LAUTERBACH
DEVELOPMENT TOOLS

Simulator for NIOS-II

MANUAL

Simulator for NIOS-II

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ..... 🗁

TRACE32 Instruction Set Simulators ..... 🗁

Simulator for NIOS-II 1

History 4

Introduction 4

TRACE32 Simulator License 5

Start the Prepared Demo 6

Quick Start 8

General Restrictions 10

CPU specific SYStem Commands 11

SYStem.CPU — Select CPU type — 11
SYStem.LOCK — Lock and tristate the debug port — 11
SYStem.MemAccess — Select run-time memory access method — 11
SYStem.Mode — Establish the communication with the simulator — 12
SYStem.CONFIG — Configure debugger according to target topology — 12
SYStem.Option.DCFLUSH — Flush data cache before Step or Go — 13
SYStem.Option.Endianness — Select endianness of core — 13
SYStem.Option.EXCADDR — Define exception address — 13
SYStem.Option.FEXCADDR — Define fast TLB miss exception address — 13
SYStem.Option.FPH — Enable the simulation of floating point instructions — 14
SYStem.Option.ICFLUSH — Invalidate instruction cache before go/step — 14
SYStem.Option.IMASKASM — Mask interrupts during assembler step — 14
SYStem.Option.IMASKHLL — Mask interrupts during HLL step — 15
SYStem.Option.IVRCode — Define code for interrupt vector instruction — 15
SYStem.Option.MMUSPACES — Separate address spaces by space IDs — 15
SYStem.Option.MULDIV — Define if mul and div instructions are supported — 17
SYStem.Option.SIMMMU — Define properties of simulated MMU — 17

TrOnchip Commands 18

TrOnchip.state — Display on-chip trigger window — 18
TrOnchip.RESet — Set on-chip trigger to default state — 18

CPU specific MMU Commands 19

MMU.DUMP — Page wise display of MMU translation table — 19

MMU.List Compact display of MMU translation table 21
MMU.SCAN Load MMU table from CPU 22

Memory Classes **24**
Overview 24

Peripheral Simulation **25**

FAQ **25**

Simulator for NIOS-II

Version 06-Jun-2024

History

20-Jul-22 For the MMU.SCAN ALL command, CLEAR is now possible as an optional second parameter.

Introduction

The Simulator is implemented as an Instruction Set Simulator. The full address range (32-Bit) is supported, all peripherals are memory mapped. Peripherals themselves are not simulated. The simulated core is the so-called “economy-core” (without cache).

The machine code is executed plainly, not cycle accurate. The following features are available:

- Disassembler
- File load
- Single Step
- Breakpoints
- Watch registers, variables and structures
- HLL debugging in C and C++
- Sampling of the program and data flow to the trace
- Runtime- and performance analysis
- CTS

The following file formats are supported:

- ELF/DWARF

TRACE32 Simulator License

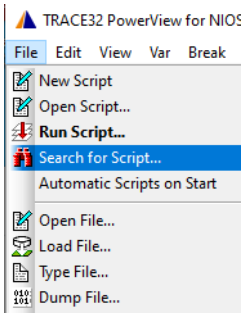
[build 68859 - DVD 02/2016]

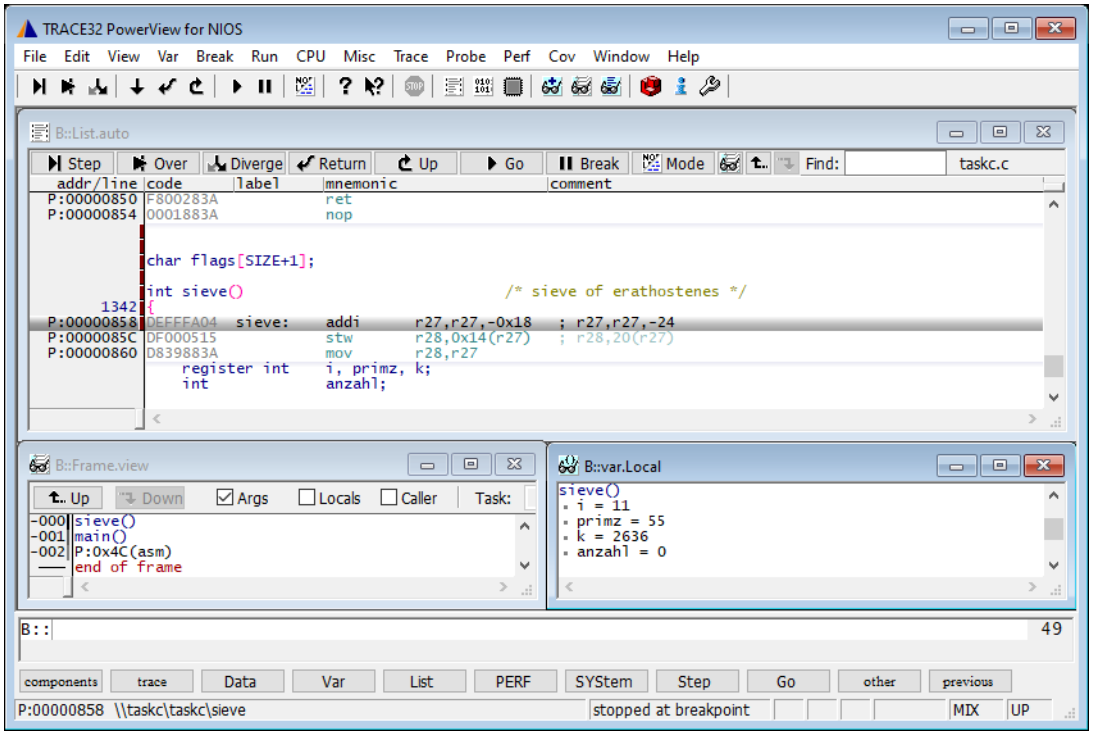
The extensive use of the TRACE32 Instruction Set Simulator requires a *TRACE32 Simulator License*.

For more information, see **www.lauterbach.com/sim_license.html**.

Start the Prepared Demo

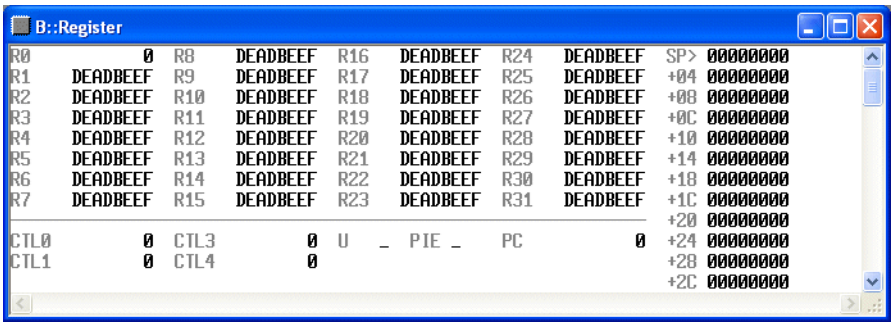
At startup of the TRACE32 NIOS Simulator, a demo file is loaded and started automatically. The program execution is stopped at a breakpoint. The grey cursor line shows the current point of program execution. If the demo does not load and start, please execute the PRACTICE script **demo.cmm**.





Display the Registers

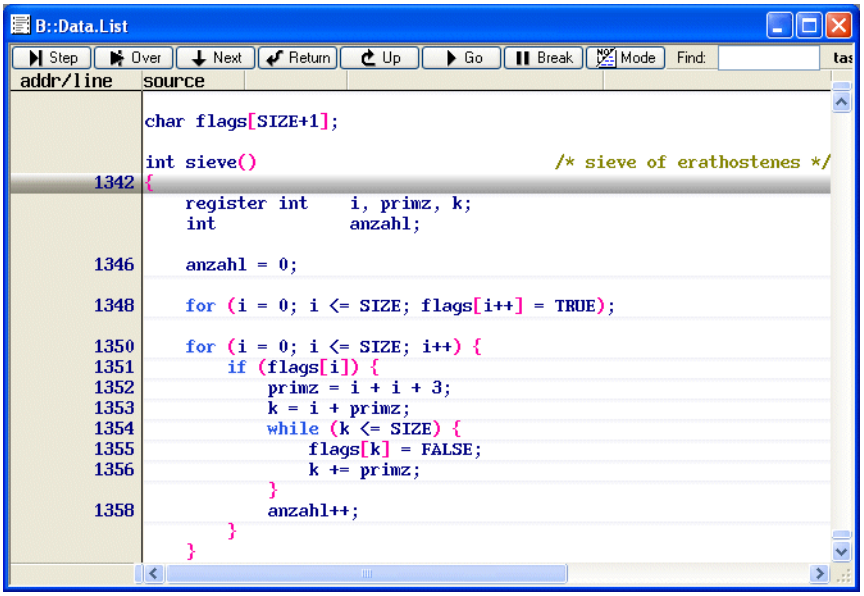
```
Register.view                                   ; displays the cpu registers
```



Display Source Code

```
Data.List                                       ; displays current source code line

Mode.Hll                                        ; select HLL mode for display
```



The way to load a file into the simulator manually is described in the chapter **Quick Start**. All general commands are described in the **"PowerView Command Reference"** (ide_ref.pdf) and **"General Commands Reference"**.

Quick Start

This section describes how:

- to prepare the simulator for debugging
- to load your own application

Starting up the simulator is done as follows:

1. Select the device prompt for the ICD Debugger and reset the system.

```
b::

RESet
```

The device prompt `B::` is normally already selected in the TRACE32 command line. If this is not the case, enter `B::` to set the correct device prompt.

The **RESet** command is only necessary if you do not start directly after booting the TRACE32 development tool.

2. Specify the CPU.

```
SYStem.CPU NIOSII
```

3. Enter the debug mode.

```
SYStem.Up
```

This command resets the CPU and enters debug mode. After this command is executed, it is possible to access memory and registers.

4. Load the program.

```
Data.LOAD.Elf taskc.elf      ; load program and symbol information
```

The format of the **Data.LOAD** command depends on the file format generated by the compiler.

A detailed description of the **Data.LOAD** command and all available options is given in the reference guide.

Complete start-up example

A typical start sequence is shown below. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** *<file>*.

```
B::                                ; Select the ICD device prompt

WinCLEAR                           ; Clear all windows

SYStem.CPU NIOSII                  ; Select CPU type

SYStem.Up                          ; Reset the target and enter debug mode

Data.LOAD.Elf taskc.elf            ; Load the application

Register.Set pc main               ; Set the PC to function main

List.Mix                           ; Open source code window          *)

Register.view /SpotLight           ; Open register window             *)

Frame.view /Locals /Caller         ; Open the stack frame with
                                   ; local variables                  *)

Var.Watch %Spotlight flags ast     ; Open watch window for variables *)
```

*) These commands open windows on the screen. The window position can be specified with the **WinPOS** command.

Number of symbols	In demo version, the number of symbols is limited to 5000.
Address range	All peripherals are memory mapped. The hole address range (32 bit) is supported. The simulator operates as if there is a physical memory of 4 GByte.
Interrupt requests while the program execution is stopped	Exceptions and interrupts are not handled while the program execution is stopped. (Only relevant if interrupt-triggering peripherals are simulated.)
Pending interrupts during single-step	(Only relevant if interrupt-triggering peripherals are simulated.) The commands **SETUP.IMASKASM** and **SETUP.IMASKHLL** disable interrupts while single-stepping. This prevent to always end up in a interrupt service routine while single stepping the code.

SYStem.CPU — Select CPU type

At the moment the only CPU type which can be selected is “Nios II”.

SYStem.LOCK — Lock and tristate the debug port

Format: **SYStem.LOCK** [**ON** | **OFF**]

Default: OFF.

If the system is locked, no access to the debug port will be performed by the debugger. While locked, the debug connector of the debugger is tristated. The main intention of the **SYStem.LOCK** command is to give debug access to another tool. The command has no effect for the simulator.

SYStem.MemAccess — Select run-time memory access method

Format: **SYStem.MemAccess Enable | StopAndGo | Denied**
SYStem.ACCESS (deprecated)

Enable / **CPU** (deprecated) — Memory access during program execution to target is enabled.

Denied — Memory access during program execution to target is disabled.

StopAndGo — Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed.

Format:	**SYStem.Mode** <*mode*> **SYStem.Down** **(alias for SYStem.Mode Down)** **SYStem.Up** **(alias for SYStem.Mode Up)**
<*mode*>:	**Down** **Up**

Down The CPU is in reset. Debug mode is not active. Default state and state after fatal errors (default).

Up The CPU is not in reset but halted. Debug mode is active. In this mode the CPU can be started and stopped. This is the most typical way to activate debugging.

SYStem.CONFIG Configure debugger according to target topology

<*parameter*>:	**DRPRE** <*bits*>
(JTAG):	**DRPOST** <*bits*>
	IRPRE <*bits*>
	IRPOST <*bits*>
	TAPState <*state*>
	TCKLevel <*level*>
	TriState [**ON** | **OFF**]
	Slave [**ON** | **OFF**]

The **SYStem.CONFIG** commands have no effect in Simulator. These commands describe the physical configuration at the JTAG port and the trace port of a multi-core hardware target. Since the simulator normally just simulates the instruction set, these commands will be ignored. Refer to the relevant **Processor Architecture Manual** in case you want to know the effect of these commands on a debugger.

SYStem.Option.DCFLUSH Flush data cache before Step or Go

Format: **SYStem.Option.DCFLUSH** [**ON** | **OFF**]

Flush the data cache before starting the target program (Step or Go). Only relevant if cache is simulated (necessary for cache coherence).

SYStem.Option.Endianness Select endianness of core

Format: **SYStem.Option.Endianness** [**AUTO** | **Little** | **Big**]

Default: AUTO.

This option tells the simulator if you use a Little- or Big-Endian Nios II core. If you select AUTO, the simulator will use Little Endian byte order.

SYStem.Option.EXCADDR Define exception address

Format: **SYStem.Option.EXCADDR** *<address>*

The Nios II core uses a fixed exception address, which is defined in the SOPC Builder. To mimic the behavior of a real Nios II core, you can use this option to define the exception address, which is used by the simulator.

SYStem.Option.FEXCADDR Define fast TLB miss exception address

Format: **SYStem.Option.FEXCADDR** *<address>*

A Nios II core with MMU will jump to the fast TLB miss exception address if a virtual to physical memory address translation fails because of a missing TLB entry. To mimic the behavior of a real Nios II core, you can use this option to define the fast TLB miss exception address.

SYStem.Option.FPH Enable the simulation of floating point instructions

Format: **SYStem.Option.FPH** [**ON** | **OFF**]

Default: OFF.

OFF Floating point instructions are not simulated.
The **List** window does not display the mnemonics of floating point instructions.

ON Floating point instructions are simulated.
The **List** window displays the mnemonics of floating point instructions.

SYStem.Option.ICFLUSH Invalidate instruction cache before go/step

Format: **SYStem.Option.ICFLUSH** [**ON** | **OFF**]

Invalidates the instruction cache before starting the target program (Step or Go). This is required when the CACHEs are enabled and software breakpoints are set to a cached location. Only relevant if cache is simulated.

SYStem.Option.IMASKASM Mask interrupts during assembler step

Format: **SYStem.Option.IMASKASM** [**ON** | **OFF**]

If enabled, the interrupt mask bits of the cpu will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step. Only relevant if interrupt-triggering peripherals are simulated.

Format: **SYStem.Option.IMASKHLL** [**ON** | **OFF**]

If enabled, the interrupt mask bits of the cpu will be set during HLL single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step. Only relevant if interrupt-triggering peripherals are simulated.

NOTE:	By changing the status register through target software, this option can affect the flow of the target program. Accesses to the interrupt-mask bits will see the wrong values.

SYStem.Option.IVRCode Define code for interrupt vector instruction

Format: **SYStem.Option.IVRCode** *<code>*

Altera offers a custom instruction which is called Interrupt Vector Instruction. With this instruction you can speed up your exception handling. To mimic the behavior of a real Nios II core, the simulator is able to simulate this special Interrupt Vector Instruction. Because this instruction doesn't have a fixed opcode, you have to specify the "N value" of the custom instruction (see the description of the Interrupt Vector Instruction in the Nios II documentation from Altera).

If you leave out the "N value", the simulator will assume that you don't have an Interrupt Vector Instruction.

SYStem.Option.MMUSPACES Separate address spaces by space IDs

Format: **SYStem.Option.MMUSPACES** [**ON** | **OFF**]
SYStem.Option.MMUspaces [**ON** | **OFF**] (deprecated)
SYStem.Option.MMU [**ON** | **OFF**] (deprecated)

Default: OFF.

Enables the use of space IDs for logical addresses to support **multiple** address spaces.

For an explanation of the TRACE32 concept of address spaces (zone spaces, MMU spaces, and machine spaces), see **"TRACE32 Concepts"** (trace32_concepts.pdf).

NOTE:	**SYStem.Option.MMUSPACES** should not be set to **ON** if only one translation table is used on the target. If a debug session requires space IDs, you must observe the following sequence of steps: 1. Activate **SYStem.Option.MMUSPACES**. 2. Load the symbols with **Data.LOAD**. Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

```
;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x012A:
Data.dump D:0x012A:0xC00208A

;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x0203:
Data.dump D:0x0203:0xC00208A
```

Format: **SYStem.Option.MULDIV** <*mode*>

<*mode*>: **NONE**
MUL
DIV
MULDIV

The Nios II core can either support multiply and divide assembler instructions or it can trigger an exception if such an instruction is executed. The behavior depends on the settings you choose, when you generate the core in the SOPC Builder. To mimic the behavior of a real Nios II core, you can use this option to define if the multiply and divide assembler instructions are implemented or not.

NONE No multiply or divide assembler instruction is implemented. Trap to exception vector if such an instruction is executed

MUL Multiply assembler instructions work as expected. Divide assembler instructions will trap to the exception vector.

DIV Divide assembler instructions work as expected. Multiply assembler instructions will trap to the exception vector.

MULDIV Multiply and Divide instructions work as expected.

SYStem.Option.SIMMMU Define properties of simulated MMU

Format: **SYStem.Option.SIMMMU** <*number*>

The Nios II core offers the option to implement an MMU. The simulator supports MMU simulation. To enable MMU simulation you have to specify the properties of the MMU with this option.

<*number*> Bit 3..0 : Logarithm of base two of the total number of TLB entries. This means: 0 => 1 entry, 1 => 2 entries, 2 => 4 entries, 3 => 8 entries, ...

Bit 7..4 : Logarithm of base two of the number of TLB ways. This means: 0 => 1 way, 1 => 2 ways, 2 => 4 ways, 3 => 8 ways, 4 => 16 ways, ...

Bit 11..8 : Number of bits used for the process ID (PID).

TrOnchip.state — Display on-chip trigger window

Format: **TrOnchip.state**

Opens the **TrOnchip.state** window.

TrOnchip.RESet — Set on-chip trigger to default state

Format: **TrOnchip.RESet**

Sets the TrOnchip settings and trigger module to the default settings.

MMU.DUMP Page wise display of MMU translation table

Format:	**MMU.DUMP** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] **MMU.***<table>***.dump** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** *<cpu_specific_tables>*

Displays the contents of the CPU specific MMU translation table.

- If called without parameters, the complete table will be displayed.
- If the command is called with either an address range or an explicit address, table entries will only be displayed if their **logical** address matches with the given parameter.

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Displays the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: displays the translation table of the specified process • else, this command displays the table the CPU currently uses for MMU translation.

KernelPageTable	Displays the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and displays its table entries.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Displays the MMU translation table entries of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and displays its table entries. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

CPU specific Tables for MMU.DUMP

ITLB	Displays the contents of the Instruction Translation Lookaside Buffer.
DTLB	Displays the contents of the Data Translation Lookaside Buffer.
TLB	Displays the contents of the Translation Lookaside Buffer.

Format:	**MMU.List** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] **MMU.***<table>***.List** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**

Lists the address translation of the CPU-specific MMU table.

- If called without address or range parameters, the complete table will be displayed.
- If called without a table specifier, this command shows the debugger-internal translation table. See **TRANSlation.List**.
- If the command is called with either an address range or an explicit address, table entries will only be displayed if their **logical** address matches with the given parameter.

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Lists the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: list the translation table of the specified process • else, this command lists the table the CPU currently uses for MMU translation.
KernelPageTable	Lists the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and lists its address translation.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Lists the MMU translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and lists its address translation. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

Format:	**MMU.SCAN** *<table>* [*<range>* *<address>*] **MMU.***<table>***.SCAN** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** **ALL** [**Clear**] *<cpu_specific_tables>*

Loads the CPU-specific MMU translation table from the CPU to the debugger-internal static translation table.

- If called without parameters, the complete page table will be loaded. The list of static address translations can be viewed with **TRANSlation.List**.
- If the command is called with either an address range or an explicit address, page table entries will only be loaded if their **logical** address matches with the given parameter.

Use this command to make the translation information available for the debugger even when the program execution is running and the debugger has no access to the page tables and TLBs. This is required for the real-time memory access. Use the command **TRANSlation.ON** to enable the debugger-internal MMU table.

PageTable	Loads the entries of an MMU translation table and copies the address translation into the debugger-internal static translation table. • if *<range>* or *<address>* have a space ID: loads the translation table of the specified process • else, this command loads the table the CPU currently uses for MMU translation.

KernelPageTable	Loads the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Loads the MMU address translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and copies its address translation into the debugger-internal static translation table. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manual**.
ALL [**Clear**]	Loads all known MMU address translations. This command reads the OS kernel MMU table and the MMU tables of all processes and copies the complete address translation into the debugger-internal static translation table. See also the appropriate **OS Awareness Manual**. **Clear:** This option allows to clear the static translations list before reading it from all page translation tables.

Memory Classes

Overview

Access Class	Description
D	Data
P	Program
NC	No Cache

All storage classes operate on the same simulated memory. In real environment, the P and NC class operate directly on memory, while D accesses via cache.

- Not supported.

FAQ

Please refer to https://support.lauterbach.com/kb.