LAUTERBACH
DEVELOPMENT TOOLS

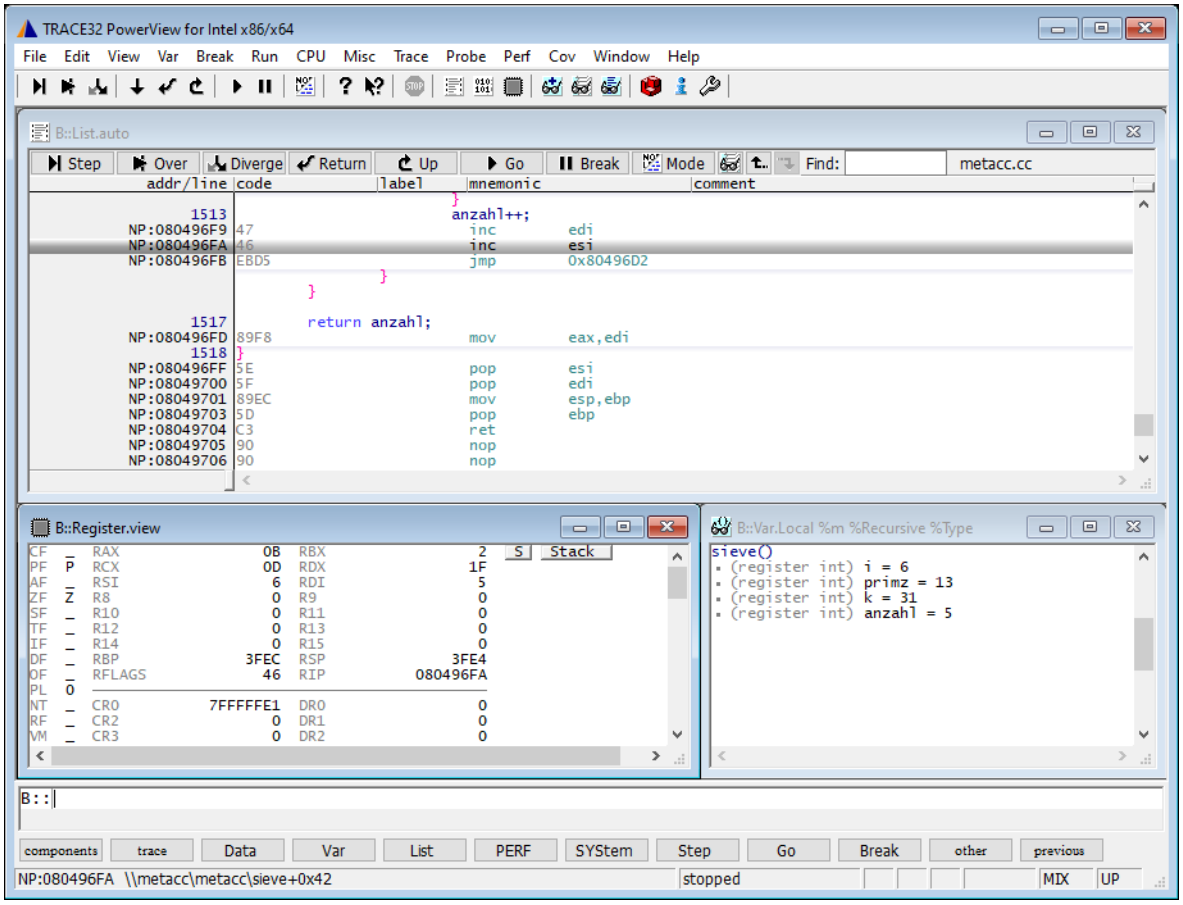
Simulator for Intel® x86/x64

MANUAL

Simulator for Intel® x86/x64

Simulator for Intel® x86/x64

Version 06-Jun-2024



All general commands are described in the **"PowerView Command Reference"** (ide_ref.pdf) and **"General Commands Reference"**.

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**.

To start the simulator, proceed as follows:

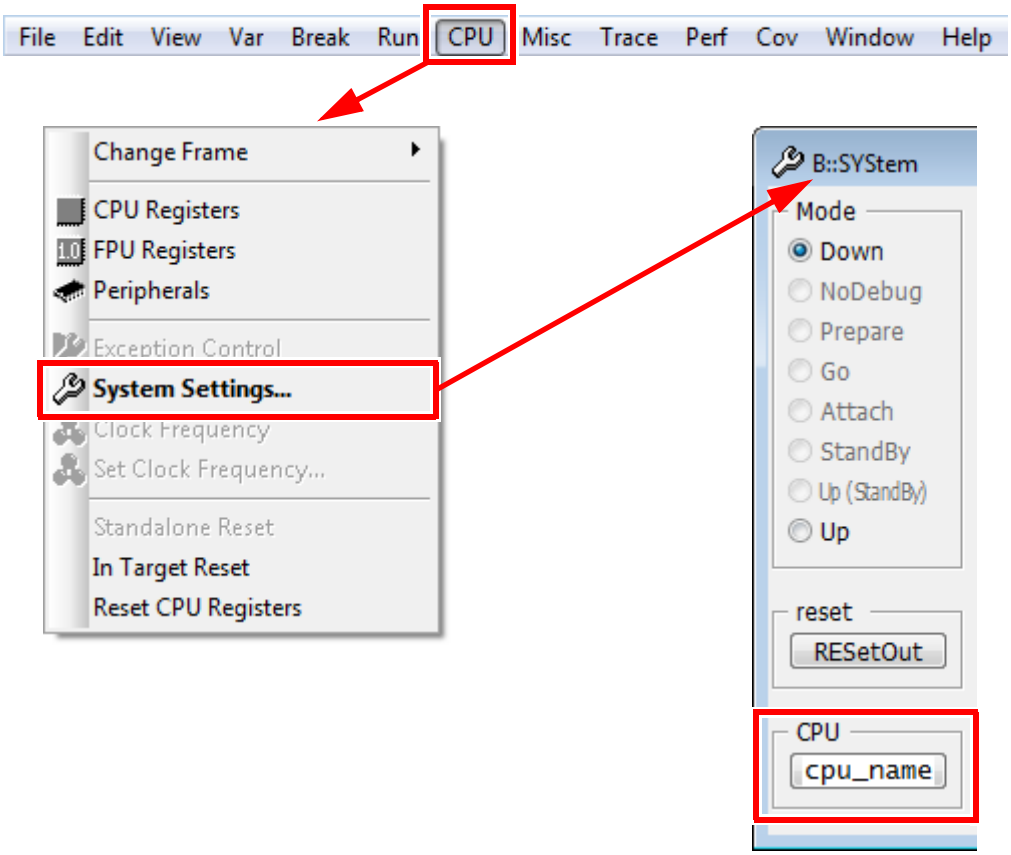
1. Select the device prompt for the Simulator and reset the system.

```
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 TRACE32.

2. Specify the CPU specific settings.




```
SYStem.CPU <cpu_name>
```

The default values of all other options are set in such a way that it should be possible to work without modification. Please consider that this is probably not the best configuration for your target.

3. Enter 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.<file_format> <file>      ; load program and symbols
```

See the **Data.LOAD** command reference for a list of supported file formats. If uncertain about the required format, try **Data.LOAD.auto**.

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

5. 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 <cpu_name>               ; Select CPU type

SYStem.Up                           ; Reset the target and enter
                                    ; debug mode

Data.LOAD.<file_format> <file>      ; Load the application

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

PER.view                            ; Show clearly arranged
                                    ; peripherals in window      *)

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.

For more information, see **“API for TRACE32 Instruction Set Simulator”** (simulator_api.pdf).

Access Classes

Overview

Access Class	Description
C	Generic
D	Data
P	Program
A	Absolute
AD	Absolute Data
AP	Absolute Program
I	Intermediate
ID	Intermediate Data
IP	Intermediate Program
L	Linear
LD	Linear Data
LP	Linear Program
R	Real Mode
RD	Real Mode Data
RP	Real Mode Program
ARD	Absolute Real Mode Data
ARP	Absolute Real Mode Program
LRD	Linear Real Mode Data
LRP	Linear Real Mode Program
N	Protected Mode (32-bit)

Access Class	Description
ND	Protected Mode Data (32-bit)
NP	Protected Mode Program (32-bit)
AND	Absolute Protected Mode Data (32-bit)
ANP	Absolute Protected Mode Program (32-bit)
LND	Linear Protected Mode Data (32-bit)
LRP	Linear Protected Mode Program (32-bit)
X	64-bit Mode
XD	64-bit Mode Data
XP	64-bit Mode Program
AXD	Absolute 64-bit Mode Data
AXP	Absolute 64-bit Mode Program
LXD	Linear 64-bit Mode Data
LXP	Linear 64-bit Mode Program
O	Protected Mode (16-bit)
OD	Protected Mode Data (16-bit)
OP	Protected Mode Program (16-bit)
AOD	Absolute Protected Mode Data (16-bit)
AOP	Absolute Protected Mode Program (16-bit)
LOD	Linear Protected Mode Data (16-bit)
LOP	Linear Protected Mode Program (16-bit)
IO	IO Ports
MSR	MSR Registers
CID	CPUID Instruction
VMCS	VMCS Registers
IOSF	IOSF Sideband
Q	Real Big Mode (Real Mode supporting 32-bit addresses)
QD	Real Big Mode Data

Access Class	Description
QP	Real Big Mode Program
AQD	Absolute Real Big Mode Data
AQP	Absolute Real Big Mode Program
LQD	Linear Real Big Mode Data
LQP	Linear Real Big Mode Program
E	Run-time Memory Access
S	System Management Mode (SMM)
SD	SMM Data
SP	SMM Program
SN	SMM Protected Mode (32-bit)
SND	SMM Protected Mode Data (32-bit)
SNP	SMM Protected Mode Program (32-bit)
SX	SMM 64-bit Mode
SXD	SMM 64-bit Mode Data
SXP	SMM 64-bit Mode Program
SO	SMM Protected Mode (16-bit)
SOD	SMM Protected Mode Data (16-bit)
SOP	SMM Protected Mode Program (16-bit)
SQ	SMM Real Big Mode (Real Mode supporting 32-bit addresses)
SQD	SMM Real Big Mode Data
SQP	SMM Real Big Mode Program
AS	Absolute SMM
ASD	Absolute SMM Data
ASP	Absolute SMM Program
LS	Linear SMM
LSD	Linear SMM Data
LSP	Linear SMM Program
G	VMX Guest Mode

Access Class	Description
H	VMX Host Mode
CSS	Current value of CS
DSS	Current value of DS
SSS	Current value of SS
ESS	Current value of ES
FSS	Current value of FS
GSS	Current value of GS

D:, P:

The D: prefix refers to the DS segment register and the P: prefix to the CS segment register. Both D: and P: memory classes access the same memory. It is not possible to split program and data memory. Real Mode or Protected Mode (16, 32 or 64-bit) addressing is chosen dependent on the current processor mode.

```
Data.Set  P:0x0--0x0ffff  0x0    ; fill program memory with zero

Data.Set  0x0--0x0ffff    0x0    ; fill data memory with zero

Data.Set  0x100 0x0              ; set location DS:0x100 to 0

Data.Assemble 0x100 nop          ; assemble to location CS:0x100

Data.Assemble 0x0--0x0fff  nop   ; fill program memory with nop
                                 ; instruction
```

A:, AD:, AP:

Absolute addressing. The address parameter specifies the absolute address thus disregarding segmentation and paging. It is possible to use “A” as a prefix to most other memory classes.

```
Data.Set  A:0x12000 0x33         ; write to absolute address 0x12000 in
                                 ; program/data memory

Data.dump AD:0x12000             ; displays absolute address 0x12000
                                 ; from data memory
```

I:, ID:, IP:

Intermediate addressing. This memory class is used in connection with virtualization. It corresponds to the guest physical address, i.e., disregards segmentation and paging of the guest, but does not disregard possible second level paging done by the host (use A: for that).

```
Data.Set  I:0x12000 0x33          ; write to guest absolute address
                                  ; 0x12000 in program/data memory

Data.dump ID:0x12000              ; displays guest absolute address
                                  ; 0x12000 from data memory
```

L:, LD:, LP:

Linear addressing. The address parameter specifies the linear address thus disregarding segmentation but not paging. It is possible to use “L” as a prefix to most other memory classes.

```
Data.Set  L:0x12000 0x33     ; write to linear address 0x12000 in
                             ; program/data memory

Data.dump LD:0x12000         ; displays absolute address 0x12000
                             ; from data memory
```

R:, RD:, RP:

Real Mode addressing.

```
Data.Set  R:0x1234:0x5678    ; write to Real Mode address 0x1234:0x5678

Data.Set  R:0x100            ; write to Real Mode address DS:0x100
```

N:, ND:, NP:

Protected Mode (32-bit) addressing. (“N” is for **N**ormal.)

```
Data.Set  N:0x0f0:0x5678     ; write to Protected Mode address 0x5678 of
                             ; selector 0x0f0

Data.dump ND:0x12345678      ; display memory at Protected Mode address
                             ; DS:0x12345678

Data.List NP:0x0C000000      ; disassemble memory in 32-bit mode at
                             ; Protected Mode address CS:0x0C000000
```

X:, XD:, XP:

64-bit Mode addressing. (“X” is for e**X**tended.)

```
Data.dump XD:0x0000123456789ABC       ;display memory at 64-bit Mode
                                      ;linear address 0x0000123456789ABC
```

O:, OD:, OP:

Protected Mode (16-bit) addressing. (“O” is for **O**ld.)

```
Data.List  OP:0x4321              ; disassemble memory in 16-bit mode at
                                  ; Protected Mode address CS:0x4321
```

Q:, QD:, QP:

Big Real Mode addressing. Real Mode (16-bit opcodes), supporting 32-bit addresses.
See **SYStem.Option.BIGREALmode ON** for details.

```
Data.Set                     ; write to 32-bit Big Real Mode address
Q:0x1234:0x5678ABCD          0x1234:0x5678ABCD

Data.Set Q:0x10008000        ; write to 32-bit Big Real Mode address
                             DS:0x10008000
```

IO:

Access IO ports.

```
Data.Out   IO:0xCF8 %long 0xF    ; output 32-bit value 0xF at IO port
                                 ; 0xCF8
```

MSR:

Accesses MSR registers. The address format is as follows:

Bits	Meaning
23-0	MSR[23-0]
27-24	MSR[31-28]
31-28	Ignored

```
Data.dump  msr:0x0               ; display MSR registers starting with
                                 ; MSR register 0

Data.dump  msr:0x0C000080        ; display MSR registers starting with
                                 ; MSR register 0xC0000080
```

CID:

Return CPUID values. The address format is as follows:

Bits	Meaning
1-0	Return Register (0=EAX, 1=EBX, 2=ECX, 3=EDX)
3-2	Ignored
14-4	EAX[10-0]
15	EAX[31]
29-16	ECX[13-0]
31-30	Ignored

```
Data.dump   cid:0x0             ; display CPUID values starting with
                                ; initial EAX value 0x0

Data.dump   cid:0x8020          ; display CPUID values starting with
                                ; initial EAX value 0x80000002

Data.In     cid:0x20041         ; return EBX CPUID value with initial
                                ; EAX value 0x4 and initial ECX value
                                ; 0x2
```

VMCS:

Access virtual-machine control data structures (VMCSs). The “address” to be used with this memory class is the corresponding field encoding of an VMCS component.

```
Data.In     VMCS:0x6C00         ; display the host CR0 VMCS component
```

IOSF:

Access IOSF sideband.

The address format uses a "**:*<offset>*" syntax, where the "segment" is 16 bits, and the "offset" 64 bits:

IOSF:<8-bit Opcode><8-bit PortID>:<8-bit FID><4-bit BAR><4-bit Reserved><48-bit Address>

"Segment" part:

Bits	Meaning
7-0	Port ID
15-8	Opcode

"Offset" part:

Bits	Meaning
47-0	Address
51-48	Reserved
55-52	BAR
63-56	FID

```
Data.In IOSF:0x0608:3C /long      ; Read IOSF sideband with opcode 0x06,
                                  ; port ID 0x08 and address 0x3C.
                                  ; (FID and BAR are both 0)

Data.Set IOSF:0x0608:3C %long     ; Write IOSF sideband with opcode 0x06,
0xdeadbeef                        ; port ID 0x08 and address 0x3C.
                                  ; (FID and BAR are both 0)

Data.In                           ; Read IOSF sideband with opcode 0x06,
IOSF:0x0608:0xFF701234567890A     ; port ID 0x08, FID 0xFF, BAR 0x7 and
B /long                           ; address 0x1234567890AB
```

E:

Run-time memory access. This access class must be used for any kind of run-time memory access (be it intrusive or non-intrusive). For that, “E” can be used as a prefix to every other access class.

```
Data.dump   END:0x12345678        ; display memory at Protected Mode
                                  ; address DS:0x12345678 during run-time
```

S:, SD:, SP:, SN:, SND:, SNP:, SX:, SXD:, SXP:, SO:, SOD:, SOP:, SQ:, SQD:, SQP: SR:

The “S” prefix refers to System Management Mode. All these access classes behave like the corresponding ones without the “S” only that they refer to SMM memory instead of normal memory.

```
Data.dump   ASD:0x3f300000        ; display SMM memory at absolute
                                  ; address 0x3f300000
```

G:, GD:, GP:, GN:, GND:, GNP:, GX:, GXD:, GXP:, GO:, GOD:, GOP:, GQ:, GQD:, GQP:
GS:, GSD:, GSP:, GSN:, GSND:, GSNP:, GSX:, GSXD:, GSXP:, GSO:, GSOD:, GSOP:, GSQ:, GSQD:, GSQP: GSR:

When the VMX mode of the target is enabled, TRACE32 indicates the affiliation of logical or linear addresses with the VMX **G**uest mode by adding the prefix “**G**” to the access class.

```
Data.dump   GD:0x2a000000         ; display data memory of address
                                  ; 0x2a000000 belonging to VMX Guest
                                  ; mode
```

H:, HD:, HP:, HN:, HND:, HNP:, HX:, HXD:, HXP:, HO:, HOD:, HOP:, HQ:, HQD:, HQP:
HS:, HSD:, HSP:, HSN:, HSND:, HSNP:, HSX:, HSXD:, HSXP:, HSO:, HSOD:, HSOP:, HSQ:, HSQD:, HSQP: HSR:

When the VMX mode of the target is enabled, TRACE32 indicates the affiliation of logical or linear addresses with the VMX **H**ost mode by adding prefix “**H**” to the access class.

```
Data.dump   HD:0x2a000000         ; display data memory of address
                                  ; 0x2a000000 belonging to VMX Host
                                  ; mode
```

Segment register aliases CSS:, DSS:, SSS:, ESS:, FSS:, GSS:

These are not real access classes but aliases which allow to modify the segment descriptor of an address. If one of these six identifiers precedes an address, the value of segment register CS, DS, SS, ES, FS or GS will be used as descriptor in the address.

These aliases are of use only if you want to work directly with segment based addressing in real or protected mode. Note that **SYStem.Option.MEMoryMODEL** must be set to LARGE to support **segmentation** to its fullest extent in protected mode.

Example: Let's assume the processor is in protected mode and the segment register FS contains the value 0x18 which is a 32-bit data segment. We want to write to an address with offset 0x12000, using FS as segment register.

```
Data.Set    FSR:0x12000 0x33      ; write 0x33 to address FSR:0x12000.
                                  ; Effectively, this will use 0x18 as
                                  ; segment descriptor.
                                  ; (If we are in protected mode and FS
                                  ; is a 32-bit data segment) you could
                                  ; alternatively use
                                  ; Data.Set ND:0x18:0x12000 0x33
                                  ;               ^ FS contains 0x18

Data.dump   SSR:0x12000           ; display memory at SSR:0x12000
```

NOTE: To avoid confusion with the access classes **ES:** and **GS:**, all six segment selector identifiers have been renamed from CS:, DS:, ES:, FS:, GS:, SS: to **CSS:**, **DSS:**, **ESS:**, **FSS:**, **GSS:**, **SSS:** as of TRACE32 build 75425 - DVD 09/2016.

- Prefix **ES:** indicates an unspecific (non-program and non-data) dual-port memory accesses in System Management Mode.
- Prefix **GS:** indicates an unspecific system management memory access in VMX Guest Mode.

Memory Model

The Intel® x86 memory model describes the way the debugger considers the six segments CS (code segment), DS (data segment), SS (stack segment), ES, FS and GS and the usage of the LDT (local descriptor table) for the current debug session.

A further introduction into the concept of x86 memory models can be found in the Intel® software developer's manual (please refer to the chapter describing segments in protected mode memory management).

TRACE32 supports a number of memory models when working with addresses and segments: **LARGE**, **FLAT**, **ProtectedFLAT**, **LDT** and **SingleLDT**. Activating the space IDs with **SYStem.Option.MMUSPACES ON** will override any other selected memory model. TRACE32 now behaves as if the memory model **FLAT** is selected and additionally uses space IDs in the address to identify process-specific address spaces (see **SYStem.Option.MMUSPACES** for more details).

Effect of the Memory Model on the Debugger Operation

In protected mode, the address translation of x86 processors support segment translation and paging (if enabled). Segment translation cannot be disabled in hardware. If the TRACE32 address translation is enabled (**TRANSlation.ON**, **TRANSlation.TableWalk ON**), the same translation steps are executed when the debugger performs a memory access to protected mode addresses.

The values loaded into *base*, *limit* and *attribute* of the segment registers CS, DS, ES, FS, GS and SS depend on the code being executed and how it makes use of the segments. Setup of the segment registers is an essential step in loading executable code into memory. Choosing the appropriate TRACE32 memory model adjusts the segment register handling on the debugger side to the segment register handling on the software side.

For this purpose, TRACE32 offers six memory models. The memory model affects:

- The TRACE32 address format
- Whether or not segment information is used when the debugger accesses memory
- Whether a LDT descriptor is used to dynamically fetch code and data segments from the local descriptor table LDT when the debugger accesses memory
- The way how the segment base and limit values are evaluated when an address is translated from a protected mode address into a linear and/or physical address
- The way the segment attribute information such as code or data width (16/32/64 bit) is evaluated when code or data memory is accessed

For a more detailed description of the memory models supported by TRACE32, see **SYStem.Option.MEMoryMODEL**.

After reset, the TRACE32 memory model **LARGE** is enabled by default. Use one of the following commands to select a different TRACE32 memory model for the current debug session:

1. **SYStem.Option.MEMoryMODEL**
2. **SYStem.Option.MMUSPACES**
3. **Data.LOAD** - When loading an executable file, specify one of these command options **FLAT**, **ProtectedFLAT**, **SingleLDT**, **LDT**, or **LARGE** to select the TRACE32 memory model you want to apply to the executable.

The PRACTICE function **SYStem.Option.MEMoryMODEL()** returns the name of the currently enabled memory model.

```
PRINT SYStem.Option.MEMoryMODEL()     ;print the name of the memory model
                                      ;to the TRACE32 message line
```

Segmentation

TRACE32 allows to work with segments, both in real and in protected mode. If the debugger address translation is enabled with **TRANSlation.ON**, real mode or protected mode addresses will be translated to linear addresses. If paging is enabled on the target and the TRACE32 table walk mechanism is enabled with **TRANSlation.TableWalk ON**, the linear addresses will finally be translated to physical addresses.

Segment translation by TRACE32 is only supported if **SYStem.Option.MEMoryMODEL** is set to one of these settings: **LARGE**, **ProtectedFLAT**, **LDT**, **SingleLDT**. For a description of these option, see **SYStem.Option.MEMoryMODEL**. The default option **LARGE**, selected after **SYStem.Up**, is suitable for most debug scenarios where segment translation is used.

Protected mode addresses can be recognized by one of these access classes:

- X:, XD:, XP: (64-bit protected mode)
- N:, ND:, NP: (32-bit protected mode)
- O:, OD:, OP: (16-bit protected mode)

If no segment descriptor is given for such an address, the descriptor from the code segment register (CS) will be augmented to program addresses, and the segment descriptor from the data segment register (DS) will be augmented to data addresses. The command **MMU.view** can be used to view the current settings of the six segment registers CS, DS, ES, FS, GS, and SS. The augmented segment descriptor is shown as part of the address.

During segment translation of a protected mode address, TRACE32 will extract the segment descriptor from the address and search for it in the six segment registers CS, DS, ES, FS, and GS. If found, the stored values of the segment shadow register (base, limit and attribute) will be used for the linear translation of the protected mode address. Else, a descriptor table walk will be performed through the global descriptor table

GDT, provided the register GDTB (global descriptor table base) points to a valid GDT in memory. If found, the base, limit, and attribute from the GDT entry will be used for the translation. If the address' segment descriptor is not found in the GDT, or the GDT entry is not suitable for the translation of the given address type, the protected mode address cannot be translated to a linear address by TRACE32.

It is possible to explicitly enforce one of the six segment registers CS, DS, ES, FS, GS or SS to be used for the segment translation of an address. This can be accomplished by specifying the segment register instead of a protected mode access class. Use one of the segment register identifiers CSS:, DSS:, ESS:, FSS:, GSS: or SSS: therefore.

Example: The address in this **Data.dump** command will use the segment descriptor of segment register FS instead of the default segment descriptor from segment register DS.

```
Data.dump FSS:0xa7000
```

NOTE:	TRACE32 will not perform segment translation at if the processor is in 64-bit mode (IA-32e mode). Further, no segment translation is performed for 64-bit protected mode addresses (addresses with access class X:, XD:, XP:). If no segment translation is performed, protected mode addresses are translated directly to linear addresses, disregarding the segment descriptor of the address. This mimics the behavior of the processor, which treats the segment base registers as zero and performs no segment limit checks if the IA-32e mode (64-bit mode) is enabled.

Troubleshooting

FAQ

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

SYStem.CONFIG — Configure debugger according to target topology

The **SYStem.CONFIG** commands have no effect on the simulator. They are only provided to allow the user to run PRACTICE scripts written for the debugger within the simulator without modifications.

SYStem.CPU — CPU type

Format:	**SYStem.CPU** *<mode>*
<mode>:	**I8086** | **I80186** | **I80186EA** | **I80186EB** | **I80186EC** | **AM186EM** | **AM186ES** | **AM186ER** | **AM186ED** | **AM186CC**

Selects the processor type.

SYStem.LOCK — Lock and tristate the debug port

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

The command has no effect for the simulator.

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

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

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.
For more information, see below.

SYStem.Mode Establish the communication with the simulator

Format: **SYStem.Mode** *<mode>*

SYStem.Down (alias for SYStem.Mode Down)
SYStem.Up (alias for SYStem.Mode Up)

<mode>:
Down
NoDebug
Go
Up

Default: Down.

Selects the target operating mode.

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

NoDebug The CPU is running. Debug mode is not active. Debug port is tristate. In this mode the target should behave as if the debugger is not connected.

Go The CPU is running. Debug mode is active. After this command the CPU can be stopped with the break command or if any break condition occurs.

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.

If the mode **Go** is selected, this mode will be entered, but the control button in the **SYStem.state** window jumps to the mode **Up**.

SYStem.Option.Address32 Use 32 bit address display only

Format: **SYStem.Option.Address32** [**ON** | **OFF**]

Default: OFF.

This option only has an effect when in 64-bit mode. When the option is ON, all addresses are truncated to 32 bit. The high 32 bits of a 64-bit address are not shown when the address is displayed, and when an address is entered the high 32 bits are ignored (thereby effectively being set to zero).

NOTE:	The actual memory access mode is NOT affected by this option.

SYStem.Option.IMASKASM Disable interrupts while single stepping

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

Default: OFF.

If enabled, the interrupt enable flag of the EFLAGS register will be cleared during assembler single-step operations. After the single step, the interrupt enable flag is restored to the value it had before the step.

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

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

Default: OFF.

If enabled, the interrupt enable flag of the EFLAGS register will be cleared during HLL single-step operations. After the single step, the interrupt enable flag is restored to the value it had before the step.

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

Default: OFF

Enables the TRACE32 support for debugging virtualized systems. Virtualized systems are systems running under the control of a hypervisor.

After loading a Hypervisor Awareness, TRACE32 is able to access the context of each guest machine. Both currently active and currently inactive guest machines can be debugged.

If **SYStem.Option.MACHINESPACES** is set to **ON**:

- Addresses are extended with an identifier called machine ID. The machine ID clearly specifies to which host or guest machine the address belongs.

  The host machine always uses machine ID 0. Guests have a machine ID larger than 0. TRACE32 currently supports machine IDs up to 30.
- The debugger address translation (**MMU** and **TRANSlation** command groups) can be individually configured for each virtual machine.
- Individual symbol sets can be loaded for each virtual machine.

SYStem.Option.MEMoryMODEL Define memory model

Format: **SYStem.Option.MEMoryMODEL** *<model>*

<model>: **LARGE** | **FLAT** | **LDT** | **SingleLDT** | **ProtectedFLAT**

Default: LARGE (Multi-Segment Model).

Selects the memory model TRACE32 uses for code and data accesses. The memory model describes how the CS (code segment), DS (data segment), SS (stack segment), ES, FS and GS segment registers are currently used by the processor.

The command **SYStem.Option.MMUSPACES ON** will override the setting of **SYStem.Option.MEMoryMODEL** with the memory model **MMUSPACES**.

The selection of the memory model affects the following areas:

- The way TRACE32 augments program or data addresses with information from the segment descriptors. Information augmented is the segment selector, offset, limit and access width.
- The TRACE32 address format
- The way TRACE32 handles segments when the debugger address translation is enabled (**TRANSlation.ON**).

LARGE

This is the default memory model. It is enabled after reset. This memory model is used if the application makes use of the six segment registers (CS, DS, ES, FS, GS, SS) and the global descriptor table (GDT) and/or the local descriptor table (LDT).

TRACE32 supports GDT and LDT descriptor table walks in this memory model. If a TRACE32 address contains a segment descriptor and the specified segment descriptor is not present in any of the six segments CS, DS, ES, FS, GS or SS, TRACE32 will perform a descriptor table walk through the GDT or the LDT to extract the descriptor information and apply it to the address.

Access classes of program and data addresses will be augmented with information from the CS and DS segments.

Segment translation is used in TRACE32 address translation. See also **Segmentation**.

TRACE32 addresses display the segment selector to the left of the address offset. The segment selector indicates the GDT or LDT segment descriptor which is used for the address.

Example address: NP:0x0018:0x0003F000

LDT

This memory model should be selected if a LDT is present and the debugger uses multiple entries from it. TRACE32 addresses contain a LDTR segment selector specifying the LDT entry which applies to an address.

Access classes of program and data addresses will be augmented with the information specified by the LDTR segment selector.

Segment translation is used in TRACE32 address translation.

TRACE32 addresses display three numeric elements:

- The 16-bit LDTR segment selector used pointing to the LDT for the address
- The 16-bit CS (for program addresses) or DS (for data addresses) segment selector, extracted from the LDT
- The 16-bit address offset

Example address: NP:0x0004:0x0018:0x8000

SingleLDT

This memory model should be selected if a LDT is present but the debugger works with only one single LDT entry. The LDT is not used to differentiate addresses.

Access classes of program and data addresses will be augmented with information from the CS (for program addresses) or DS (for data addresses) segment.

Segment translation is used in TRACE32 address translation.

TRACE32 addresses display the segment selector to the left of the address offset.

Example address: NP:0x001C:0x0003F000

ProtectedFLAT

Use this memory model to only apply segment translation and limit checks for the segments CS and DS. The segment register contents are kept constant. Consequently, TRACE32 addresses contain no segment descriptor because no descriptor table walk is used to reload the segment registers.

Access classes of addresses are not augmented with segment information.

TRACE32 addresses display only the access class and the address offset.

Example address: NP:0x0003F000

Segment translation is used in TRACE32 address translation for limit checking. Accesses to program addresses use the CS segment, accesses to data addresses use the DS segment.

FLAT

This memory model is used if segmentation plays no role for an application and memory management makes use of paging only.

Segments are ignored, no segment translation is performed. Accesses to program and data addresses are treated the same.

Example address: NP:0x0003F000

MMUSPACES

This memory model can only be enabled with the command **SYStem.Option.MMUSPACES ON**.

The memory model MMUSPACES is used if TRACE32 works with an OS Awareness and memory space identifiers (space IDs). Space IDs are used in addresses to identify process-specific address spaces.

Segments are ignored, no segment translation is performed.

TRACE32 addresses display a 16-bit memory space identifier to the left of the address offset.

Example address: NP:0x29A:0x0003F000

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
```

NOTE: The command **SYStem.Option.MMUSPACES ON** overrides the command **SYStem.Option.MEMoryMODEL**.

Format:	**SYStem.Option.REL** *<value>*

REL option must be set to the same value the user program write to the REL register.

The adjusted I/O base address can be read back with the functions **IOBASE**() and **IOBASE.ADDRESS**(). They return the offset or the complete address (offset and access mode) for the I/O area.

SYStem.Option.ZoneSPACES Enable symbol management for zones

[Examples]

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

Default: OFF.

The **SYStem.Option.ZoneSPACES** command must be set to **ON** if separate symbol sets are used for the following CPU operation modes:

- VMX host mode (access class H: and related access classes)
- VMX guest mode (access class G: and related access classes)
- System management mode (access class S: and related access classes)
- Normal (non-system management mode)

Within TRACE32, these CPU operation modes are referred to as zones.

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

In each CPU operation mode (zone), the CPU uses separate MMU translation tables for memory accesses and separate register sets. Consequently, in each zone, different code and data can be visible on the same logical address.

OFF	TRACE32 does not separate symbols by access class. Loading two or more symbol sets with overlapping address ranges will result in unpredictable behavior. Loaded symbols are independent of the CPU mode.
ON	Separate symbol sets can be loaded for each zone, even with overlapping address ranges. Loaded symbols are specific to one of the CPU zones.

SYStem.Option.ZoneSPACES ON

SYStem.Option.ZoneSPACES is set to **ON** for two typical use cases:

- Debugging of virtualized systems. Typically separate symbol sets are used for the VMX host mode and the VMX guest mode. The symbol sets are loaded to the access classes H: (host mode) and G: (guest mode).
- Debugging of system management mode (SMM). The CPU typically enters and leaves the SMM, so loading separate symbol sets for the SMM and the normal mode are helpful. Symbols valid for the SMM zone use SMM access classes. SMM access classes are preceded by the letter S (such as SND:, SNP:, SXD:, SXP:). Symbols valid for the normal mode zone use access classes which are not preceded by the letter S (such as ND:, NP:, XD:, XP:).

If **SYStem.Option.ZoneSPACES** is **ON**, TRACE32 enforces any memory address specified in a TRACE32 command to have an access class which clearly indicates to which zone the memory address belongs.

If an address specified in a command uses an anonymous access class such as D:, P: or C:, the access class of the current PC context is used to complete the addresses' access class.

If a symbol is referenced by name, the associated access class of its zone will be used automatically, so that the memory access is done within the correct CPU mode context. As a result, the symbol's logical address will be translated to the physical address with the correct MMU translation table.

Examples

Example 1: Use **SYStem.Option.ZoneSPACES** for VMX host and guest debugging.

```
SYStem.Option.ZoneSPACES ON

; 1. Load the Xen hypervisor symbols for the VMX host mode
; (access classes H:, HP:and HD: are used for the symbols):
Data.LOAD.ELF xen-syms H:0x0 /NoCODE

; 2. Load the vmlinux symbols for the VMX guest mode
; (access classes G:, GP: and GD: are used for the symbols):
Data.LOAD.ELF vmlinux G:0x0 /NoCODE

; 3. Load the sieve symbols without specification of a target access
; class:
Data.LOAD.ELF sieve /NoCODE
; Assuming that the current CPU mode is VMX host mode in this example,
; the symbols of sieve will be assigned the access classes H:, HP:
; and HD: during loading.
```

Example 2: Use **SYStem.Option.ZoneSPACES** for system management mode (SMM) debugging.

```
SYStem.Option.ZoneSPACES ON

; 1. Load the symbols for non-SMM (normal) mode
; (32 bit protected mode access classes N:, NP: and ND:):
Data.LOAD.ELF bootloader N:0x0 /NoCODE

; 2. Load the symbols for the SMM mode
; (32 bit protected mode access classes SN:, SNP: and SND:):
Data.LOAD.ELF smmdriver SN:0x0 /NoCODE
```

MMU.DUMP Page wise display of MMU translation table

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

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 or a specific machine if a space ID and/or a machine ID is given.
PageTable	Displays the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID and/or machine ID: displays the translation table of the specified process and/or machine • 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**.
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*	The following options are only available if **SYStem.Option.MACHINESPACES** is set to **ON**. Dumps a page table of a virtual machine. The **MACHINE** option applies to **PageTable** and **KernelPageTable** and some *<cpu_specific_tables>.* The parameters *<machine_magic>*, *<machine_id>* and *<machine_name>* are displayed in the **TASK.List.MACHINES** window.
Fulltranslation	For page tables of guest machines both the intermediate address and the physical address is displayed in the **MMU.DUMP** window. The physical address is derived from a table walk using the guest's intermediate page table.

EPT	Displays the contents of the Extended Page Table (EPT). The EPT is used for VMX guest mode translations.
GDT **MMU.GDT** (deprecated)	Displays the contents of the Global Descriptor Table.
IDT **MMU.IDT** (deprecated)	Displays the contents of the Interrupt Descriptor Table.
LDT **MMU.LDT** (deprecated)	Displays the contents of the Local Descriptor Table.
IntermedPageTable	Displays the Intermediate Page Table (IPT). The IPT is the translation table used by the TRACE32 debugger address translation to translate intermediate addresses to physical addresses when **SYStem.Option.MACHINESPACES ON** is set. If the CPU's VMX mode is enabled, the IPT is identical to the EPT. When the VMX mode is not enabled or not available on a CPU, an IPT can be specified using the command **MMU.FORMAT** *<format> <ipt_base_address>* **/Intermediate**

Format:	**MMU.List** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] [**/***<option>*]
	MMU.*<table>***.List** (deprecated)
<table>:	**PageTable**
	KernelPageTable
	TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**
	<cpu_specific_tables>
<option>:	**MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>*
	Fulltranslation

Lists the address translation of the CPU-specific MMU table.
In contrast to **MMU.DUMP,** multiple consecutive page table entries with identical page attributes are listed as a single line, showing the total mapped address range.

- 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 or a specific machine if a space ID and/or a machine ID is given.
PageTable	Lists the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID and/or machine ID: list the translation table of the specified process and/or machine • 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**.
<option>	For description of the options, see **MMU.DUMP**.

CPU specific Tables in MMU.List <table>

EPT	Displays the contents of the Extended Page Table (EPT). The EPT is used for VMX guest mode translations.
IntermedPageTable	Displays the Intermediate Page Table (IPT). The IPT is the translation table used by the TRACE32 debugger address translation to translate intermediate addresses to physical addresses when **SYStem.Option.MACHINESPACES** is set to **ON**. If the CPU's VMX mode is enabled, the IPT is identical to the EPT. When the VMX mode is not enabled or not available on a CPU, an IPT can be specified using the command **MMU.FORMAT** *<ipt_base_address>* **/Intermediate**

Format:	**MMU.SCAN** *<table>* [*<range>* *<address>*] [*/<option>*] **MMU.***<table>***.SCAN** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** **ALL** *<cpu_specific_tables>*
<option>:	**MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **Fulltranslation**

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 and/or machine ID: loads the translation table of the specified process and/or machine • 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	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**.
<option>	For description of the options, see **MMU.DUMP**.

CPU specific Tables in MMU.SCAN <table>

EPT	Loads the translation entries of the Extended Page Table to the debugger-internal static translation table.
GDT	Loads the Global Descriptor Table from the CPU to the debugger-internal static translation table.
GDTLDT	Loads the Global and Local Descriptor Table from the CPU to the debugger-internal static translation table.
LDT	Loads the Local Descriptor Table from the CPU to the debugger-internal static translation table.
IntermedPageTable	Loads the Intermediate Page Table (**IPT**) into the debugger-internal static translation table. The **IPT** is the translation table used by the TRACE32 debugger address translation to translate intermediate addresses to physical addresses when **SYStem.Option.MACHINESPACES ON** is set. If the CPU's VMX mode is enabled, the **IPT** is identical to the **EPT**. When the VMX mode is not enabled or not available on a CPU, an **IPT** can be specified using command **MMU.FORMAT** *<ipt_base_address>* **/Intermediate**

TrOnchip — Onchip triggers

This command group has no effect on the TRACE32 Instruction Set Simulator.