LAUTERBACH
DEVELOPMENT TOOLS

TRACE32 Concepts

Release 02.2026

MANUAL

TRACE32 Concepts

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ..... 🗁

TRACE32 Concepts 1

History 5

Access Classes 6

Access Class Expansion 12

Address Spaces 13

Zones 13

Zone Spaces 14

MMU Space 15

Machine Spaces 16

Address Types 17

Absolute Physical Address 18

Guest Logical Address 18

Host Logical Address 19

Intermediate Address (synonym: guest physical address) 19

Logical Address (synonyms: virtual address, effective address) 19

Physical Address (synonym: real address) 20

Awareness 21

Hypervisor Awareness 21

OS Awareness 21

Build Path 22

Chip Timestamp 23

Common Address Range 23

Cycle-accurate Tracing 23

CombiProbe 24

Extension 24

Hypervisor 24

integrated Asymmetrical Multi-Processing (iAMP) 25

Hypervisor Machine ID 26

Machine 28

Guest Machine (synonym: virtual machine, VM) 28
Host Machine 28

Magic Number **29**
Machine Magic Number 29
Space Magic Number 29
Task Magic Number 30

MCDS in Infineon TriCore AURIX MCUs **30**

Memory Management Unit (MMU) **31**

Multicore Debugging **31**

Multiprocessor Debugging **31**

Observabiltiy Gap **31**

Order of Source Code Lines **32**

OS-aware Debugging **34**
OS (No Dynamic Memory Management) 34
AUTOSAR/OSEK Operating Systems 35
OS+MMU (Dynamic Memory Management) 35

OS-aware Tracing **36**
Task Switch by Tracing Special Write Accesses 37
Task Switch by Tracing Task Switch Packets 40

Process **44**

RTOS **44**

Run-time Memory Access **45**

Sample-based Profiling **53**

OS Space ID **54**

Spot Breakpoint **55**

StopAndGo Mode **57**

Symmetrical Multi-Processing (SMP) **58**

Task **58**

Thread **58**

TRACE32 Virtual Memory **59**

Trace Errors **62**
TARGET FIFO OVERFLOW 62
FLOWERROR 63

Trace Sources **66**

Tool Timestamp **67**

TRACE32 Concepts

Version 03-Apr-2026

History

20-Jul-23	Initial version of this manual.

Access Classes

Access classes are used to specify how TRACE32 PowerView accesses memory, registers of peripheral modules, addressable core resources, coprocessor registers and the **TRACE32 Virtual Memory**.

Addresses in TRACE32 PowerView consist of:

- An access class, which consists of one or more letters/numbers followed by a colon (:)
- A number that determines the actual address

Examples for access classes are:

- The program memory class
- The data memory class

Program Memory Classes

The most often used letter to identify the program memory class is **P:**.

```
List P:0x4568                          ; display the source code starting
                                       ; at Program address 0x4568
```

There are other letters used by core architectures that provide more than one instruction set encoding. Here a few examples:

```
List R:0x456378                        ; R representing ARM instruction
                                       ; set encoding for the
                                       ; ARM architecture

List T:0x456378                        ; T representing THUMB instruction
                                       ; set encoding for the
                                       ; ARM architecture

List V:0x456378                        ; V representing VLE instruction
                                       ; set encoding for the
                                       ; Power Architecture
```

The available program memory classes are dependent on the processor architecture in use. Therefore refer to the **Access Class/Memory Class** section of your **Processor Architecture Manual** for more details.

Data Memory Classes

The letter **D:** is normally used to identify the data memory class.

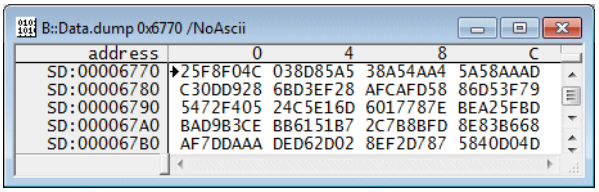
```
Data.dump D:0x6770                      ; display a hex dump starting at
                                        ; Data address 0x6770
```

Other letters are used only in some rare cases.

```
Data.dump X:0x6770                      ; use X-Bus to access data memory
                                        ; MMDSP architecture
```

Access Classes and Commands

TRACE32 PowerView always displays the access class information in its display windows. **SD** stands for "Supervisor Data" in the example below.

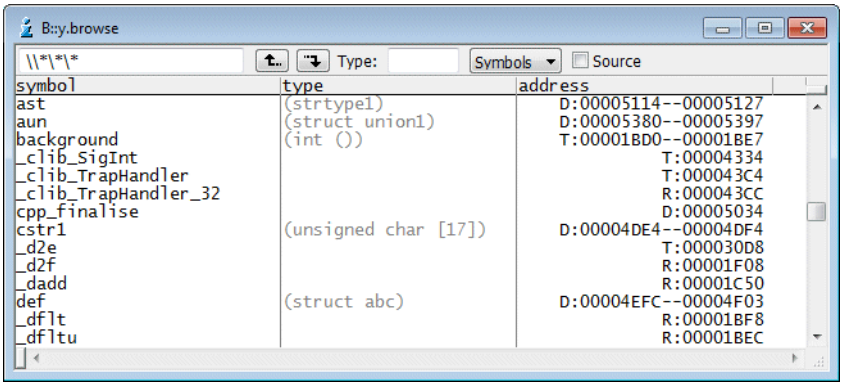


If access classes are omitted from the command input the default access class for the command is used.

```
List 0x456378                           ; the default access class for a
                                        ; source code listing is the
                                        ; program memory class

Data.dump 0x6770                        ; the default access class for a
                                        ; hex dump is the data memory
                                        ; class
```

The addresses of debug symbols include the applicable access class.




```
Data.dump background                    ; display a hex dump starting at
                                        ; program address 0x1BD0

                                        ; THUMB instruction set encoding
                                        ; for the ARM architecture used
```

Some commands apply only to a specific access class.

```
Trace.STATistic.sYmbol                  ; analyze the execution time in
                                        ; different symbol regions
                                        ;
                                        ; this command uses only program
                                        ; memory class symbols for its
                                        ; analysis
```

Access Classes for Core Resources

Frequently used access classes for core resources are the cache access classes:

IC	Instruction Cache
DC	Data Cache
L2	Level 2 Cache
NC	No Cache (access with caching inhibited)
...	

```
Data.dump DC:0x6770                     ; display a hex dump starting at
                                        ; address 0x6770, get the
                                        ; information from the Data Cache

Data.dump DC:flags                      ; display a hex dump starting at
                                        ; the address represented by the
                                        ; debug symbol flags, get the
                                        ; information from the Data Cache

Data.dump NC:0x6770                     ; display a hex dump starting at
                                        ; address 0x6770, get the
                                        ; information from the physical
                                        ; memory (No Cache)
```

The available access classes for core resources are highly dependent on the processor architecture in use. Therefore refer to the **Access Class/Memory Class** section of your **Processor Architecture Manual** for more details.

Access Classes for Coprocessor Registers

Some processor architectures use access classes to access the coprocessor registers.

```
                                        ; example for the ARM architecture
PER.Set C15:0x1 %Long 0x2               ; write 0x00000002 to
                                        ; Coprocessor 15 register 1

                                        ; example for the MIPS architecture
Data.dump CP0:0x25                      ; display the contents of
                                        ; register 5 of register set 1
                                        ; of CoProcessor 0

                                        ; register set start address is
                                        ; n * 0x20
```

The available access classes for coprocessor registers are highly dependent on the processor architecture in use. Therefore refer to the **Access Class/Memory Class** section of your **Processor Architecture Manual** for more details.

Access Class Attributes

Access class attributes are used to supply TRACE32 PowerView with more details on the access. Access class attributes have to be placed in front of the access class.

Examples:

E	**Run-time access** (non-intrusive if possible, otherwise intrusive)
A	Physical address (bypass MMU)
S	Supervisor memory (privileged access)
U	User memory (non-privileged access)
Z	Secure access (e.g. for ARM TrustZone)
N	Non-secure access (e.g. for ARM TrustZone)

```
Data.dump A:0x29876                     ; display a hex dump starting at
                                        ; physical address 0x29876

Data.dump AD:0x29876                    ; same as previous

Data.dump ADC:0x29876                   ; display a hex dump starting at
                                        ; physical address 0x29876, get
                                        ; the data from the data cache
```

The available access classes are highly dependent on the processor architecture in use. Therefore refer to the **Memory Class** section of your **Processor Architecture Manual** for more details.

Access Classes in TRACE32

There are two access classes specific for TRACE32.

1. VM: **TRACE32 Virtual Memory**.
2. USR: User Access Class

   The USR: access class can be used to read or write resources that require a complex access mechanism (e.g. indirectly addressed registers). The access is performed by a target algorithm. For details refer to the command **Data.USRACCESS**.

   Please be aware that NAND FLASH devices, serial FLASH devices and EEPROMs can be accessed by using the command **FLASHFILE**.

Access Class Expansion

If you omit access class specifiers in an access class combination, then TRACE32 will make an educated guess to fill in the blanks. The access class is expanded based on:

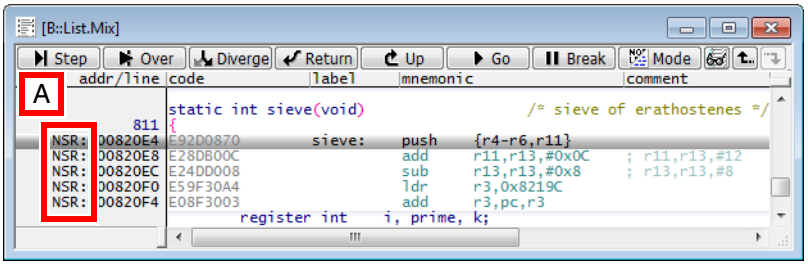
- The current CPU context (architecture specific)
- The used window type (e.g. **Data.dump** window for data or **List.Mix** window for code)
- Symbol information of the loaded application (e.g. combination of code and data)
- Segments that use different instruction sets
- Debugger specific settings (e.g. **SYStem.Option.***)

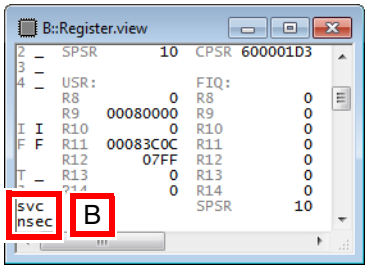
Examples: Memory Access through CPU

Let's assume the CPU is in non-secure supervisor mode, executing 32-bit code.

User input at the command line	Expansion by TRACE32	These access classes are added because...
`List.Mix` (see also illustration below)	**NSR**:	**N**: … the CPU is in non-secure mode. **S**: … the CPU is in supervisor mode. **R**: … code is viewed (not data) and the CPU uses 32-bit instructions.
`Data.dump A:0x0`	A**NSD**:0x0	**N**: … the CPU is in non-secure mode. **S**: … the CPU is in supervisor mode. **D**: … data is viewed (not code).
`Data.dump Z:0x0`	Z**SD**:0x0	**S**: … the CPU is in supervisor mode. **D**: … data is viewed (not code).
NOTE: 'E' and 'A' are not automatically added because the debugger cannot know if you intended a run-time or physical access.		

Your input, here `List.Mix` at the TRACE32 command line, remains unmodified. TRACE32 performs an access class expansion and visualizes the result in the window you open, here in the **List.Mix** window.





A TRACE32 makes an educated guess to expand your *omitted* access class to "NSR".

B Indicates that the CPU is in non-secure supervisor mode.

Address Spaces

TRACE32 assumes a single linear *address space* for all CPU modes of RISC processors by default. If this is not the case, TRACE32 has to be configured to recognize multiple overlapping address spaces. This is necessary to allow TRACE32 to load symbol and debug information per address space and to maintain separate MMU translation tables for each address space.

TRACE32 supports three types of address spaces.

- Zone Spaces
- MMU Spaces
- Machine Spaces

Zones

Subject area: Address translation, Symbols

Zones are address spaces in TRACE32 which are used to keep symbols and **MMU/TRANSlation** setups separate from each other. Zones are characterized by a dedicated MMU and register set.

We use the zone concept for two purposes:

- A CPU operation mode with dedicated MMU and register set is defined as a zone.
- In hypervisor-based environments, each machine (guest or host machines) is a zone because each machine has its own address translation and register set.

By default, there is only one zone in TRACE32. Multiple zones are made available in TRACE32 by setting **TRANSlation.CONFIG.ZoneSPACES** or **TRANSlation.CONFIG.MACHINESPACES** or both to **ON**:

ZoneSPACES ON	Defines each CPU operation mode with an individual address space as an individual zone.
MACHINESPACES ON	Defines the host machine and each guest machine as individual zones.

Formats of addresses with zones:

In TRACE32, the zone of an address is fully defined through:

- The access class (only if **TRANSlation.CONFIG.ZoneSPACES** is set to **ON**)
- Followed by the machine ID (only if **TRANSlation.CONFIG.MACHINESPACES** is set to **ON**)

Thus, the format of a TRACE32 address with a `<zone>` looks like this:

- `<access_class>:<machine_id>:::<address_offset>`
- `<access_class>:<machine_id>:::<space_id>::<address_offset>`

Example: `N:2:::0x123::0xC0000000`

Zone Spaces

Subject area: Address translation, Symbols

A *zone space* is the address space of a zone. Zones are identified within TRACE32 by preceding an address with the appropriate access class.

```
TRANSlation.CONFIG.ZoneSPACES ON

; load the debug symbols for the hypervisor zone
; the hypervisor zone is represented by the access class H:
; preceding the actual address

Data.LOAD.Elf xen-syms H:0x0 /NoCODE
```

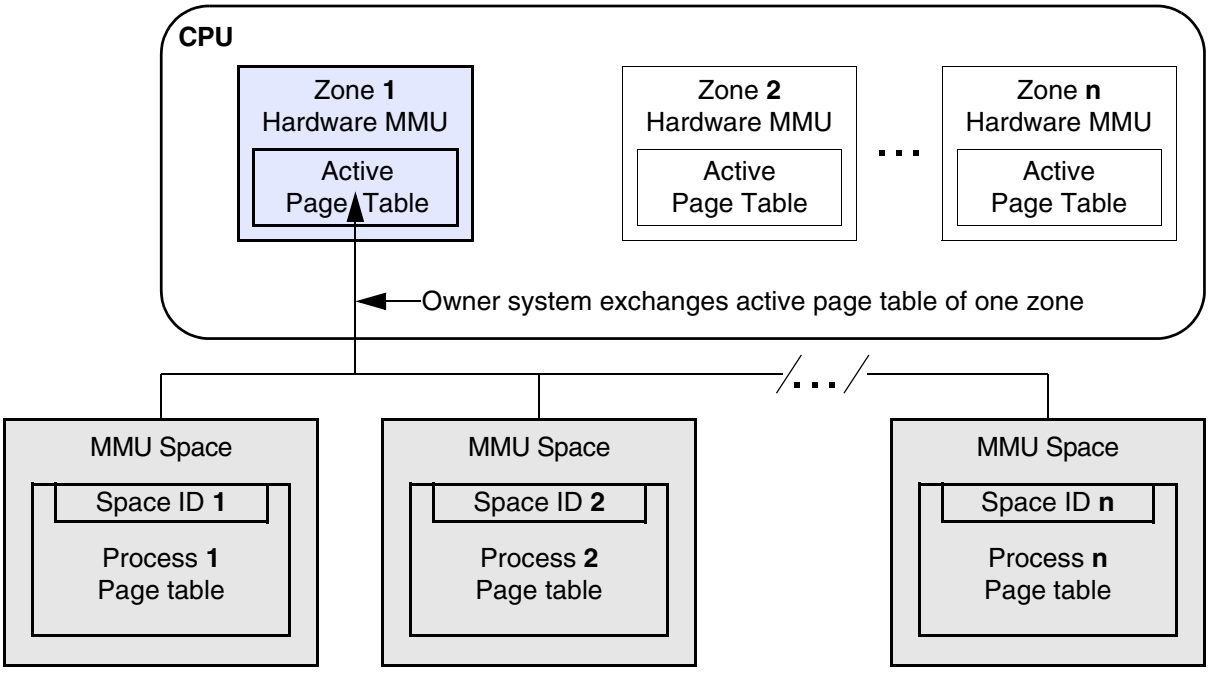
Refer to **TRANSlation.CONFIG.ZoneSPACES** command description for details.

MMU Space

Subject area: Address translation, OS Awareness, Symbols

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 **TRANSlation.CONFIG.MMUSPACES**.

For more information, see space ID.

Machine Spaces

Subject area: Address translation, Hypervisor Awareness, Symbols

TRACE32 uses the term *machine spaces* if a hypervisor is used to manage virtual machines. TRACE32 assigns the machine ID 0 to the hypervisor and machine IDs greater than 0 to the guest machines.

NOTE:	Machine spaces, MMU spaces and zone spaces are often used concurrently.

In TRACE32, machine spaces are enabled with the command **TRANSlation.CONFIG.MACHINESPACES**.

```
TRANSlation.CONFIG.MACHINESPACES ON
; load the debug symbols for the FreeRTOS operating system
; the machine ID is a number preceding the actual address
; it is separated from the address by three colons

Data.LOAD.Elf ../FreeRTOS/FreeRTOS.elf N:3:::0x0 /NoClear /NoCODE
```

Memory-mapped address types in systems with MMU (non-virtualized systems)

In systems with an MMU, generally two types of memory mapped addresses exist:

- Logical addresses
- Physical addresses

When the MMU is enabled, all instruction and data accesses done by the CPU refer to logical addresses. The MMU translates each logical address to a physical address. The resulting physical address becomes visible on the system's memory bus and is used to access memory contents or memory mapped peripheral registers. Only such logical addresses which match a valid translation entry in the MMU can be translated to physical addresses.

If a system does not have an MMU or the MMU is not enabled, the addresses used by the CPU for instruction or data accesses go to the system's memory bus without modification. In this case, all memory accesses done by the CPU use physical addresses directly.

There are systems which are more complex. They have more than only one logical and one physical address space:

In some systems the translation done by the MMU depends on the mode of the CPU. Such a CPU uses an independent logical address space per CPU mode.

Examples of such systems would be:

- ARM CPUs with TrustZone extension, where code can be executed in secure mode or in non-secure mode. The translation of logical addresses to physical addresses can be configured individually for code execution from secure mode and non-secure mode.
- Intel® CPUs where the address translation for code execution in normal mode and code execution in system management mode can be configured independently.

Note that there is usually only one physical address space, but there are systems where more than one physical address space exists. An example would be ARM CPUs with TrustZone extension where the system manufacturer may chose to implement independent physical addresses for secure mode and for non-secure mode.

Systems with multi-step translation processes

There are systems where the translation from logical to physical addresses uses two or more steps. Thus, we have to deal with different address types, depending on where in the translation process we look at.

An example would be Intel® CPUs where in protected mode protected mode addresses are translated to linear addresses in the first translation step. The linear addresses are finally translated to physical addresses in the second step.

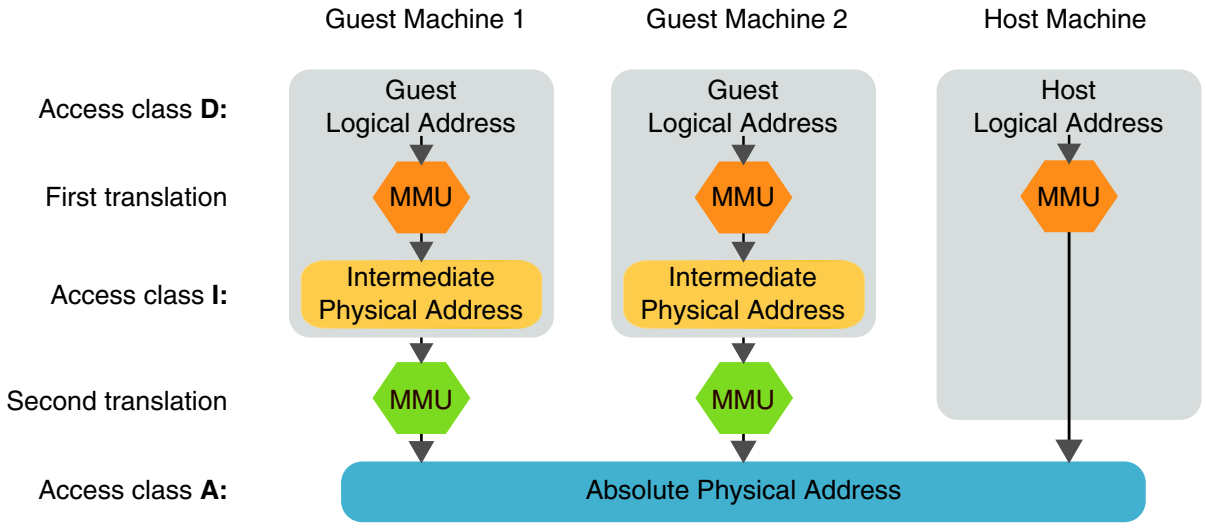
Memory-mapped address types in virtualized systems

In hardware-virtualized systems running under the control of a hypervisor, two translation steps are required to translate guest logical addresses to absolute physical addresses.

1. In the first translation step, a translation table owned by the guest operating system is used to translate a guest logical address to an intermediate address (access class **I:** for intermediate).
2. In the second translation step, a translation table owned by the hypervisor is used to translate the intermediate address to an **a**bsolute physical address (access class **A:** for absolute).

A separate translation table is used to translate a host logical address to an absolute physical address. The code and data sections of the hypervisor operate on host logical addresses.

Figure 1:



Absolute Physical Address

Subject area: Hypervisor Awareness

An absolute physical address is the technical counterpart to logical address. The term *absolute physical address* is used in systems involving virtualization where it is necessary to make a distinction between intermediate addresses and absolute physical addresses.

See also Figure 1.

Guest Logical Address

Subject area: Address translation, Hypervisor Awareness

Logical address of a guest machine within a virtualized system.

For details, see logical address.

See also Figure 1.

Host Logical Address

Subject area: Address translation, Hypervisor Awareness

Logical address of a host machine within a virtualized system.

For details, see logical address.

See also Figure 1.

Intermediate Address (synonym: guest physical address)

Subject area: Address translation, Hypervisor Awareness

A synonym used by some chip manufacturers is: *guest physical address*

- The **General Commands Reference Guides** and the **Function** manuals of TRACE32 use the term *intermediate address*.
- The **Processor Architecture Manuals** of TRACE32 use the chip manufacturer's preferred terminology.

A physical address of a guest machine within a virtualized system is referred to as *intermediate address*. TRACE32 uses the access class **I:** for intermediate addresses.

Intermediate addresses usually differ from absolute physical addresses. A dedicated address translation step is needed to translate an intermediate address to an absolute physical address.

See Figure 1.

Logical Address (synonyms: virtual address, effective address)

Subject area: Address translation

Synonyms used by some chip manufacturers are: *virtual address*, *effective address*

- The **General Commands Reference Guides** of TRACE32 use the term *logical address*.
- The **Processor Architecture Manuals** of TRACE32 use the chip manufacturer's preferred terminology.

In CPUs that have a memory management unit (MMU), a distinction is made between logical addresses and physical addresses. Code execution and data fetching by the CPU core use a logical address to describe the memory location which is accessed. The MMU translates the logical address of such a memory access to a physical address before it is finally sent to the system memory bus.

Physical Address (synonym: real address)

Subject area: Address translation

A synonym used by some chip manufacturers is: *real address*

A *physical address* is the technical counterpart to *logical address*.

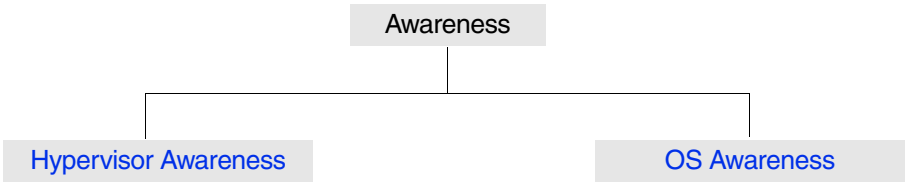
For details, see logical address.

Awareness

Subject area: Hypervisor Awareness, OS 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.



Hypervisor Awareness

Subject area: 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**.

OS Awareness

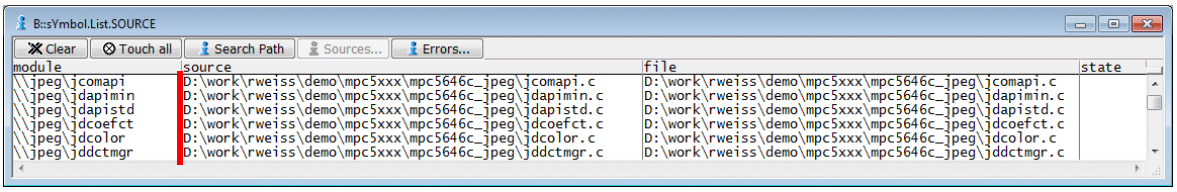
Subject area: OS Awareness

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**.

Build Path

If a files with debug information is loaded (**Data.LOAD.<sub_cmd>**), this file also provides the paths for the high-level source files as they were on the build machine.

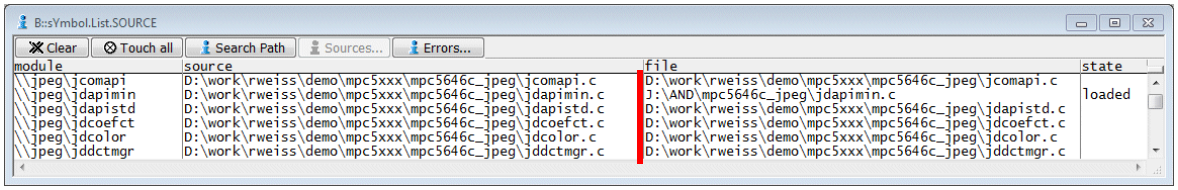


The **source** column in the **sYmbol.List.SOURCE** window shows the build paths.

If TRACE32 is not running on the build machine, the build paths may not be valid and have to be adjusted for the debug host. The adjusted paths are called the debug paths.

TRACE32 includes some auto-adjustments, but also provides various commands for adjusting the paths. Examples are:

Data.LOAD.Elf *<file>* **/CYGDRIVE**	Load *.elf file, convert cygdrive paths to Window paths.
Data.LOAD.Elf *<file>* **/RelPATH**	Load *.elf file with relative paths only.
sYmbol.SourcePATH.SetDir *<directory>*	Define directory as direct search path for source files.
sYmbol.SourcePATH.SetBaseDir *<directory>*	Provide start of source paths directly.
sYmbol.SourcePATH.Translate *<invalid_part> <correct_part>*	Translate *<invalid_part>* of source file paths to *<correct_part>*.



After a file is loaded, the **file** column in the **sYmbol.List.SOURCE** window shows its debug path.

Chip Timestamp

The onchip trace infrastructure adds timestamp information to its generated trace information. Examples are:

- Cycle-accurate tracing for the ARM/Cortex architectures. For details refer to **"Arm ETM Trace"** (trace_arm_etm.pdf).
- Global timestamps for ARM CoreSight. For details refer to **"Arm ETM Trace"** (trace_arm_etm.pdf).
- Ticks, relative and absolute timestamps for the TriCore architecture. For details refer to **"MCDS User's Guide"** (mcds_user.pdf).
- Timestamps for the Nexus Messages. For details refer to **"Training MPC5xxx/SPC5xx Nexus Tracing"** (training_nexus_mpc5500.pdf).

Common Address Range

Subject area: Address translation, OS Awareness

Common address ranges are ranges of logical addresses belonging to the kernel. The common address ranges contain code or data which is shared by various processes.

When the address of a memory access falls into a common address range, TRACE32 uses the kernel address translation (and not the task page table of the current process). Internally, TRACE32 always uses the kernel space ID 0x0000 to find the translation of a common address.

In Linux, shared kernel modules and libraries must be declared as common address range.

In TRACE32, common address ranges are defined with the command **TRANSlation.COMMON**.

Cycle-accurate Tracing

The trace generation logic does not only export which instructions were executed, it also exports the number of clocks an instruction took to execute.

CombiProbe

Subject area: TRACE32 hardware

The CombiProbe is mainly used on Cortex-M derivatives or in case a system trace port is available because it includes besides the debug interface a 4-bit wide trace port which is sufficient for Cortex-M program trace or for system trace.

Extension

Subject area: 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**.

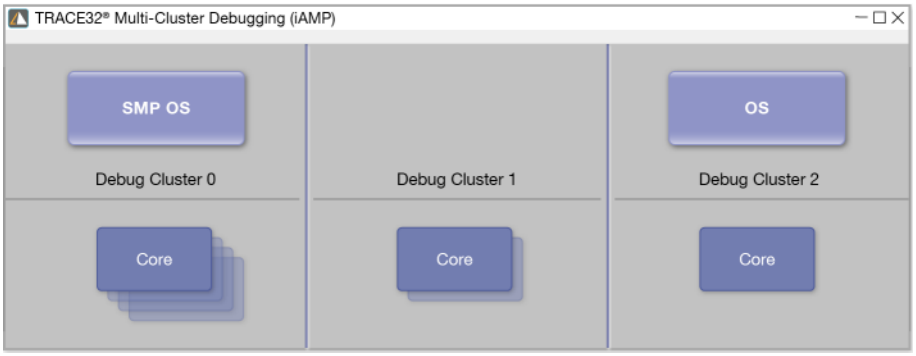
Hypervisor

Subject area: Hypervisor Awareness

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

integrated Asymmetrical Multi-Processing (iAMP)

iAMP allows debugging of multiple cores that share the same instruction set and are SMP (Symmetric Multiprocessing) capable within a single TRACE32 instance. It is typically used in scenarios with multiple OS instances, a single global physical address space, and individual OSes maintaining their own set of virtual address spaces.



Within a TRACE32 instance, debug clusters are formed and managed as machines. Each machine has its machine ID, its own symbol space and, if applicable, its own operating system.

Hypervisor Machine ID

Subject area: Hypervisor Awareness

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`

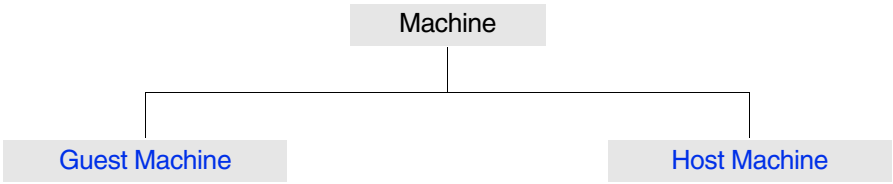
Notes:

- Machine IDs can only be used if a TRACE32 Hypervisor Awareness is loaded with the command **EXTension.LOAD**.
- Use command **TRANSlation.CONFIG.MACHINESPACES ON** to enable machine IDs in TRACE32.

Machine

Subject area: Hypervisor Awareness

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



Guest Machine (synonym: virtual machine, VM)

Subject area: Hypervisor Awareness

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.

Host Machine

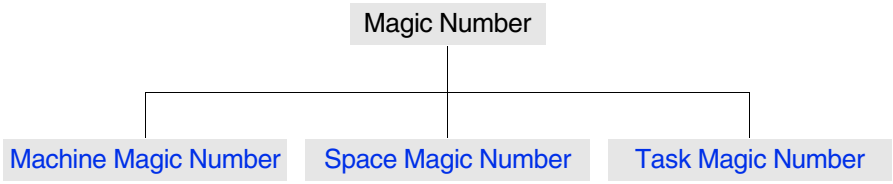
Subject area: Hypervisor Awareness

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.

Magic Number

Subject area: Hypervisor Awareness, OS Awareness



Machine Magic Number

Subject area: Hypervisor Awareness

A machine magic number is an arbitrary hex value, used by TRACE32 to uniquely identify a machine (host machine or guest machine). The meaning of the value depends on the Hypervisor Awareness; often it refers to the guest control block of the hypervisor or to the machine ID.

<machine_magic>	**Parameter Type**: Hex value. **Range**: machine magic number > **`0xFF`** Machine magic numbers are displayed, for example, in the **magic** column of the **TASK.List.MACHINES** window as hex values.

Space Magic Number

Subject area: OS Awareness

A space magic number is an arbitrary hex value used by TRACE32 to uniquely identify an MMU space. The meaning of the value depends on the OS Awareness; often it refers to the process control block of the target OS or to the space ID.

<space_magic>	**Parameter Type**: Hex value. Space magic numbers are displayed, for example in the **magic** column of the **TASK.List.SPACES** window.

Task Magic Number

Subject area: OS Awareness

The task magic number is an arbitrary hex value, used by TRACE32 to uniquely identify a task of an operating system. The meaning of the value depends on the OS Awareness; often it refers to the task control block of the target OS or to the task ID.

<task_magic>	**Parameter Type**: Hex value. **Example**: `TASK.select 0xEFF7B040`

MCDS in Infineon TriCore AURIX MCUs

28-Aug-2023

MCDS (Multi Core Debug Solution) is a peripheral block of AURIX MCUs that implements trace and trigger functions. This peripheral block can be used to debug, trace, profile and verify what is happening in the cores and MCU internal buses. Processor Observation Blocks (POBs) and Bus Observation Blocks (BOBs) both make this possible. Observation Blocks must be assigned to cores and bus slaves to generate trace information:

- Program trace (executed instructions) and data trace (executed memory accesses) for the observed cores.
- Data trace (performed bus transfers) for the observed bus slaves.

MCDS is available in different variants: MCDS, MCDSlight and miniMCDS. The TRACE32 function **MCDS.Module.NAME()** can be used to find out which variant is deployed in the TriCore AURIX under debug. The individual MCDS variants differ in the following:

- The number of Processor Observation Blocks
- The number of Bus Observation Blocks

The TRACE32 Command **MCDS.INFO** provides details about the MCDS variant deployed in the TriCore AURIX under debug.

Memory Management Unit (MMU)

Subject area: Address translation

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.

Multicore Debugging

Multiple cores are in one SoC (System-on-Chip), all cores share the same debug port.

Multiprocessor Debugging

Multiple processors/chips are concurrently debugged, each processor/chip has its own debug port.

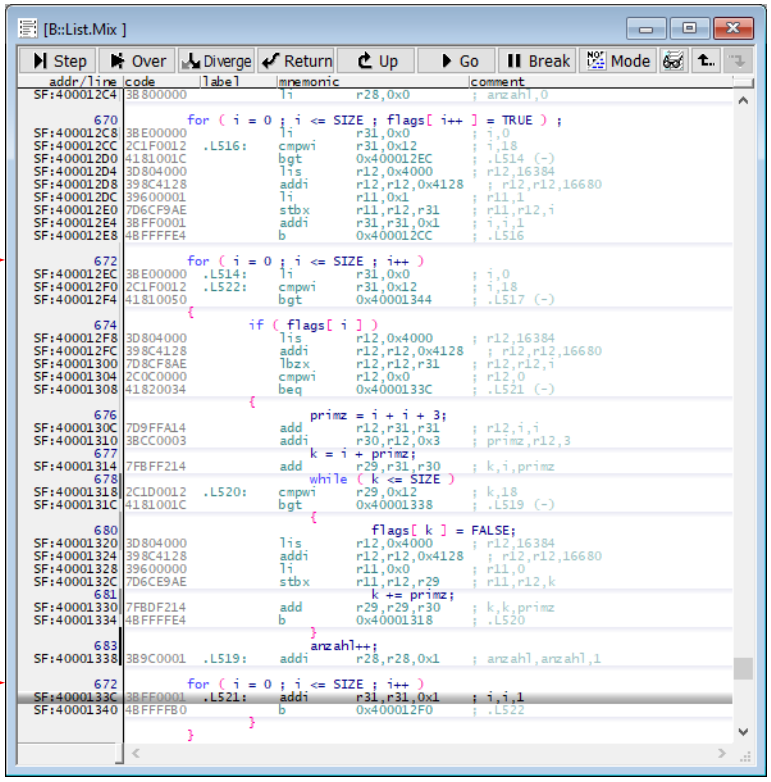
Observabiltiy Gap

This is a term from trace-based code coverage. It is used in conjunction with the code coverage metrics condition, decision and MC/DC, in very rare cases also in conjunction with statement coverage. It means that the result of a decision/condition cannot be monitored. This results is a gap in the code coverage that must be closed to achieve completeness.

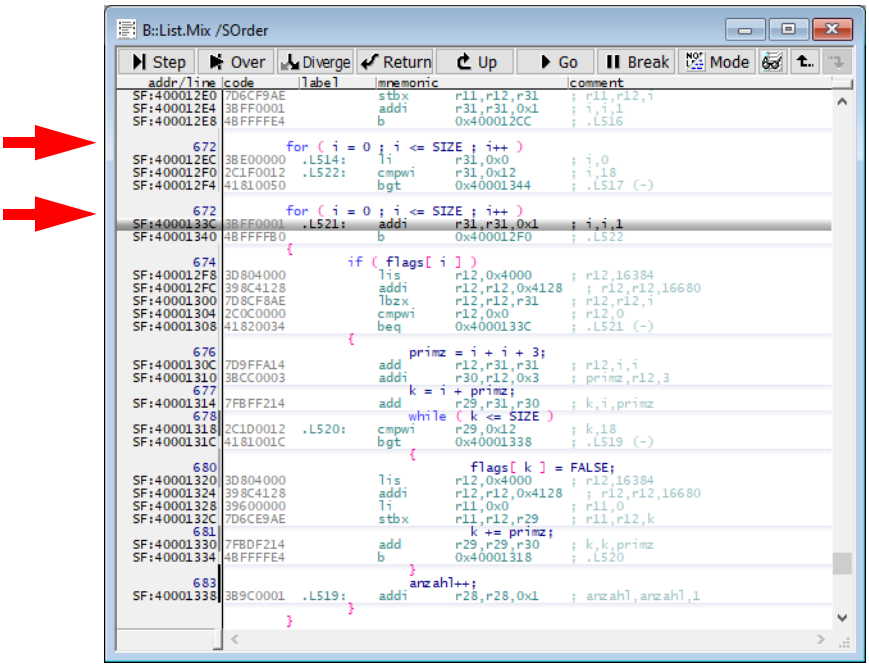
Order of Source Code Lines

TRACE32 distinguishes between Target Order and **Source Order**. The two orders differ, when the compiler changes the execution order of the source code, or when it creates two or more disjointed assembly blocks for one line of source code. This happens especially with inline functions, with for-loops, and when compiler optimizations are enabled.

The example below shows code in **Target Order**. The representation is determined by the memory addresses. The memory addresses are strictly increasing. As you can see, the source code line 672 is implemented in two disjointed assembly blocks. This is because of the special nature of the "for" statement in C, which actually describes three actions (entry action, loop condition, loop action).



The example below shows code in **Source Order** (due to the used option **/SOrder**). The representation is determined by the line numbers of the source code. The line numbers of the source code are monotonically increasing.

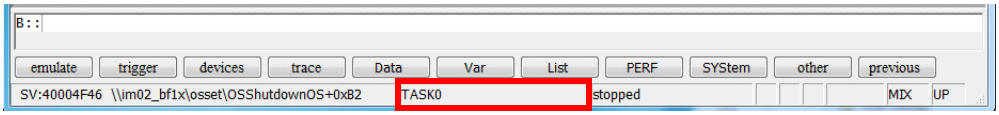


As soon as you switch the display to **Source Order**, the display is no longer determined by the memory addresses of the assembly instructions, but by the line numbers of the source code lines.

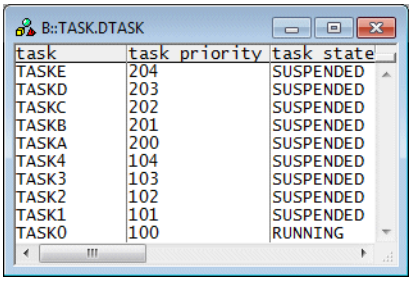
OS-aware Debugging

TRACE32 includes a configurable target-OS debugger that allows to include the task/process context information into the debug process. The main features are:

- The name of the currently running task/process is displayed in the TRACE32 PowerView state line.



- The task/process list can be displayed.



- The context of all tasks can be inspected (register set, stack frame etc.).

For detailed information on all TRACE32 PowerView features provided for your OS, refer to **"OS Awareness Manuals"** (rtos_*<os>*.pdf).

What has to be configured for an OS Awareness debugging depends on the type of the target-OS in use. TRACE32 distinguishes:

- OS (no dynamic memory management).
- AUTOSAR/OSEK operating systems.
- OS+MMU (dynamic memory management).

OS (No Dynamic Memory Management)

Ready-to-use configuration files are provided for most common available OS of this type.

If your kernel is compiled with symbol and debug information, the configuration for your target-OS can be activated as follows:

TASK.CONFIG *<file>*	Configures the target OS debugger using a configuration file provided by Lauterbach

All necessary files can be found in `~~/demo/<architecture>/kernel` where `~~` is expanded to the `<trace32_installation_directory>`, which is `c:/T32` by default.

Example:

```
; load ready-to-use file to configure eCos aware debugging
TASK.CONFIG ~~/demo/arm/kernel/ecos/ecos.t32
```

AUTOSAR/OSEK Operating Systems

The OSEK System Builder can be configured to create an ORTI file. This ORTI file can be loaded to TRACE32 to activate OSEK-aware debugging.

TASK.ORTI *<orti_file>*	Load the ORTI file

Example:

```
; load ORTI file to configure OSEK aware debugging
TASK.ORTI im02_bf1x.ort
```

For details refer to **"OS Awareness Manual OSEK/ORTI"** (rtos_orti.pdf).

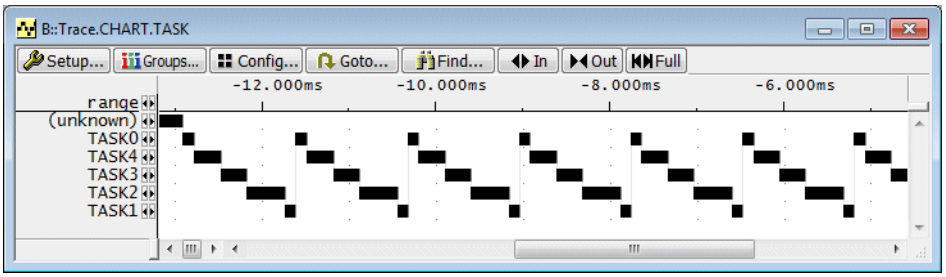
OS+MMU (Dynamic Memory Management)

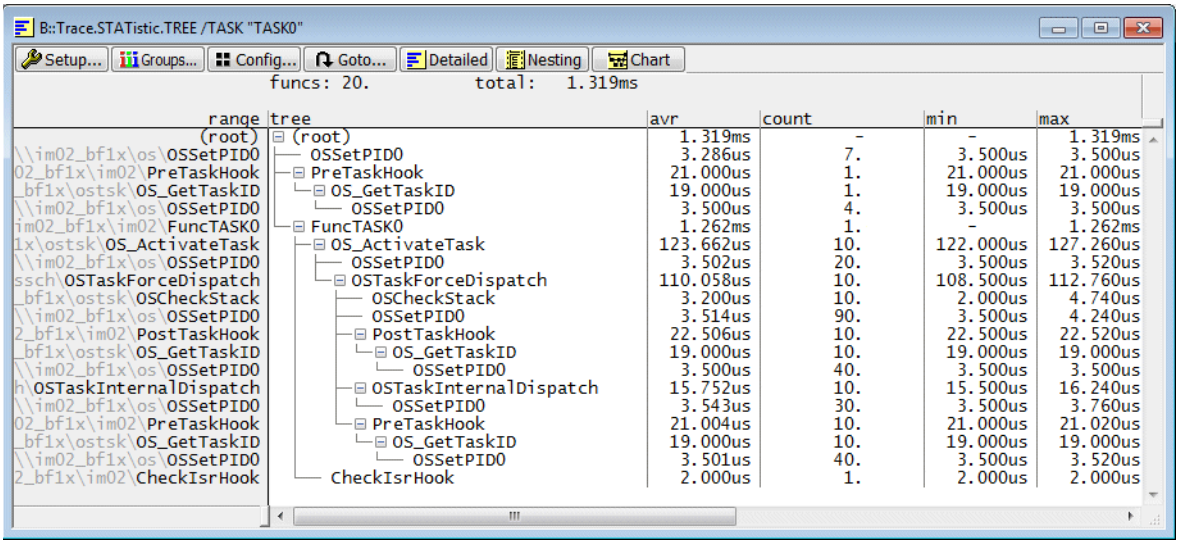
OS that use dynamic memory management require a more complex configuration.

Since Linux is the most popular OS here **"Training Linux Debugging"** (training_rtos_linux.pdf) is provided to show how to activate the Linux awareness. For details on the other operating systems, refer to **"OS Awareness Manuals"** (rtos_*<os>*.pdf).

If an OS is running on your target, **OS-aware debugging** has to be configured in order to use OS-aware tracing.

OS-aware tracing allows e.g. to analyze task/process run-times as well as task/process specific call trees.





OS-aware tracing requires that the on-chip trace generation logic can generated task information.

There a two methods how task switch information can be generated:

- **By generating a trace packet when the OS writes the ID of the current task to variable that contains the information which task is currently running.**

  This method requires that the on-chip trace generation logic can generate Data Address information and Data Value information for write accesses. This method has the advantage that it **does not require any support from the operating system.**

- **By generating task switch packets.**

  This method requires that the processor/core provides a register especially for this purpose. The OS has to write an identifier for the current task to this register on every task switch and the on-chip trace generation logic has to generate a trace packet, whenever a write access to this register occurs. This method needs support from the OS. If the OS does not operate this special purpose register it has to be patched in order to do so.

Task Switch by Tracing Special Write Accesses

Every OS has a variable that contains the information which task/process is currently running. TRACE32 PowerView uses a generic function to identify this variable.

TASK.CONFIG(magic)	Returns the address of the variable that contains the information which task/process is running.
TASK.CONFIG(magic[*<core>*])	Returns the address of the variable that contains the information which task/process is running on the specified core (SMP systems).

If an OS not supported by Lauterbach is used, the so-called “simple” awareness can be used to prepare OS-aware tracing. Details on the “simple” awareness can be found in ~~/demo/kernel/simple/readme.txt.

Depending on the trace analysis to be performed one of the following trace filters has to be set.

Break.Set TASK.CONFIG(magic) /Write /TraceEnable	Advise the on-chip trace generation logic to generate a trace packet whenever a write access to the variable identified by TASK.CONFIG(magic) occurs.
Break.Set TASK.CONFIG(magic) /Write /TraceData	Advise the on-chip trace generation logic to generate trace packets for the instruction execution sequence and for the write accesses to the variable identified by TASK.CONFIG(magic).

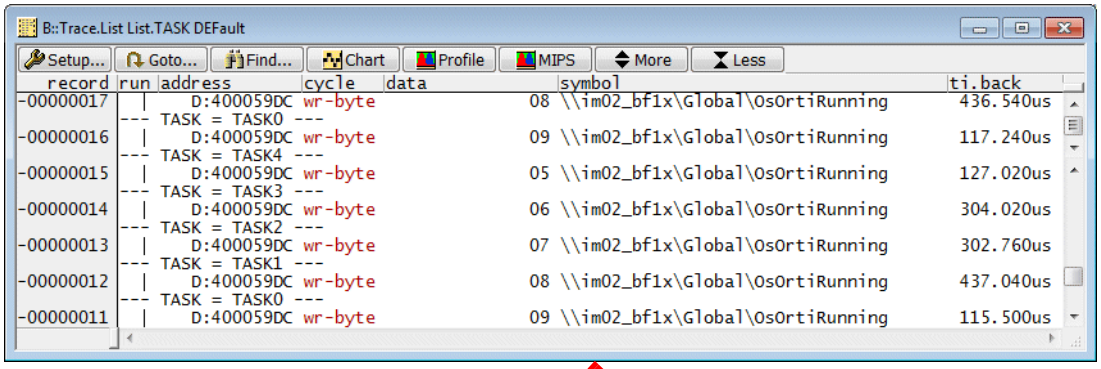
Example 1: A time chart of the tasks/processes running is required. A MPC5646C with a Nexus trace port is used for this example.

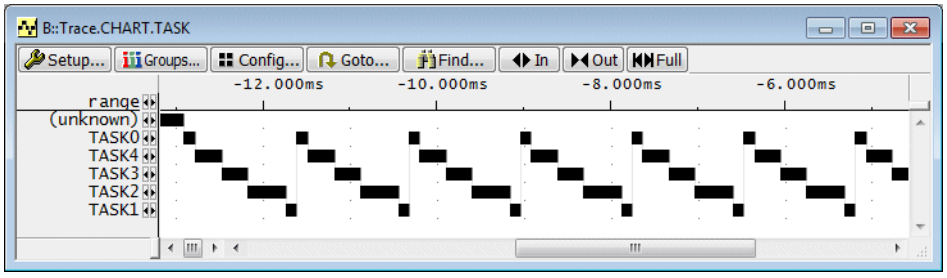
```
Break.Set TASK.CONFIG(magic) /Write /TraceEnable

…

Trace.List List.TASK DEFault

Trace.Chart.TASK
```





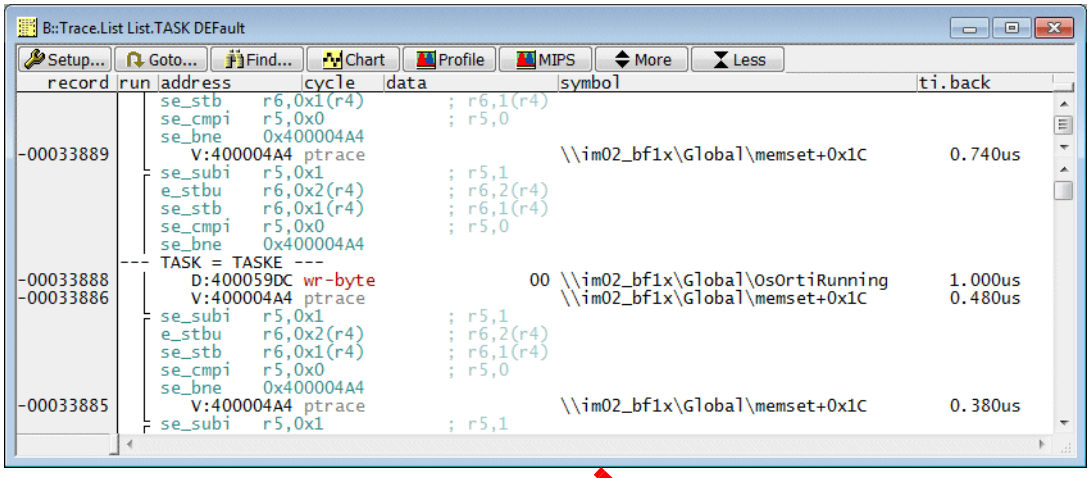
Example 2: A detailed function nesting is required for each task/process. A MPC5646C with a Nexus trace port is used for this example.

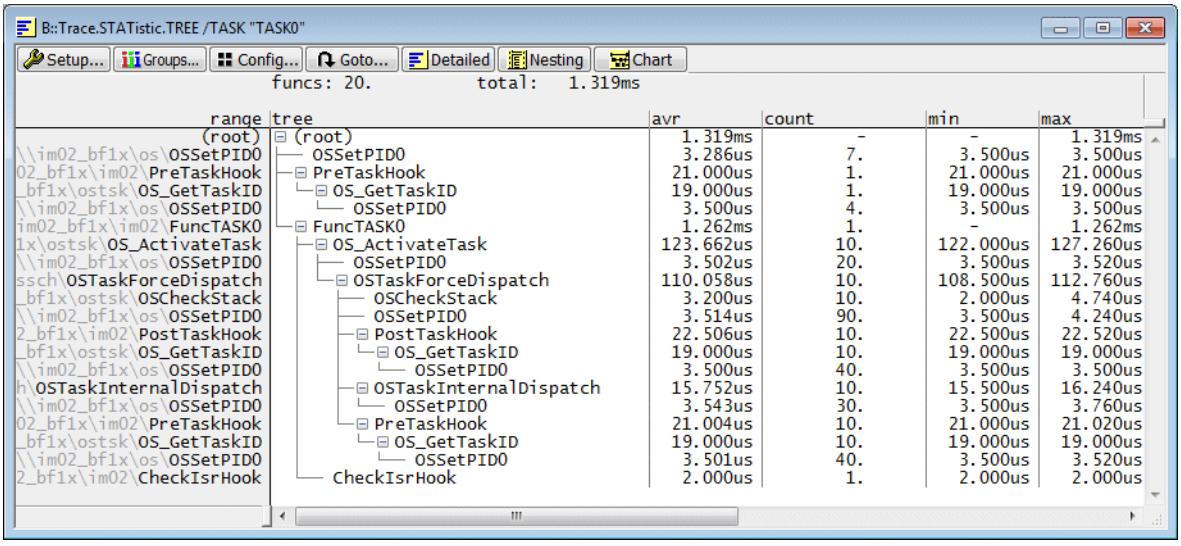
```
Break.Set TASK.CONFIG(magic) /Write /TraceData

…

Trace.List List.TASK DEFault

Trace.STATistic.TREE /TASK "TASK0"
```





This generic description does not cover all details for specific trace protocols. For details refer to **"Training MPC5xxx/SPC5xx Nexus Tracing"** (training_nexus_mpc5500.pdf) or to the **"OS Awareness Manuals"** (rtos_<*os*>.pdf).

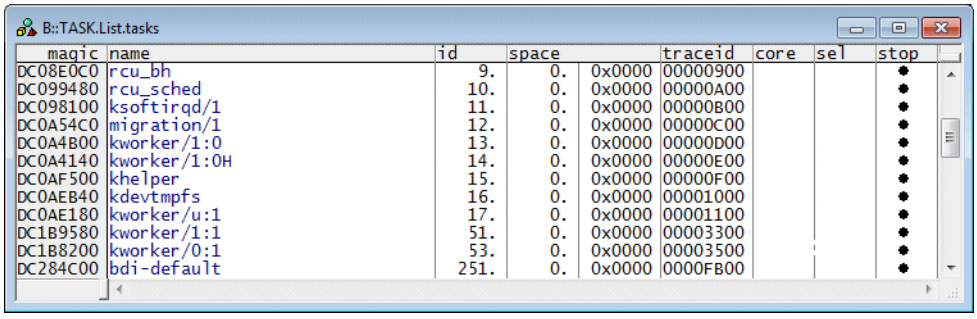
Task Switch by Tracing Task Switch Packets

An alternative way to generate task switch information is required, if the on-chip trace generation logic can not generate Data Address information and Data Value information for write accesses. The common solution here is that the processor/core provides a special register. The OS has to write a task-identification for the current task to this register on every task switch and the on-chip trace generation logic has to generate a trace packet, whenever a write access to this register occurs. Examples for such registers are:

- the Context ID register for ARM/Cortex processors/cores.
- the Nexus PID Register for Freescale Qorivva processors/cores.

If the OS in use operates this special purpose register, it can be used for OS-aware tracing even if the on-chip trace generation logic can generate Data Address information and Data Value information for write accesses.

The task-identification used by the OS (called **traceid** in TRACE32) does not have to be identical to the task ID. The command **TASK.List.tasks** lists the trace IDs assigned to the individual tasks.



magic	name	id	space		traceid	core	sel	stop
DC08E0C0	rcu_bh	9.	0.	0x0000	00000900			●
DC099480	rcu_sched	10.	0.	0x0000	00000A00			●
DC098100	ksoftirqd/1	11.	0.	0x0000	00000B00			●
DC0A54C0	migration/1	12.	0.	0x0000	00000C00			●
DC0A4B00	kworker/1:0	13.	0.	0x0000	00000D00			●
DC0A4140	kworker/1:0H	14.	0.	0x0000	00000E00			●
DC0AF500	khelper	15.	0.	0x0000	00000F00			●
DC0AEB40	kdevtmpfs	16.	0.	0x0000	00001000			●
DC0AE180	kworker/u:1	17.	0.	0x0000	00001100			●
DC1B9580	kworker/1:1	51.	0.	0x0000	00003300			●
DC1B8200	kworker/0:1	53.	0.	0x0000	00003500			●
DC284C00	bdi-default	251.	0.	0x0000	0000FB00			●

Example 1: A time chart of the tasks/processes running is required. A MPC5646C with a Nexus trace port is used for this example.

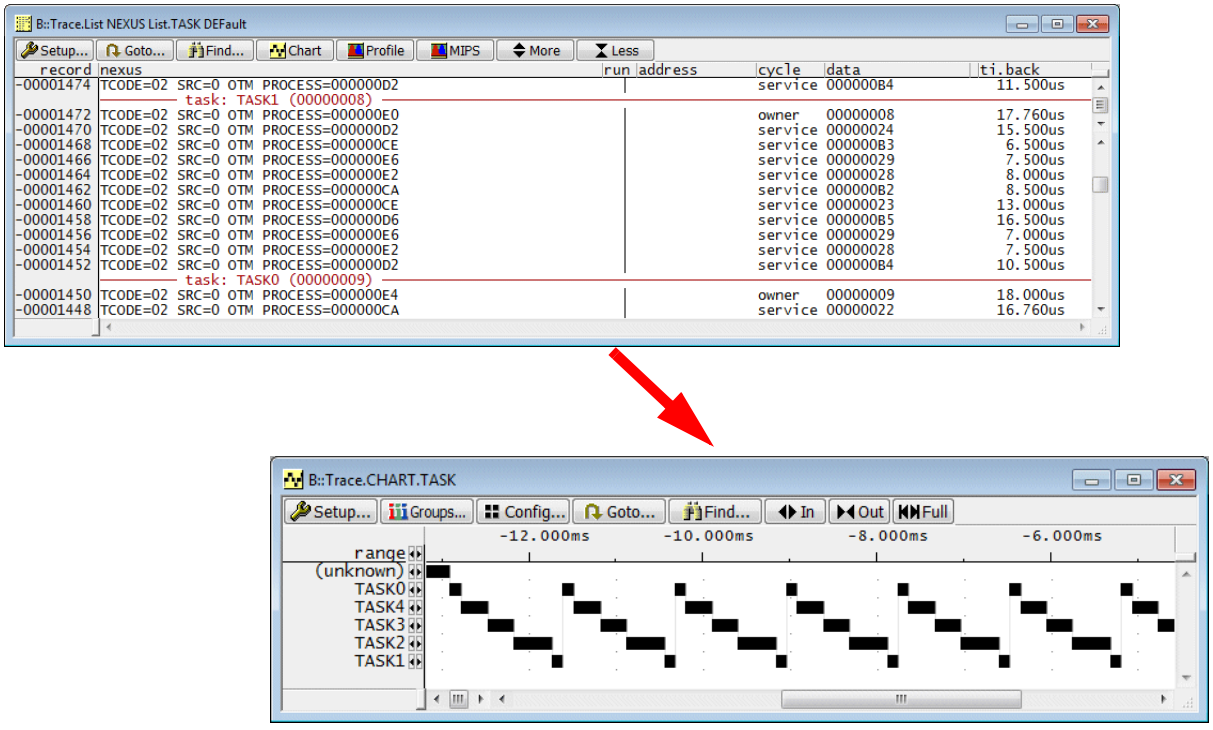
```
; disable the generation of trace packets for the instruction execution
; sequence
NEXUS.BTM OFF

; enable the generation of so-called Ownership Trace Messages
; the Nexus module generates an Ownership Trace Message on every write
; access to the Nexus PID Register
NEXUS.OTM ON

…

Trace.List NEXUS List.TASK DEFault

Trace.Chart.TASK
```



Please be aware that the Nexus PID Register is used here to generate information on task switches (owner) and other OS specific information (services, isr2s).

Example 2: A detailed function nesting is required for each task/process. A MPC5646C with a Nexus trace port is used for this example.

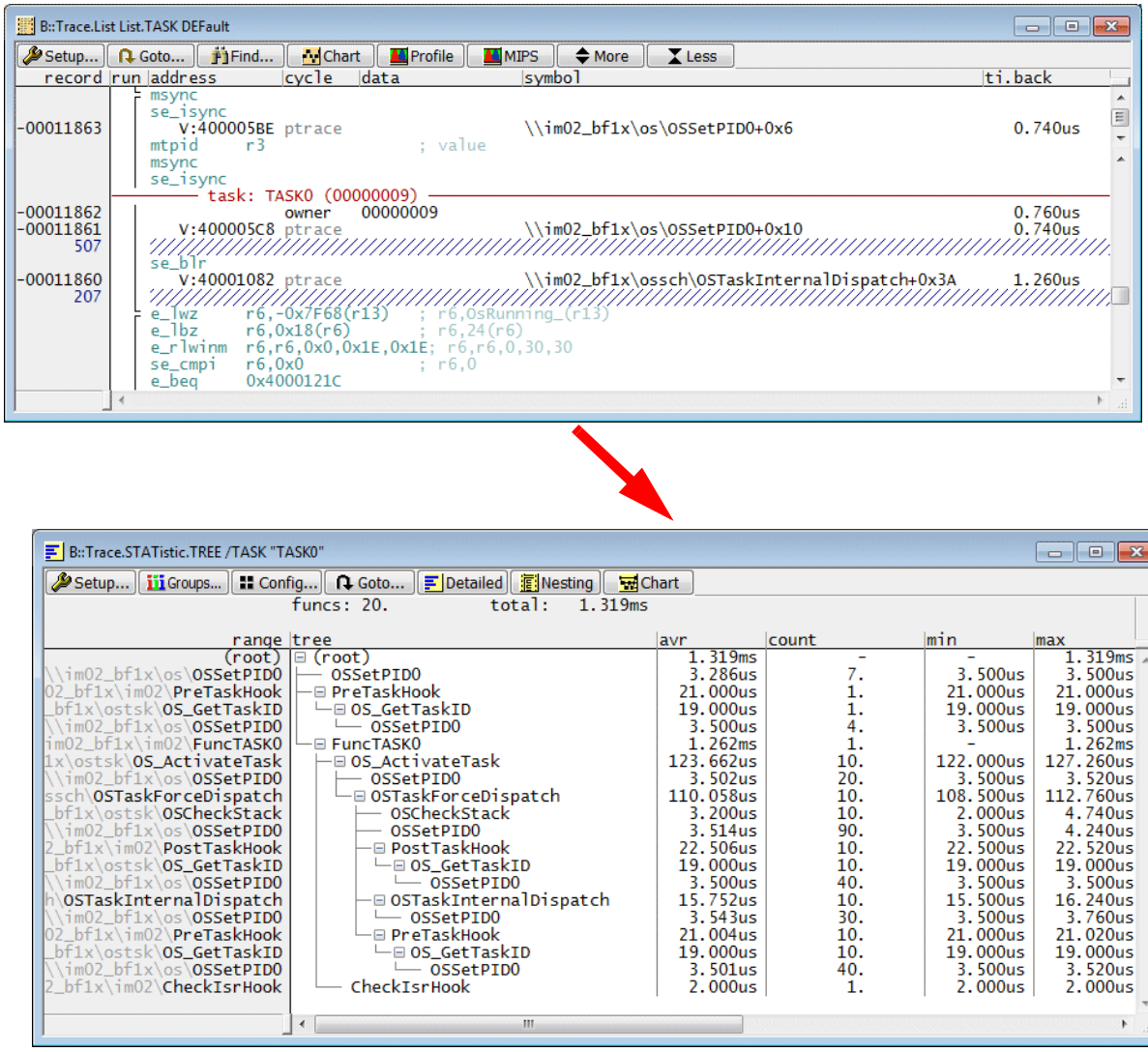
```
; enable the generation of trace packets for the instruction execution
; sequence (default setting)
NEXUS.BTM ON

; enable the generation of so-called Ownership Trace Messages
; the Nexus module generates an Ownership Trace Message on every write
; access to the Nexus PID Register
NEXUS.OTM ON

…

Trace.List List.TASK DEFault

Trace.STATistic.TREE /TASK "TASK0"
```



This generic description does not cover all details for specific trace protocols. For details refer to **"Training MPC5xxx/SPC5xx Nexus Tracing"** (training_nexus_mpc5500.pdf) or to the **"OS Awareness Manuals"** (rtos_*<os>*.pdf).

Process

Subject area: OS Awareness

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

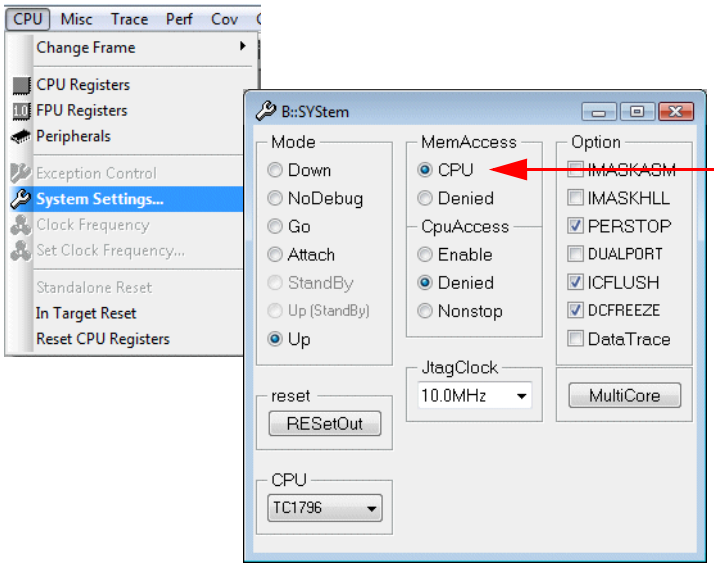
Subject area: OS Awareness

Real Time Operating System - equivalent to kernel.

Technology

Various cores/processors allow a debugger to read/write memory and memory-mapped registers while the core is executing the program. The debugger has in most cases direct access to the system memory bus, so no extra load for the core is generated by this feature.

Open the **SYStem** window in order to check if your processor architecture allows a debugger to read/write memory while the core is executing the program:



MemAccess CPU/NEXUS/DAP indicates, that the core allows the debugger to read/write memory while the core is executing the program

Please be aware that caches, MMUs, tightly-coupled memories and suchlike add conditions to the run-time memory access or at worst make its use impossible.

The following description is only a rough overview on the restrictions. Details about your core can be found in the **Processor Architecture Manual**.

Cache

If run-time memory access for a cached memory location is enabled the debugger acts as follows:

- **Program execution is stopped**

  The data is read via the cache respectively written via the cache.

- **Program execution is running**

  Since the debugger has no access to the caches while the program execution is running, the data is read from physical memory. The physical memory contains the current data only if the cache is configured as write-through for the accessed memory location, otherwise out-dated data is read.

  Since the debugger has no access to the cache while the program execution is running, the data is written to the physical memory. The new data has only an effect on the current program execution if the debugger can invalidate the cache entry for the accessed memory location. This useful feature is not available for most cores.

MMU

Debuggers have no access to the TLBs while the program execution is running. As a consequence run-time memory access cannot be used, especially if the TLBs are dynamically changed by the program.

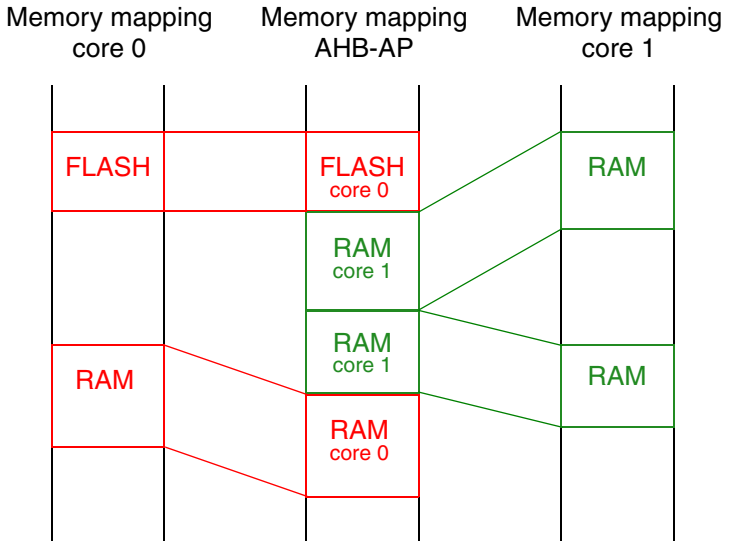
In the exceptional case of static TLBs, the TLBs can be scanned into the debugger. This scanned copy of the TLBs can be used for the address translation while the program execution is running.

Tightly-Coupled Memory

Tightly-coupled might not be accessible via the system memory bus.

The command **SYStem.MemAccess DAP** enables the run-time memory access for the ARM/Cortex architecture. The run-time memory access is done via the bus that is configured with the command **SYStem.CONFIG.MEMORYACCESSPORT** *<port>*. In most cases the AHB bus is used.

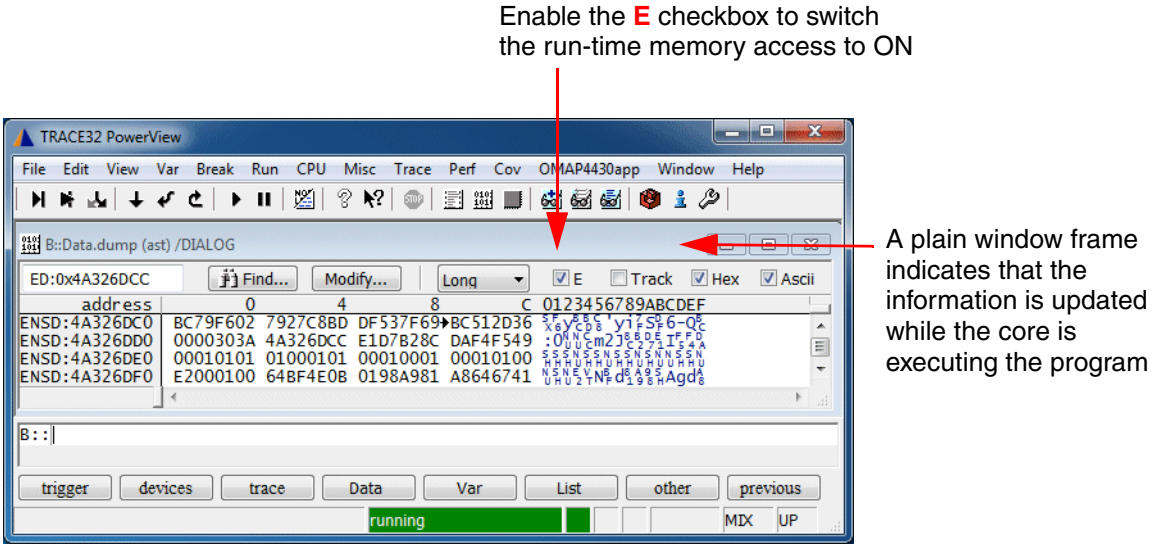
The memory mapping of the cores might be different from the memory mapping of the AHB bus e.g. in an AMP system.



The command **SYStem.Option.DAPREMAP** *<address_range> <address>* allows to inform the debugger about the AHB memory mapping.

Run-time Access to Memory

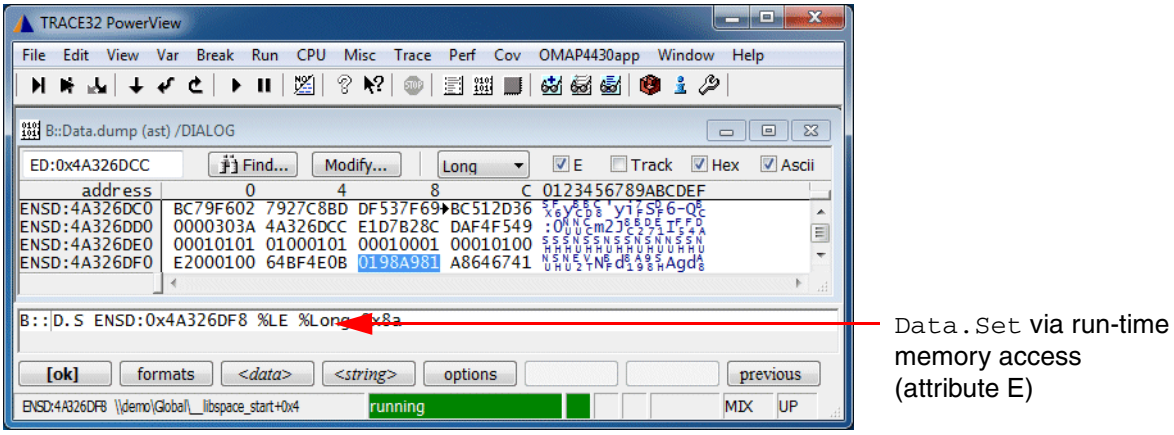
Configure the run-time memory access for a specific memory area.



If the **E** checkbox is enabled, the attribute E is added to the **access class**:

EP:1000	**Program** address 0x1000 with run-time memory access
ED:6814	**Data** address 0x6814 with run-time memory access

Write accesses to the memory work correspondingly:



Command line examples:

```
SYStem.MemAccess Enable                 ; Enable the non-intrusive
                                        ; run-time memory access

…

Go                                      ; Start program execution

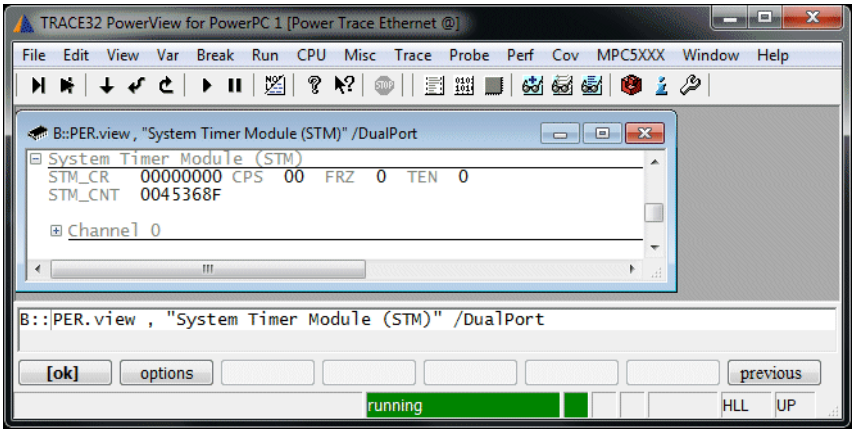
Data.dump E:0x6814 /DIALOG              ; Display a hex dump starting at
                                        ; address 0x6814 via run-time
                                        ; memory access

Data.Set E:0x6814 0xAA                  ; Write 0xAA to the address
                                        ; 0x6814 via run-time memory
                                        ; access

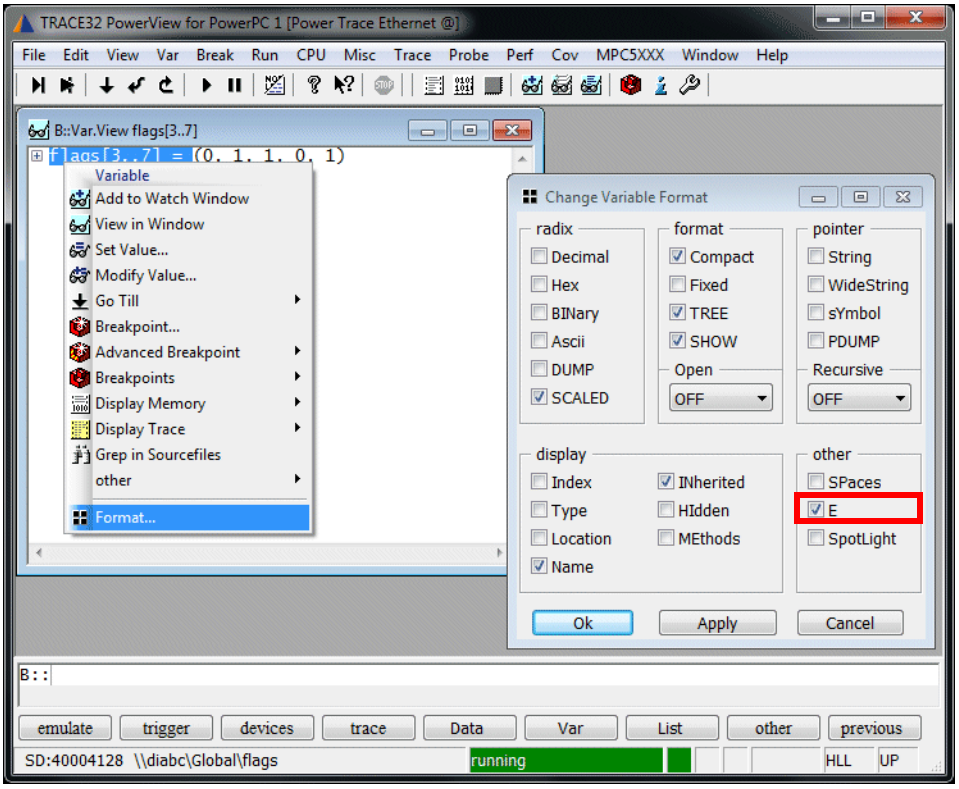
List E:                                 ; Display a source listing via
                                        ; run-time memory access
```

Run-time Access to Memory-mapped Registers

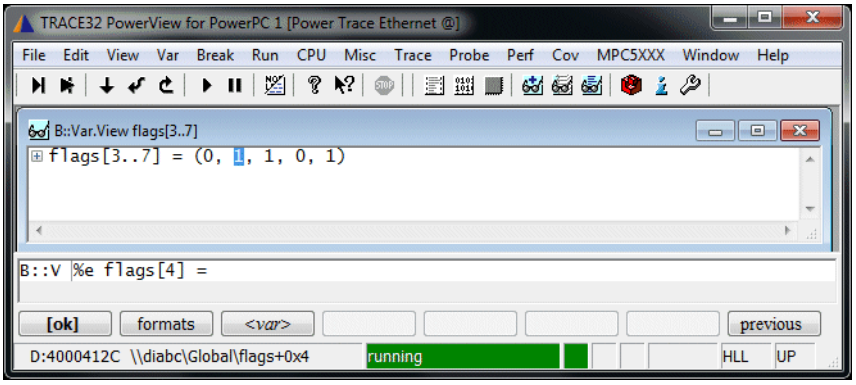
Use the option **/DualPort** when you use the **PER** command to display the SFRs.



Configure the run-time access for the selected variable by enabling the **E** checkbox in the **Change Variable Format** dialog.



If you want to change the contents of a variable while the program execution is running, double click to the variable.



The following command is displayed in the TRACE32 PowerView command line:

```
Var.set %E flags[4] =
```

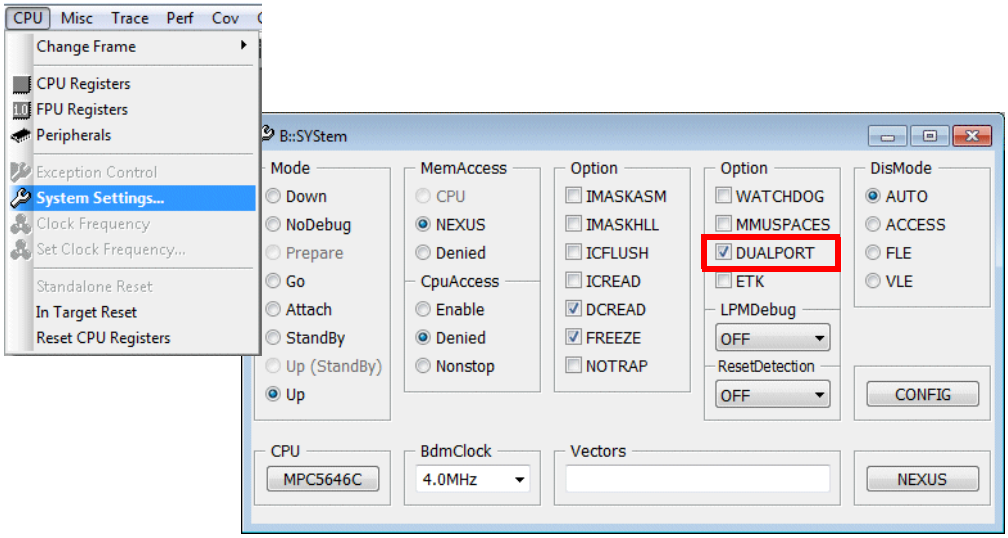
The format parameter **%E** advises the debugger to write the new variable value via the run-time memory access.

Command line examples:

```
Var.View %E flags                       ; display variable flags via
                                        ; run-time memory access

Var.set %E flags[3]=22                  ; write 22 to variable flags[3]
                                        ; via run-time memory access
```

Most cores/processors that allow a run-time memory access provide the checkbox DUALPORT in the SYStem window. When DUALPORT is enabled run-time, access is automatically enabled for all windows that display memory (e.g. source listing, memory dumps, variable displays, displays of SFR).



Command line example:

```
SYStem.Option.DUALPORT ON
```

Sample-based Profiling

Sample-based profiling collects periodically the actual program counter or the actual contents of a memory location in order to calculate:

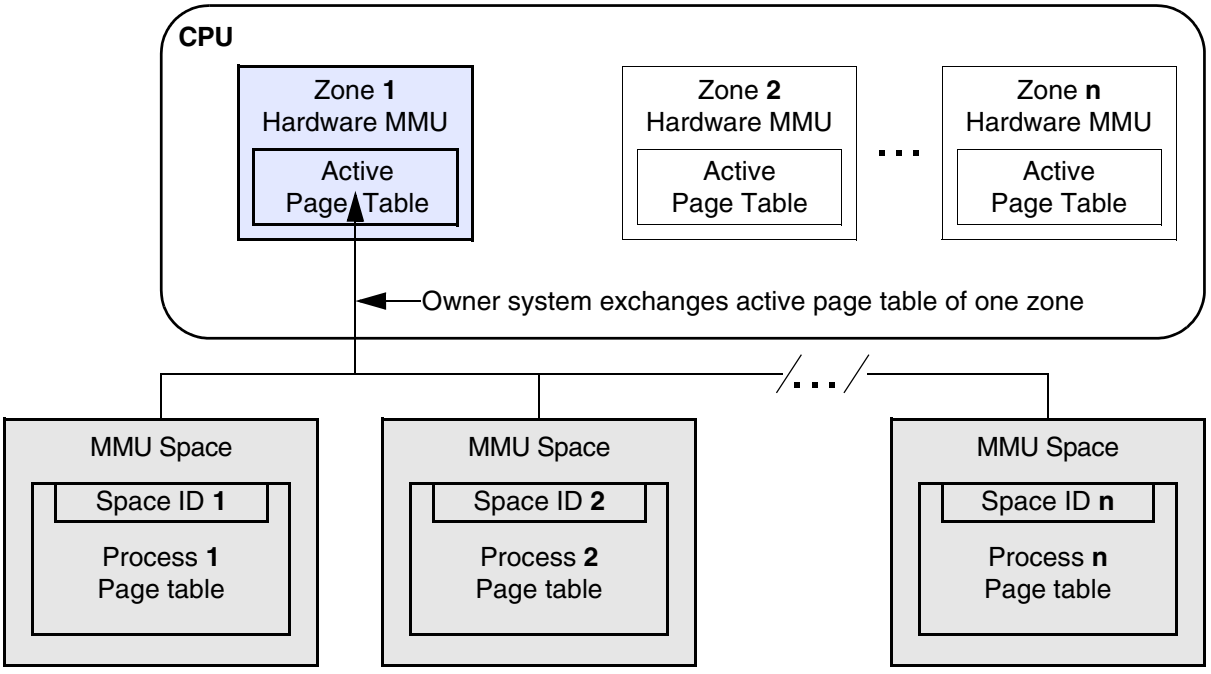
- The percentage of run-time used by a high-level language function
- The percentage of run-time a variable had a certain contents.
- The percentage of run-time used by a task.

For details refer to the **PERF** command group.

OS Space ID

Subject area: Address translation, OS Awareness

A space ID is a 16-bit memory space identifier which extends a logical TRACE32 address. With space IDs, TRACE32 can handle multiple address spaces (= MMU spaces) in the debugger address translation.



Space IDs are defined within a loaded TRACE32 OS Awareness. Often, space IDs are directly derived from the OS process ID. Be aware that this depends on the OS and the loaded OS Awareness.

- The space ID **0xFFFF** indicates an error. The OS Awareness loaded into TRACE32 is not able to determine the process to which the address belongs.
- A space ID **0x0000** refers to addresses belonging to the kernel address range, such as the kernel itself or common libraries or common modules.

NOTE:	With **TRANSlation.CONFIG.ZoneSPACES** enabled and **TASK.ACCESS** set to a specific zone, addresses belonging to another zone also show **0x0000** as space ID.

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

If enabled, the TRACE32 address format is extended by the space ID in the leading position, directly before the address offset. The space ID is followed by a single or double colon to separate the space ID from the address offset. The format of a TRACE32 address containing a space ID looks like this:

```
<access_class>:<space_id>:<address_offset>
```

Examples:

- `G:0x020A:0x80000000`
- `G:0x0:0x4000C000`
- `0x170:0x1F000000`

```
TRANSlation.CONFIG.MMUSPACES ON
; load the debug symbols for the process hello
; the space ID is a 16-bit number preceding the virtual address

Data.LOAD.Elf hello N:0x0229:0x0 /NoCODE /NoClear
```

Refer to the manual for your target operating system for details, e.g. **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf).

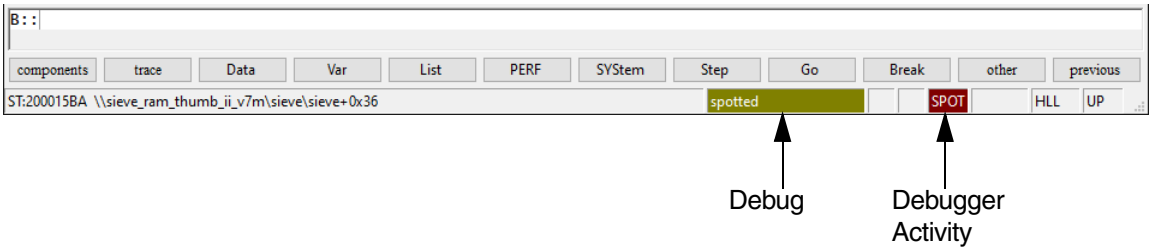
Spot Breakpoint

In case of a breakpoint hit, the program execution is stopped briefly:

- to refresh the screen.
- to perform actions that are configured in advance.

The stop usually takes about 100 ms, depending on the speed of the debug interface, the information displayed in TRACE32 PowerView and the actions to be performed.

```
Var.Break.Set myspot /Write /Spot
```



The debug field in the TRACE32 state line shows spotted, the olive color indicates that the program execution is mostly stopped and runs only very short. The Debug Activity field shows SPOT in red.

The following actions can be performed at a Spot breakpoint:

- Logging variables into an AREA window (**Var.LOG** command).
- Logging the current program runtime to the RunTime log (**RunTime.List** command).
- Exchange data between the target application and the debugger using a terminal window (**TERM.METHOD** command).
- Exchange data between the target application and an application running on the host computer (**FDX.METHOD** command).

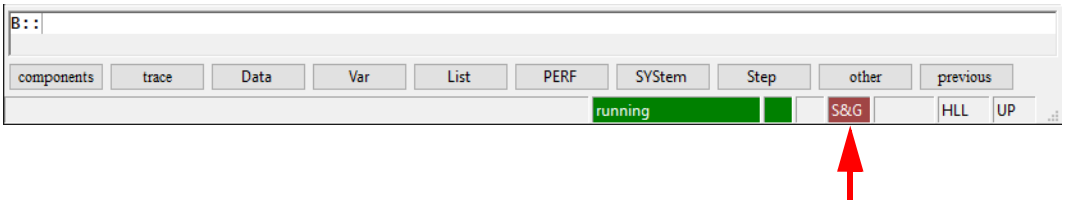
StopAndGo Mode

If a TRACE32 Debugger is used the program execution on the processor/core is real-time by default. If your program does not run under hard real-time condition you can trade in real-time for more debug features.

Example 1: The user wants to monitor a global variable while the program execution is running, but the debugger can not read physical memory while the program execution is running. TRACE32 can now be configured (**SYS.MemAccess StopAndGo**) to stop the program execution 10 times per second shortly in order to read the variable of interest and restart it as quickly as possible. The debugger is now working in **StopAndGo** mode.

Example 2: The user want to find out which function of his program is the most time-consuming (**performance analysis**), but the debugger can not read the current program counter while the program execution is running. The debugger automatically uses **StopAndGo** mode here. In this operation mode the debugger stops the program execution periodically to read the program counter and restarts it as quickly as possible.

Whenever the debugger uses **StopAndGo** mode to provide helpful debug features this is indicated by a red S&G in the Debugger Activity field of the TRACE32 PowerView state line.



The time taken by a short stop depends on various factors:

- The time required by the debugger to start and stop the program execution on a processor/core (main factor).
- The number of cores that need to be stopped and restarted.
- Cache and MMU assesses that need to be performed to read the information of interest.
- The type of information that is read during the short stop.

  Reading the program counter is optimized and can usually be done quickly.

Symmetrical Multi-Processing (SMP)

SMP = Symmetrical MultiProcessing

A multicore-chip that contains only cores of the same type can be configured as an SMP system. In an SMP system the task are assigned by an SMP operating system dynamically to the cores.

For debugging SMP systems, only one TRACE32 instance is opened and all cores are controlled from this one point.

Task

Subject area: OS Awareness

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

Subject area: OS Awareness

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

TRACE32 Virtual Memory

The TRACE32 Virtual Memory is memory on the host computer which can be displayed and modified with the same commands as a real target memory (**Data** command group). The memory class **VM:** provides access to this memory.

The following examples show some use cases for the TRACE32 Virtual Memory:

Example 1

A part of the target memory contents is copied to the virtual memory to allow you to check all changes performed by the program execution.

```
; copy contents of specified address range to TRACE32 Virtual Memory
Data.Copy 0x3fa000++0xfff VM:

; display contents of TRACE32 Virtual Memory at specified address
Data.dump VM:0x3fa000

Go

Break

; compare contents of target memory with contents of TRACE32 Virtual
; Memory for specified address range
Data.Compare 0x3fa000++0xfff VM:0x3fa000

; search for next difference
Data.Compare

…
```

An advanced example can be found in:
`~~/demo/powerpc/hardware/qoriq_p1_p2/all_boards/program_bootsequencer.cmm`

Example 2

Loading the code to the target memory is not working, you can inspect the code by loading it to the virtual memory.

```
Data.LOAD.Elf demo.elf /VM            ; load program code to TRACE32
                                      ; Virtual Memory

Data.List VM:                         ; display a source listing based on
                                      ; the code in the TRACE32 Virtual
                                      ; Memory

sYmbol.List.MAP                       ; display the addresses to which
                                      ; the code/data was written
```

Example 3

TRACE32 does not support the ELF file format for NAND FLASH programming.

```
…                                          ; configure TRACE32 NAND FLASH
                                           ; programming

Data.LOAD.Elf demo.elf /VM                 ; load program to TRACE32
                                           ; Virtual Memory

sYmbol.List.MAP                            ; check the addresses to
                                           ; which the code/data was
                                           ; written

FLASHFILE.COPY VM:0x180000++0x3ffff 0x0    ; copy the program from
                                           ; TRACE32 Virtual Memory to
                                           ; NAND FLASH
```

Example 4

TRACE32 needs to read the source code from the target memory in order to decompress the exported trace information. If target memory can not be read while the program execution is running the object code can be provided via the TRACE32 Virtual Memory.

The object code is still read from the target memory when the program execution is stopped.

Please keep the object code in the TRACE32 Virtual Memory up-to-date, out-of-date source code versions will cause errors.

```
Data.LOAD.Elf demo.elf /PlusVM         ; load program code to target and
                                       ; to TRACE32 Virtual Memory

…

Trace.Mode Stack                       ; advise TRACE32 to stop trace
                                       ; recording as soon a trace memory
                                       ; is filled

                                       ; program execution continues

Go

Trace.List                             ; since program execution is still
                                       ; running the source code
                                       ; information required for the
                                       ; trace decompression can not be
                                       ; read from the target memory

                                       ; source code is read from the
                                       ; TRACE32 Virtual Memory instead
```

Example 5

TRACE32 needs to read the source code from the target memory in order to decompress the exported trace information. If the JTAG interface is very slow, reading target memory is slow. As a result the trace evaluation is slow.

The trace evaluation can be speeded-up by providing the source code via the TRACE32 Virtual Memory. Please keep the source code in the TRACE32 Virtual Memory up-to-date, out-of-date source code versions will cause errors.

```
Data.LOAD.Elf demo.elf /PlusVM        ; load program code to target and
                                      ; to TRACE32 Virtual Memory

Trace.ACCESS VM:                      ; advise TRACE32 to read source
                                      ; code required for the trace
                                      ; decompression always from TRACE32
                                      ; Virtual Memory because reading
                                      ; the code from the target memory
                                      ; is very slow

…

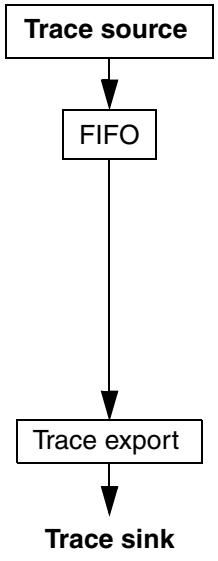
Trace.List
```

TARGET FIFO OVERFLOW

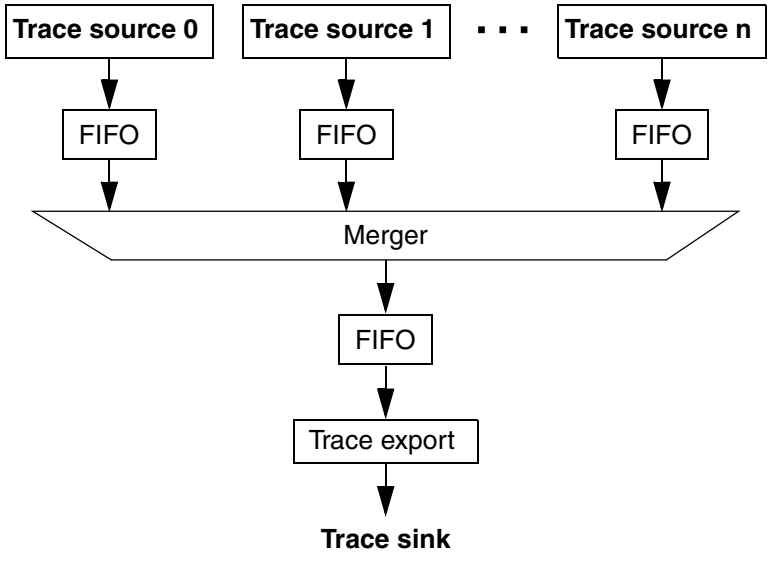
The trace information generated by trace sources has to be conveyed to the trace sink. Typical examples for a trace source are core traces, System Trace Macrocells and bus traces. Typical examples for a trace sink are off-chip trace ports and on-chip trace RAMs.

In the case of a single trace source, a trace export logic is responsible to convey the trace information to the trace sink. If two or more trace sources generate trace information, this information is usually first merged to a single trace data stream.

FIFOs are used to pass trace information between the individual components of the trace infrastructure. If the generating components queue more trace data into the FIFO then the receiving component can process, a TARGET FIFO OVERFLOW is indicated.

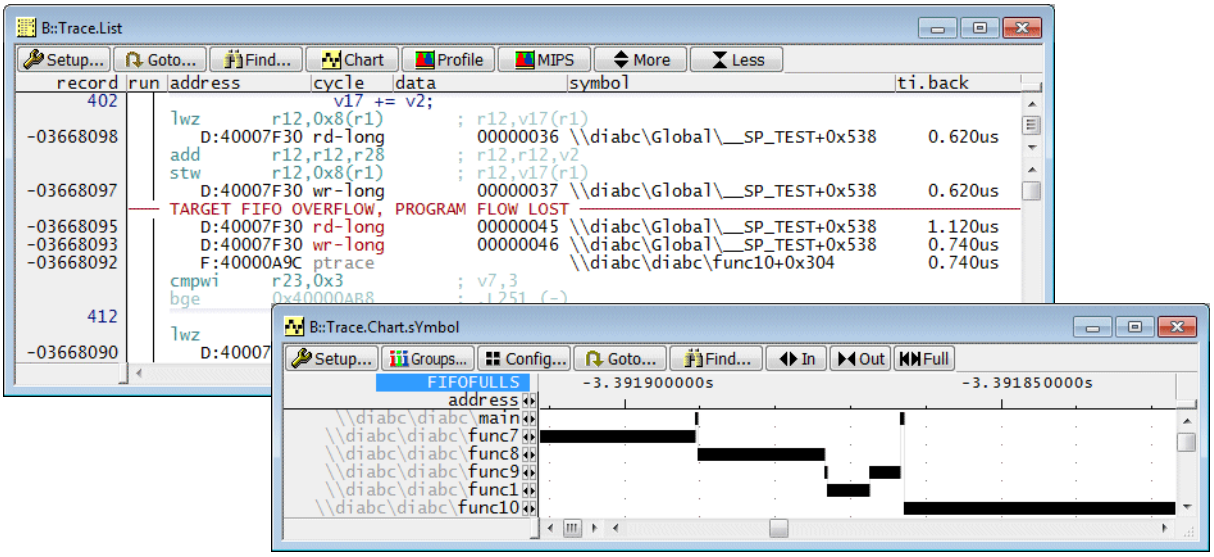


Single trace source



Two or more trace sources

A TARGET FIFO OVERFLOW always results in the loss of trace data. A synchronization packet (including the full program counter and if required the task/process information) will signal that FIFOs are emptied and a lossless transport for sources to sink is guaranteed.



A TARGET FIFO OVERFLOW is strictly spoken not an error, but normal behavior. TRACE32 indicates TARGET FIFO OVERFLOWS/FIFOFULL for two reasons:

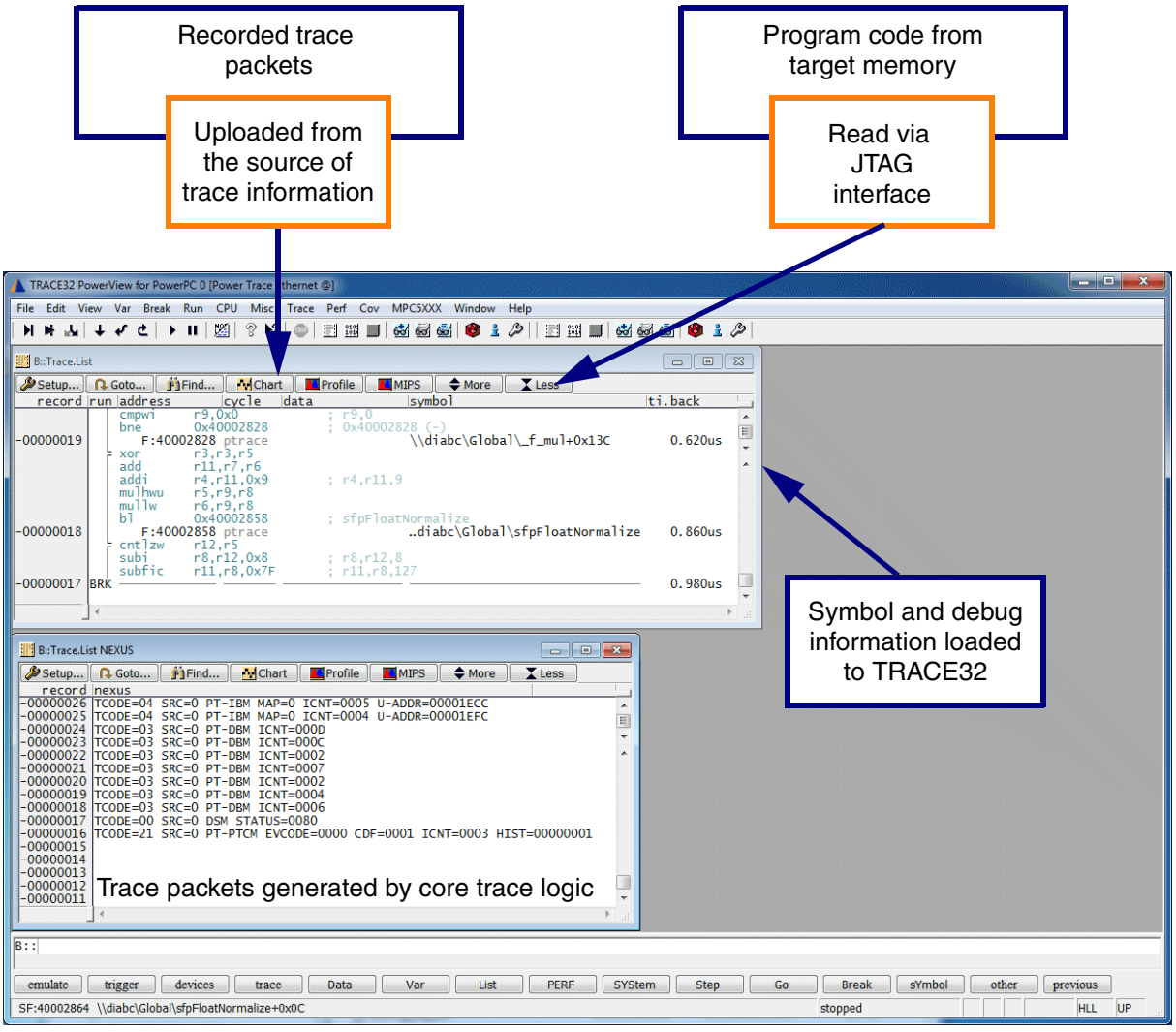
- to tag trace data losses in the trace recording.
- to make the user aware of TARGET FIFO OVERFLOWs because a number of trace analyzes only make sense if the trace recording is lossless.

FLOWERROR

The core trace generation logic on the processor/chip generates trace packets to indicate the instruction execution sequence (program flow).

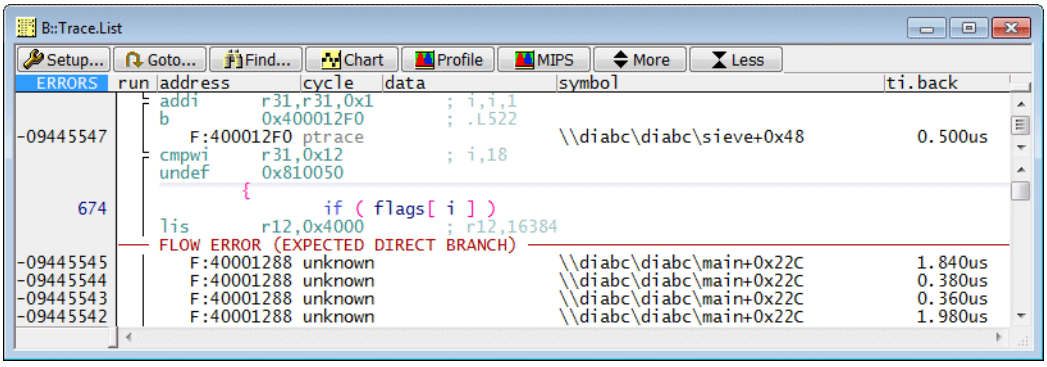
TRACE32 merges the following sources of information in order to provide an intuitive trace display.

- The trace packets recorded.
- The program code from the target memory (read via the JTAG interface).
- The symbol and debug information already loaded to TRACE32.



A FLOW ERROR is indicated (list not complete):

- if TRACE32 detects an invalid trace packet.
- if TRACE32 can not decode a trace packet.
- if the content of a trace packet is not consistent with the program code read from the target memory.



If the trace contains FLOW ERRORs, please try to set up a proper trace recording before you start to evaluate or analyze the trace contents.

Trace Sources

Trace information can be generated by different trace sources:

- **Core trace**

  A core trace provides detailed visibility of the program execution on a core. Trace data are generated for the instruction execution sequence and the task/process switches. Some core traces generate also trace data for the load/store operations. Classic core traces are ARM ETM/PTM or Intel® PT.

- **Function trace**

  In contrast to a core trace, a function trace provides only information on the function entries/exits and if necessary on the task switches. Examples for a function trace are the MCDS Compact Function Trace mode (call/return detection) from Infineon or the SFT Trace (instrumentation based) for the RH850 family.

- **System trace**

  A system trace provides visibility of various events/states inside an SoC. Trace data can be generated by instrumented application code and/or by hardware modules within the SoC.

- **Bus trace**

  Bus traces provide disability on bus transfers. SoC internal busses require a special trace logic on the SoC, while external busses can be traced with a TRACE32 logic analyzer.

  A classical SoC internal bus traces are ARM HTM or Infineon MCDS SPB trace.

Tool Timestamp

All TRACE32 tools that record trace information have a timestamp counter.

Recorded trace information is timestamped with this **tool timestamp** when it is entered into the trace buffer.

The resolution of the timestamp counter is tool dependent. Here examples for the most popular trace tools:

- POWER TRACE II / POWER TRACE III: 5 ns
- POWER TRACE SERIAL: 5 ns
- POWER TRACE / ETHERNET: 20 ns
- CombiProbe: 20ns
- POWER PROBE / LOGIC ANALYZER: 10ns
- POWER INTEGRATOR: at least 4ns, mode dependent higher

Several TRACE32 tools can be chained via the PODBUS/PODBUS EXPRESS connector. The timestamps of chained TRACE32 tools are started synchronously and then run synchronously. The synchronization is established when the first debugger in the chain establishes its communication with its processor/chip via **SYStem.Up** or **SYStem.Attach**.

The synchronized timestamps allow to establish a time reference between trace information recorded by different tools.

VCPU

Subject area: Hypervisor Awareness

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

-