call stack of a task that is being executed on the selected core 4. The **Task** field tells you the name of the running task [**B**].

Opened without the options, the **Frame.view** window by default displays the call stack [**C**] of the task running on the selected core.
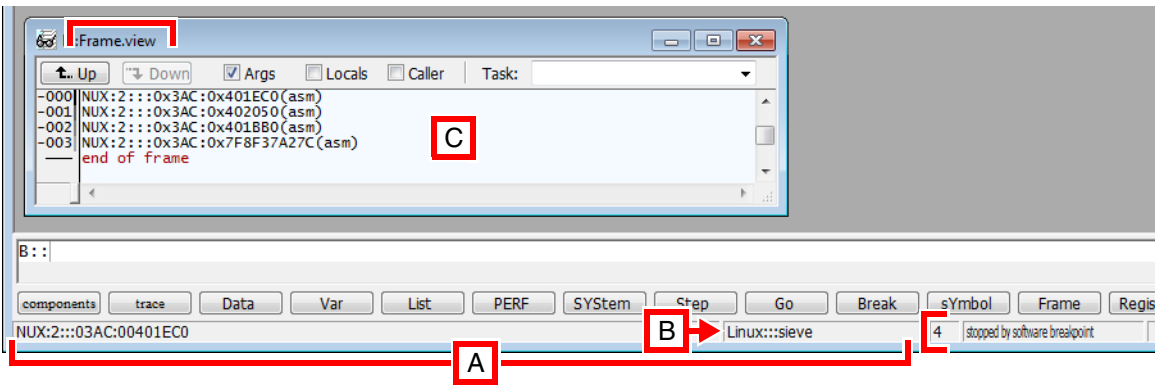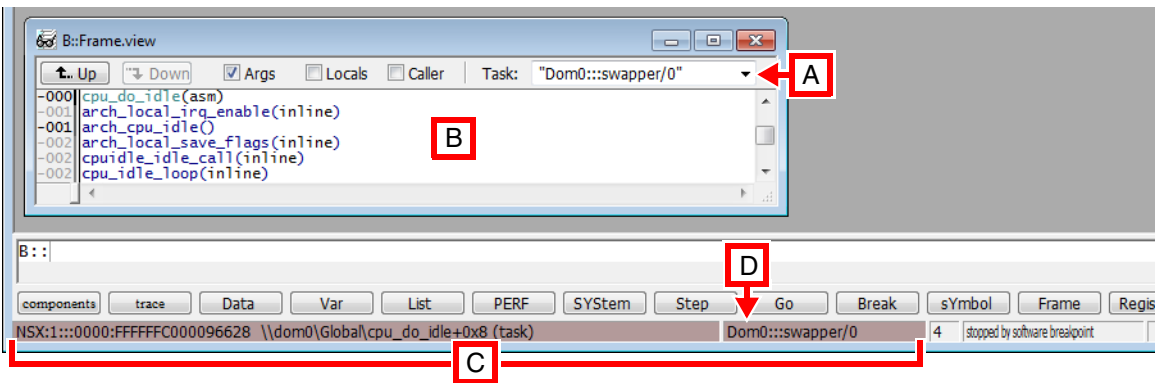
You can switch the viewing context by selecting a different core from the **Cores** drop-down list. As a result, the **Frame.view** window now displays the call stack of the running task on the core you have just selected. The **Task** field again reveals the name of the task. The state line remains gray because task you have just selected also is an running task - just on a different core.

Non-Running Tasks

In the **Frame.view** window, you can view the call stack of any task by selecting it from the **Task** drop-down list [**A**].

When you set the viewing context to a non-running task, the background of the window turns white [**B**] and the **Cursor** and the **Task** field in the state line are highlighted in pale brown red [**C**]. The colors white and pale brown red indicate that the selected task is currently not being executed on any core. The **Task** field tells you the name of the non-running task [**C**].

You can set the viewing context back to the running task by selecting the empty line in the **Task** drop-down list. The color of the state line returns to light gray again.

Going Up or Down in the Call Stacks

The Frame window provides two buttons "Up" and Down" that allow to change the current viewing context to the call frame one level (function call) higher or lower, see also the commands **Frame.Up** and **Frame.Down**. Please note that only the viewing context is set temporarily, the registers on the target are not touched.

Setting Breakpoints

Setting breakpoints follows the same scheme as using addresses of symbols. I.e. when using a symbol (name of function or variable), the breakpoint is automatically set to the machine ID and space ID that is bound to this symbol. When using an address, machine ID and space ID are set explicitly or implicitly according to the address given.

If not explicitly specified, the debugger tries to set program breakpoint as soft breakpoints, by writing a breakpoint instruction into the target code. If this fails, the debugger automatically sets an onchip breakpoint.

Read/write breakpoints are always set as onchip breakpoints.

Soft Breakpoints

Software breakpoints are implemented by writing a breakpoint instruction into memory at the location, where the program should halt. The debugger bypasses the MMU, which means, it can even write software breakpoints into address ranges that are set "read only" by the MMU. Because the debugger does the MMU translation itself, you can set software breakpoints on any code location that is known to the VM MMU tables of the hypervisor and process MMU tables of the OSes; even if the according virtual page is not addressable by the CPU itself.

If the code area of the software breakpoint is not in memory, e.g. not yet loaded by demand paging, or swapped out, then the debugger is not able to write the breakpoint instruction. In this case only an onchip breakpoint will work.

If a software breakpoint is set into a code area that is shared between different threads and/or processes (e.g. library code), the breakpoint will hit, regardless which task runs onto the breakpoint. If you want to have the breakpoint hit only if a specific task runs onto it, use the Break.Set /TASK option. However, be aware that the breakpoint will halt anyway, but the debugger resumes execution if the halted task is not the selected one. Of course, this means a significant delay in run-time.

Onchip Breapoints

Onchip breakpoints are set in the core itself. The hardware of the CPU provides special registers that allow to enter debug mode, if the program counter equals one of these registers. This means, the CPU halts the execution as soon as the onchip breakpoint location is hit, regardless which VM, process or thread is running. I.e. you may set an onchip breakpoint onto process 1 of guest 1, and the CPU may halt at the same address but in process 2 of guest 2.

If you want to have the breakpoint hit only if a specific task runs onto it, use the Break.Set /TASK option. However, be aware that the breakpoint will halt anyway, but the debugger resumes execution if the halted task is not the selected one. Of course, this means a significant delay in run-time.

You can instruct the debugger to ensure that the breakpoint matches to the set machine ID, zone or ASID. Again, this may cause a halt anyway, but the debugger resumes execution if the halted location doesn't match the set machine ID, zone or ASID. Use the commands **Break.CONFIG.MatchMachine** / **Break.CONFIG.MatchZone** / **Break.CONFIG.MatchASID** to set this.

On some architectures, the core hardware allows to combine the breakpoint with the content of internal registers. E.g. on Arm, you can link the breakpoint logic to the context ID register, and the machine ID register. If the OS and hypervisor use these registers accordingly, guest and process dependent onchip breakpoints may be set without intrusion. Enable these comparisons with **Break.CONFIG.UseContextID** / **Break.CONFIG.UseMachineID**.

Using OS Awarenesses

Each machine, i.e. both, the hypervisor and each guest, may run an own operating system. These OSes may be even different. In order to support each of the OSes, TRACE32 allows to load an OS Awareness to any machine. If the hypervisor is an OS in its own, the Hypervisor Awareness also serves as OS Awareness to machine 0.

Use the **EXTension.LOAD** command to load an OS Awareness and apply option **/MACHINE** with the appropriate machine. The option **/NAME** allows to rename the extension, in case of identical awareness files.

EXTension.LOAD *<extension>* [**/MACHINE** *<machine>*] [**/NAME** *<name>*]

Each OS Awareness comes with a dedicated menu. When loading these, you will get an own menu entry for each machine.

Example: Load awareness for Hypervisor Xen, two Linux guests and a FreeRTOS guest:

```
; load awareness for Hypervisor Xen
EXTension.LOAD ~~/demo/arm/kernel/xen/xen.t32 /Machine 0
MENU.ReProgram ~~/demo/arm/kernel/xen/xen.men

; load awareness for first Linux guest, called "Dom0"
EXTension.LOAD ~~/demo/arm/kernel/linux/awareness/linux.t32 \
    /Machine 1 /Name "Dom0"

; load awareness for second Linux guest, called "DomU"
EXTension.LOAD ~~/demo/arm/kernel/linux/awareness/linux.t32 \
    /Machine 2 /Name "DomU"

; load awareness for FreeRTOS guest
EXTension.LOAD ~~/demo/arm/kernel/freertos/freertos.t32 \
    /Machine 3 /Name "FreeRTOS"
```

Specific OS Awareness commands have the form of **TASK.<command>** (see their manuals). E.g. the Linux specific task list can be viewed with the command **TASK.DTASK**. If you loaded several awareness files, especially when having guests with the same awareness, these commands may no longer be unique. The debugger then uses the first awareness that defines the entered command.

To address a specific awareness (i.e. a specific machine), the OS specific **TASK** commands can be selected with an infix that represents the addressed extension:

> **TASK.***<extension>***.***<command>* [**arguments**]

Example: guest task lists for the above loaded extensions:

```
; task list for machine 1, Linux guest "Dom0"
TASK.Dom0.DTASK

; task list for machine 2, Linux guest "DomU"
TASK.Domu.DTASK

; task list for machine 3, FreeRTOS guest
TASK.FreeRTOS.TaskList
```

Viewing Page Tables

The commands **MMU.List** and **MMU.DUMP** allow to display the contents of all page tables on the system. **MMU.List** provides a compact display while **MMU.DUMP** a page wise display of the MMU translation tables.

With **PageTable** as parameter, the current page table is displayed

```
MMU.List PageTable
```

The current page table corresponds to the Hypervisor page table if the current core is stopped in the hypervisor, otherwise to the current page table of the current machine, which can then be the kernel page table or task page table.

By specifying a **/MACHINE** option, it is possible to display the current page table for a specific machine:

```
MMU.List PageTable /MACHINE 1
```

The intermediate page table can be displayed using **IntermedPageTable** as parameter. A **/MACHINE** option can also be used:

```
MMU.DUMP IntermedPageTable /MACHINE 1
```

The Kernel page table for a specific machine can be displayed using the **KernelPageTable** parameter.

```
MMU.List KernelPageTable
MMU.List KernelPageTable /MACHINE 2
```

The task page table for a specific guest user task can be displayed using the TaskPageTable parameter followed by a task identifier:

```
MMU.List TaskPageTable /TASK "mytask"
MMU.DUMP KernelPageTable /TASK "mytask" /MACHINE 2
```