





Intel® x86/x64 Debugger

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents	
ICD In-Circuit Debugger	
Processor Architecture Manuals	
x86	
Intel® x86/x64 Debugger	1
History	7
Brief Overview of Documents for New Users	8
Welcome Dialog	8
Help Menu	9
Further Documents	10
Warning	12
Quick Start	13
Troubleshooting	16
FAQ	16
x86 specific Implementations	17
Tool Identification	17
Onchip Breakpoints	17
Breakpoints after Reset/Power Cycle	18
Access Classes	19
Overview	19
Memory Model	30
Segmentation	31
Platform Controller Hub (PCH)	32
Debugging a CPU only	33
Debugging a PCH only	33
Debugging a CPU and a PCH	33
Systems Using a Merged Debug Port	33
Systems Using Separate Debug Ports	34
PCH Selection for CPU Debug on a Merged Debug Port	34
Slave Core Debugging	34
Start Master Debugger	35
Locating the Slave Core	35

Starting the Slave Debugger	36
CPU specific JTAG.CONFIG Commands	37
JTAG.CONFIG	Electrical characteristics of MIPI-60 debug signals 37
JTAG.CONFIG.DRiVer	Set slew rate of JTAG signals 37
JTAG.CONFIG.PowerDownTriState	Automatically tristate outputs 38
JTAG.CONFIG.TckRun	Free-running TCK mode 38
JTAG.CONFIG.TDOEdge	Select TCK edge 38
JTAG.CONFIG.Voltage.HookKThreshhold	Set hook threshold voltages 39
JTAG.CONFIG.Voltage.REFerence	Set reference voltage source 40
JTAG.CONFIG.Voltage.THreshhold	Set JTAG threshold voltages 40
CPU specific SYSTem.DETECT Commands	41
SYSTem.DETECT.CLTapchain	Show SOC IDs of SOC slave cores 41
SYSTem.DETECT.CORES	Detect core/thread number 41
SYSTem.DETECT.HyperThreads	Detect hyper thread status 42
SYSTem.DETECT.TARGET	Fully automatic board setup 43
SYSTem.DETECT.TOPOlogy	Detect board topology 44
CPU specific SYSTem Settings	45
SYSTem.CONFIG.state	Display target configuration 45
SYSTem.CONFIG	Configure debugger according to target topology 46
Multicore Settings (daisy chain)	46
SYSTem.CORESTATES	Core states overview 49
SYSTem.CPU	Select the target CPU/SOC 51
SYSTem.JtagClock	Define JTAG clock 51
SYSTem.LOCK	Tristate the JTAG port 51
SYSTem.MemAccess	Select run-time memory access method 52
SYSTem.Mode	Establish the communication with the target 52
SYSTem.Option.Address32	Use 32 bit address display only 53
SYSTem.Option.BIGREALmode	Enable Big Real mode handling 54
SYSTem.Option.BranchSTEP	Enables branch stepping 55
SYSTem.Option.BreakDELAY	Set max. break delay 55
SYSTem.Option.C0Hold	Hold CPU in C0 state 55
SYSTem.Option.IgnoreDEbugReDirections	Ignore debug redirections 56
SYSTem.Option.IgnoreSOC	Ignore SoC TAP chain structure 56
SYSTem.Option.IgnoreSWBPreDirections	Ignore SW BP redirections 56
SYSTem.Option.IMASKASM	Disable interrupts while single stepping 57
SYSTem.Option.IMASKHLL	Disable interrupts while HLL single stepping 57
SYSTem.Option.InstrSUBmitForcePHYSicalPRDY	Use physical PRDY 57
SYSTem.Option.InstrSUBmitIgnorePHYSicalPRDY	Ignore physical PRDY 57
SYSTem.Option.InstrSUBmitTimeout	Timeout for instruction submission 58
SYSTem.Option.IntelSOC	Slave core is part of Intel® SoC 58
SYSTem.Option.JTAGDirectCPU	JTAG directly to CPU TAPs 59
SYSTem.Option.JTAGOnly	Use only JTAG signals 59

SYStem.Option.MACHINESPACES	Address extension for guest OSes	59
SYStem.Option.MEMoryMODEL	Define memory model	60
SYStem.Option.MMUSPACES	Separate address spaces by space IDs	63
SYStem.Option.MultiCoreWhiskers	Server board whisker setup	64
SYStem.Option.NoDualcoreModule	Disable dualcore module support	64
SYStem.Option.NoHyperThread	Disable HyperThreading support	65
SYStem.Option.NoIPAdjust	Do not adjust IP at reset vector	65
SYStem.Option.NoReBoot	Disable watchdog causing reboot	65
SYStem.Option.OSWakeupTIME	Set the OS wake up time	66
SYStem.Option.PC10MODE	Wake up target from package C10	66
SYStem.Option.PreserveDRX	Preserve DRx resources	66
SYStem.Option.PreserveLBR	Preserve LBR resources	66
SYStem.Option.ProbeModeNOSaveRestore	No save/restore	67
SYStem.Option.ProbeModeONDEmand	On demand save/restore	68
SYStem.Option.PWRCycleTime	Set power cycle time	68
SYStem.Option.PWROFFTime	Set power off assertion time	68
SYStem.Option.PWRONTime	Set power on assertion time	69
SYStem.Option.PWRONWaitTime	Set power on time	69
SYStem.Option.ReArmBreakPoints	Rearm breakpoints on reset	69
SYStem.Option.REL	Relocation register	69
SYStem.Option.RESetDELAY	Set reset delay	70
SYStem.Option.RESetDetection	Select reset detection source	70
SYStem.Option.RESetMode	Select reset method	71
SYStem.Option.RESetTIME	Set reset assertion time	71
SYStem.Option.RESetWaitTIME	Set reset input wait time	71
SYStem.Option.S0Hold	Hold SoC in S0 state	72
SYStem.Option.SOFTLONG	Use 32-bit access to set SW breakpoint	72
SYStem.Option.STandBYAttach	In standby mode, only attach to target	72
SYStem.Option.STandBYAttachDELAY	Delay after standby	73
SYStem.Option.STepINToEXC	Step into interrupt or exception handler	73
SYStem.Option.TOPOlogy	Select server board topology	73
SYStem.Option.WatchDogWaitTIME	Set the reset watch dog time	74
SYStem.Option.WFSMemAccess	Allow WFS memory access	74
SYStem.Option.WHISKER	Select a whisker	74
SYStem.Option.ZoneSPACES	Enable symbol management for zones	75
SYStem.PCH	Select the target PCH	77
SYStem.POWER	Control target power	77
SYStem.STALLPhase	Set system into stall phase	78
SYStem.StuffInstruction	Submit instruction to CPU in probe mode	78
SYStem.StuffInstructionRead	Submit instruction and read	78
SYStem.TIMINGS	Display timings window	79
Command Groups for Special Registers		80
CPU specific MMU Commands		81

MMU.DUMP	Page wise display of MMU translation table	81
MMU.List	Compact display of MMU translation table	85
MMU.SCAN	Load MMU table from CPU	87
MMU.Set	Set MMU register	88
CPU specific TrOnchip Commands - Onchip Triggers		89
TrOnchip.PrintList	Print possible onchip triggers	89
TrOnchip.RESet	Reset settings to defaults	89
TrOnchip.Set	Break on event	89
TrOnchip.Set.BootStall	Enter bootstall	89
TrOnchip.Set.C6Exit	Break on C6 exit	91
TrOnchip.Set.ColdRESet	Break on cold reset	91
TrOnchip.Set.CpuBootStall	Enter CPU bootstall	91
TrOnchip.Set.ENCLU	Break on ENCLU event	92
TrOnchip.Set.GeneralDetect	Break on general detect	92
TrOnchip.Set.INIT	Break on init	92
TrOnchip.Set.MachineCheck	Break on machine check	92
TrOnchip.Set.RESet	Break on target reset	93
TrOnchip.Set.ShutDown	Break on shutdown	93
TrOnchip.Set.SMMENtRy	Break on SMM entry	93
TrOnchip.Set.SMMEXit	Break on SMM exit	93
TrOnchip.Set.SMMINto	Step into SMM when single stepping	94
TrOnchip.Set.TraceHub	Enter/leave trace hub break	94
TrOnchip.Set.VMENtRy	Break on VM entry	94
TrOnchip.Set.VMEXit	Break on VM exit	95
TrOnchip.state	Display onchip trigger window	97
CPU specific Events for the ON and GLOBALON Command		98
CPU specific BenchmarkCounter Commands		99
BMC.<counter>	Select BMC event to count	99
BMC.<counter>.COUNT	Select count mode for BMC	99
CPU specific Onchip Trace Commands		100
Onchip.Buffer	Configure onchip trace source	100
CPU specific Functions		102
SYStem.CoreStates.APIC()		102
SYStem.CoreStates.HYPER()		102
SYStem.CoreStates.MODE()		102
SYStem.CoreStates.PHYS()		103
SYStem.CoreStates.PRIOR()		103
SYStem.CoreStates.SMM()		103
SYStem.CoreStates.VMX()		104
SYStem.Option.MEMoryMODEL()		104
SYStem.Option.TOPOlogy()		104
SYStem.Option.TOPOlogy.SOCKETS()		104

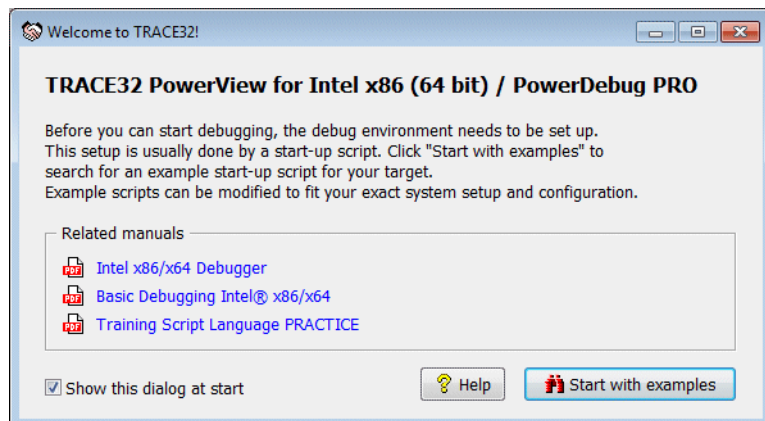
SYStem.ReadPDRH()	105
SYStem.ReadPDRL()	105
TrOnchip.IsAvailable()	105
TrOnchip.IsSet()	106
VMX()	106
VMX.Guest()	106
SYStem Trace Settings	107
Connectors	108
JTAG Connector	108
MIPI34 Connector	109
MIPI60-C Connector	110
MIPI60-Cv2 Connector	112
MIPI60-Q Connector	114

History

07-Jun-2022 New command: [JTAG.CONFIG.TckRun](#).

Welcome Dialog

The **Welcome to TRACE32!** dialog provides access to the most important manuals when TRACE32 is started the first time.



For the Intel® x86/x64 architecture the following manuals are listed:

- **Intel x86/x64 Debugger** is the manual you are currently reading. It provides all the information you need to establish a TRACE32 debug session for an Intel® x86/x64 chip.
- **“Training Script Language PRACTICE”** (training_practice.pdf) teaches you how to write, test and use a start-up script to establish a debug session.
- **“Training Basic SMP Debugging for Intel® x86/x64”** (training_debugger_x86.pdf) teaches you how to use the standard features of the TRACE32 debugger.

If you unchecked **Show this dialog at start** in the **Welcome to TRACE32!** dialog, you can use the following command to get access to this dialog:

```
WELCOME.view
```

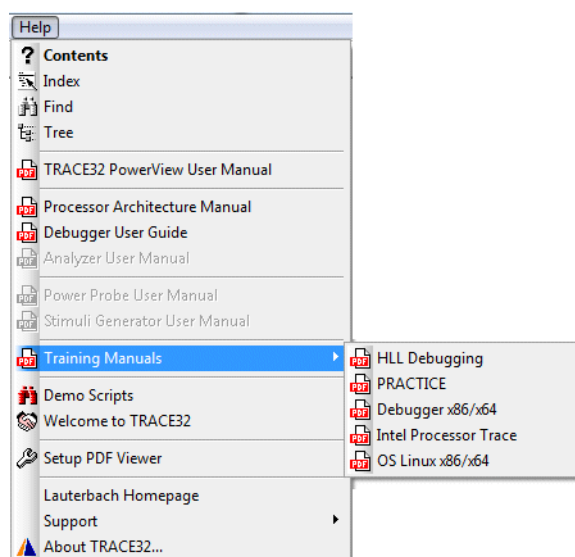
The following documents are also good starting points:

- **“Tools for Intel® x86/x64”** (tools_intel_x86.pdf) presents the delivery contents of the individual TRACE32 products and describes which steps are required to assemble a ready-to-use debug tool.
- **“Intel® Application Note for Server Setup”** (app_x86_server.pdf) explains the configuration of TRACE32 for Intel® Xeon® server systems.

The following documents are a good starting point for USB Debugging via Intel® DCI:

- **“Debugging via Intel® DCI User’s Guide”** (dci_intel_user.pdf).
- **“Debugging via USB User’s Guide”** (usbdebug_user.pdf).

The **Help** menu provides additionally access to all **Training Manuals**.

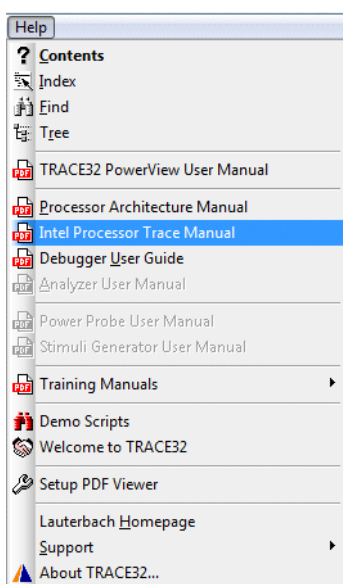


Beside the Processor Architecture Manual, which is the generic name for this manual within TRACE32 a number of training manuals are provided:

- **HLL Debugging** provides access to **“Training Source Level Debugging”** (training_hll.pdf) which mainly teaches you how to load the application program, how to display and format C-variables. If you are using C++ refer to **“Application Note C++ Debugging”** (app_cpp_debugging.pdf).
- **PRACTICE** provides access to **“Training Script Language PRACTICE”** (training_practice.pdf).
- **Debugger x86/x84** provides access to **“Training Basic SMP Debugging for Intel® x86/x64”** (training_debugger_x86.pdf).
- **Intel Processor Trace** provide access to **“Training Intel® Processor Tracing”** (training_ipt_trace.pdf). This manual teaches you how to configure the Intel® Processor Trace, how to record trace information, how to analyze and display the recorded information.
- **OS Linux x86/x64** provides access to **“Training Linux Debugging for Intel® x86/x64”** (training_rtos_linux_x86.pdf). This manual teaches you how to set up TRACE32 for Linux-aware debugging and how to use the Linux-awareness in a TRACE32 debug session.

All TRACE32 menus can be extended by the user. The following script shows a short example of how to add a manual to the TRACE32 **Help** menu.

```
MENU.ReProgram
(
  ADD
  MENU
  (
    POPUP "&Help"
    (
      AFTER "Processor Architecture Manual"
      MENUITEM "[:manual]Intel Processor Trace Manual" "HELP __ICRIPT_"
    )
  )
)
```



If you need the code of a manual (like __ICRIPT_ in the above example) please contact support@lauterbach.com.

Further Documents

The following manuals might also be of interest for Intel® x86/x64 users:

Trace manuals:

- **“Intel® Processor Trace”** (trace_intel_pt.pdf) provides configuration information, a command reference for the IPT command group and connector details.

UEFI-aware debugging:

- **“UEFI Awareness Manual BLDK”** (uefi_bldk.pdf) provides configuration information, a feature overview for the TRACE32 UEFI debugger for Intel® BLDK, an overview of all relevant EXTension commands and functions.
- **“UEFI Awareness Manual H2O”** (uefi_h2o.pdf) provides configuration information, a feature overview for the TRACE32 UEFI debugger for InsydeH2O, an overview of all relevant EXTension commands and functions.

OS-aware debugging:

- **“OS Awareness Manual Linux”** (rtos_linux_stop.pdf) provides configuration information, a feature overview for Linux stop-mode debugging, an overview of all relevant commands, functions and error messages.

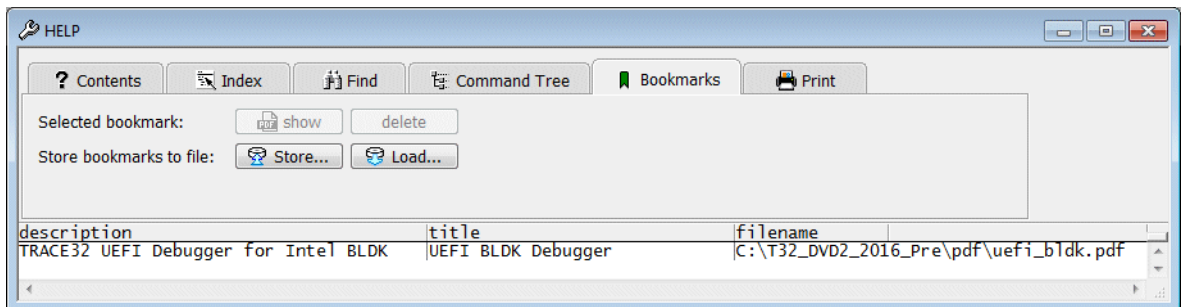
This manual is automatically added to the TRACE32 Help menu, when the TRACE32 Linux menu is programmed.

- **“OS Awareness Manual Windows Standard”** (rtos_windows.pdf) provides configuration information, a feature overview for standard windows debugging, an overview of all relevant commands and functions.

This manual is automatically added to the TRACE32 Help menu, when the TRACE32 MSWindows menu is programmed.

The following command allows to add manuals of interest to the Bookmarks tab of the TRACE32 online help:

```
HELP.Bookmark.ADD.file <file> <description> <title>
```



NOTE:	<p>To prevent debugger and target from damage it is recommended to connect or disconnect the debug cable only while the target power is OFF.</p> <p>Recommendation for the software start:</p> <ol style="list-style-type: none">1. Disconnect the debug cable from the target while the target power is off.2. Connect the host system, the TRACE32 hardware and the debug cable.3. Power ON the TRACE32 hardware.4. Start the TRACE32 software to load the debugger firmware.5. Connect the debug cable to the target.6. Switch the target power ON.7. Configure your debugger e.g. via a start-up script. <p>Power down:</p> <ol style="list-style-type: none">1. Switch off the target power.2. Disconnect the debug cable from the target.3. Close the TRACE32 software.4. Power OFF the TRACE32 hardware.
--------------	--

Quick Start

After starting TRACE32 PowerView for Intel® x86 (64 bit) please proceed as follows to debug your platform:

If you have been provided with a start-up script for your platform, first make sure that the platform is powered. Then simply execute the script as follows:

```
DO <file>
```

If you do not have a start-up script and want to debug an Intel® Xeon® server system platform, please refer to **“Intel® Application Note for Server Setup”** (app_x86_server.pdf) for how to do the necessary setup.

If you do not have a start-up script and want to debug an Intel® Atom™ or an Intel® Core™ i3/i5/i7 Client platform, please type (make sure the platform is powered first):

```
SYStem.DETECT TARGET
```

```
SYStem.Mode.Attach
```

```
Break
```

In most cases this setup is sufficient, and after the commands have been executed successfully it is possible to debug the target, including accessing memory and registers.

If for some reason the above is not successful, please follow the more detailed steps on the next page.

1. First TRACE32 must know which CPU/SOC your platform has. TRACE32 can normally detect this automatically as follows (make sure the platform is powered first):

```
SYStem.DETECT CPU
```

2. Such automatic detection is not supported for all possible platforms. If the automatic detection does not succeed, please select the CPU/SOC of the connected platform directly:

```
SYStem.CPU <cpu> | <soc>
```

3. If you are not sure about the name of the CPU/SOC you can open a window with a list of available names:

```
SYStem.CPU
```

Note that this is not a full list of all supported CPUs/SOCs. It only contains names of public, already launched products.

4. Next TRACE32 must know the number of cores/threads of the selected CPU/SOC. This step is required for Intel® Core™ i3/i5/i7 Client platforms, but can often be skipped for Intel® Atom™ platforms:

```
SYStem.DETECT CORES
```

NOTE: SYStem.DETECT TARGET (as used on the previous page) is basically SYStem.DETECT CPU followed by SYStem.DETECT CORES.

5. After the platform CPU/SOC has been detected/selected and the number of cores/threads have been detected (as necessary), further target-specific settings and options can be selected.

But in most cases the default values of other settings and options have automatically been set to the most useful values at this point. This means that in most cases it should now be possible to do basic debugging without any further initial configuration of TRACE32.

6. Attach to the target and enter debug mode.

```
SYStem.Mode.Attach
```

```
Break
```

The first command attaches the debugger to the running target. The second command stops the target and enters debug mode (often called *probe mode* for x86/x64 targets). After these commands are executed it is possible to access memory and registers.

A simple 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>.

```
RESet                ; Reset the TRACE32 software settings
WinCLEAR             ; Close all windows
SYStem.DETECT TARGET ; Detect platform CPU/SOC and cores/threads
SYStem.Mode.Attach   ; Attach to the running target
Break                ; Stop the target and enter debug mode
Register.view /SpotLight ; Open register and stack window *)
List.Mix              ; Open source code window          *)
```

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

Troubleshooting

No information available

FAQ

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

Tool Identification

The following TRACE32 functions allow you to check which Intel® x86/x64 specific TRACE32 tool are controlled by the TRACE32 software.

hardware.COMBIPROBE()	Returns TRUE if a TRACE32 CombiProbe is connected.
hardware.QUADPROBE()	Returns TRUE if a TRACE32 QuadProbe is connected.
ID.WHISKER(<int>)	Returns the identifier for the connected TRACE32 whisker cable.
ID.CABLE()	Returns 0x3836 if <i>Intel® x86/x64 XDP60 Debug Cable</i> is connected.

```
IF hardware.COMBIPROBE()
(
  IF ID.WHISKER(0)==0x10
  (
    PRINT "Connected Tool is CombiProbe MIPI60-C"
  )
  IF ID.WHISKER(0)==0x11
  (
    PRINT "Connected Tool is CombiProbe MIPI60-Cv2"
  )
  IF ID.WHISKER(0)==(0x2 | 0x09)
  (
    PRINT "Connected tool is CombiProbe DCI OOB"
  )
)
```

Onchip Breakpoints

The list below gives an overview of the availability and the usage of the **onchip breakpoints**. The following notations are used:

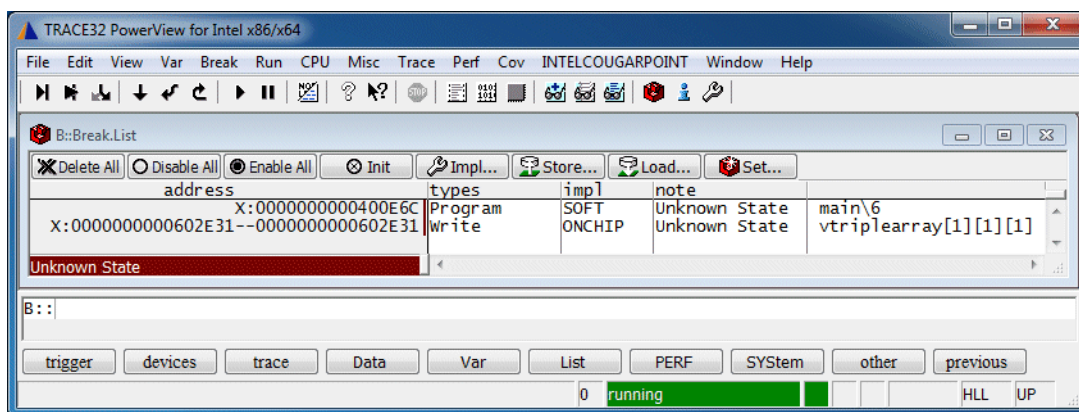
- **Onchip breakpoints:** Total amount of available onchip breakpoints.
- **Instruction breakpoints:** Number of onchip breakpoints that can be used to set Program breakpoints.
- **Read/Write breakpoints:** Number of onchip breakpoints that stop the program when a write or read/write to a certain address happens.
- **Data value breakpoint:** Number of onchip data breakpoints that stop the program when a specific data value is written to an address or when a specific data value is read from an address.

Family	Onchip Breakpoints	Instruction Breakpoints	Read/Write Breakpoint	Data Value Breakpoints
Intel® x86/x64	4	4 single address	4 Write or Read/Write single address or ranges up to 8 bytes (aligned)	—

A detailed introduction into the breakpoint handling can be found in [“Training Basic SMP Debugging for Intel® x86/x64”](#) (training_debugger_x86.pdf).

Breakpoints after Reset/Power Cycle

TRACE32 PowerView displays **Unknown State** in the **note** column of the **Break.List** window, if TRACE32 detects that the target is reset/re-powered and the cores immediately start the program execution. In this case it is likely that the breakpoint settings are cleared.



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)
ND	Protected Mode Data (32-bit)
NP	Protected Mode Program (32-bit)
AND	Absolute Protected Mode Data (32-bit)

Access Class	Description
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
QP	Real Big Mode Program
AQD	Absolute Real Big Mode Data
AQP	Absolute Real Big Mode Program

Access Class	Description
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
H	VMX Host Mode
CSS	Current value of CS

Access Class	Description
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 **eX**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   Q:0x1234:0x5678ABCD ; write to 32-bit Big Real Mode address
                                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 “<segment>:<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 0xdeadbeef	; Write IOSF sideband with opcode 0x06, ; port ID 0x08 and address 0x3C. ; (FID and BAR are both 0)
Data.In IOSF:0x0608:0xFF701234567890A B /long	; Read IOSF sideband with opcode 0x06, ; port ID 0x08, FID 0xFF, BAR 0x7 and ; 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:, QGD:, 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 Guest 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 Host 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 **SYSem.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**.

Selecting the Memory Model

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.

Platform Controller Hub (PCH)

All Intel® client/server platforms have a Platform Controller Hub (PCH) separate from the CPU. This section describes selection and usage of the PCH in TRACE32. How to select the PCH - and when it is necessary - depends on the usage model and the physical debug port.

There are three types of physical debug ports on Intel® client/server platforms:

1. Separate CPU debug port (no access to the PCH)
2. Separate PCH debug port (no access to the CPU)
3. Merged CPU/PCH debug port (access to both CPU and PCH)

Note that most Atom based platforms do not have a separate PCH. For such platforms, most information in this section does not apply.

In the following the three main usage models are described.

Debugging a CPU only

When debugging the CPU only, it is normally not required to select a PCH (but see [“PCH Selection for CPU Debug on a Merged Debug Port”](#), page 34), and it is advisable to set **SYStem.PCH NONE**. This is the default setting, so it only needs to be set if it has been changed earlier in the debug session.

```
SYStem.PCH NONE ; there is no PCH
SYStem.CPU <cpu>
SYStem.Option.MultiCoreWhiskers A0 ; select CPU whisker/TCK if needed
```

This setup is applicable to both a separate CPU debug port and a merged CPU/PCH debug port.

Debugging a PCH only

When debugging the PCH only, it is necessary to select NONE for the CPU.

```
SYStem.PCH <pch>
SYStem.CPU NONE ; there is no CPU
SYStem.Option.MultiCoreWhiskers D1 ; select PCH whisker/TCK if needed
```

This setup is applicable to both a separate PCH debug port and a merged CPU/PCH debug port.

Debugging a CPU and a PCH

Systems Using a Merged Debug Port

When debugging a system with the CPU and the PCH on a single merged debug port, it is possible to control both through a single instance of TRACE32. To do so, you must select both the CPU and the PCH:

```
SYStem.PCH <pch>
SYStem.CPU <cpu>
SYStem.Option.MultiCoreWhiskers A0 ; select CPU whisker/TCK if needed
; do not select any PCH whisker/TCK
; TCK1 of the CPU whisker is automatically chosen as PCH whisker
```

Note, with this combined CPU/PCH setup, low-level JTAG shifts can only be made to the CPU from the TRACE32 instance. To make low-level JTAG shifts to the PCH, the merged debug port system must be handled by TRACE32 as having two separate debug ports, see [“Systems Using Separate Debug Ports”](#), page 34.

Systems Using Separate Debug Ports

When debugging a system with the CPU and the PCH on separate debug ports, you must start two separate instances of TRACE32. The 1st instance must be set up as described in [“Debugging a CPU only”](#), page 33, and the 2nd instance must be set up as described in [“Debugging a PCH only”](#), page 33. Additionally, the 2nd instance must include the command:

```
SYStem.CONFIG.Slave off
```

PCH Selection for CPU Debug on a Merged Debug Port

There are several cases where it is necessary to select the PCH, even when the main intention is to debug only the CPU on a merged debug port:

- When using a DCI debug port instead of a dedicated Intel XDP60 or Intel MIPI60 debug port. This is because the DCI engine on the target is located in the PCH. For more information on DCI usage, see [“Debugging via Intel® DCI User’s Guide”](#) (dci_intel_user.pdf).
- When the target has a PMODE signal, as PMODE is controlled by the PCH. For more information, see [SYStem.Option.RESetDetection](#).
- When wanting to debug a core in the PCH concurrently with the main CPU. In this case the core in the PCH must be debugged as a slave in a multicore setup, see [“Slave Core Debugging”](#), page 34.

For all these cases the setup described in [“Systems Using a Merged Debug Port”](#), page 33, must be used.

Slave Core Debugging

Following the steps in the [Quick Start](#) Section, TRACE32 is set up and attached to an Intel® platform. With these steps, you can debug the main application CPU of the platform. All cores of the main CPU are typically handled in a single TRACE32 instance as an SMP setup. Beside the main application CPU cores, there could be other, often special purpose cores, integrated on the same platform. These are called *slave cores* in TRACE32 terminology. On Intel® client/server platforms, slave cores can also exist in the PCH. This section describes the steps needed to debug slave cores.

Each slave core requires a dedicated TRACE32 instance in addition to the instance for the main application CPU cores, i.e., an AMP multicore setup is needed. We call the TRACE32 instance that handles the main application CPU cores the *master debugger* and the TRACE32 instance that handles a slave core the *slave debugger*. TRACE32 also supports simultaneous debugging of multiple slave cores using multiple slave debuggers.

A slave cores is first characterized by its core type. Make sure the TRACE32 executable that matches the core type is installed, for example t32mx86 for a 32-bit x86 slave core. Then, if the an SoC integrates multiple slave cores of the same type, TRACE32 needs to know exactly which core your want to debug. This is done by assigning each slave core an SOC ID, which is unique to cores of the same type. If there is just one core of a certain core type, the corresponding ID is always 0.

In general, debugging a slave core consists of 3 steps introduced in the sections below.

Start Master Debugger

The master debugger must be started before any slave. In the master debugger, follow the [Quick Start](#) Section to set up the CPU type. If you need to debug a slave core in the PCH, make sure to configure the PCH type as well (see section [Platform Controller Hub](#)).

The master debugger must be set to at least **Prepare** mode before you can debugging any slave core. If the master debugger remains in **Down** or **NoDebug** mode, the debug port is disabled and no access to the slave core is possible.

Locating the Slave Core

To find out the ID for a slave core if there is more than one core of that type, use the following command in the **master** debugger.

```
SYStem.DETECT CLTapchain
```

A window will open with a column showing the SOC ID associated with each TAP of the platform, including both CPU and PCH. Please consult your Intel® support to know if a slave core exists behind a certain TAP and which core type it is.

Note that the SoC ID assignment is valid only for a given CPU and PCH configuration. If you change the CPU or PCH selection in the master debugger, the SOC ID may become different for the same slave core. In this case please re-run this command to get the up-to-date IDs. The following example shows the SOC ID assignment for difference configurations, considering a system that contains 2 x86 slave cores in the CPU and 1 in the PCH.

- CPU only

```
SYStem.CPU <cpu_name>
SYStem.PCH NONE
;SOC ID assignment:
;slave core 1 in CPU -> 0
;slave core 2 in CPU -> 1
;slave core in PCH   -> N/A
```

- PCH only

```
SYStem.CPU NONE
SYStem.PCH <pch_name>
;SOC ID assignment:
;slave core 1 in CPU -> N/A
;slave core 2 in CPU -> N/A
;slave core in PCH   -> 0
```

- CPU+PCH

```
SYStem.CPU <cpu_name>
SYStem.PCH <pch_name>
;SOC ID assignment:
;slave core 1 in CPU -> 0
;slave core 2 in CPU -> 1
;slave core in PCH    -> 2
```

Starting the Slave Debugger

In the next step, start the TRACE32 executable that matches the slave core type and do the following setup in the slave debugger:

```
SYStem.CPU <slave_core_type>
SYStem.Option.IntelSOC ON <soc_id>
;do other slave core specific settings if necessary
SYStem.Mode.Attach
```

The command **SYStem.Option.IntelSOC** is essential for the slave debugger setup. It indicates TRACE32 to locate the slave core in the master SoC according to its ID, instead of to treat it as a stand-alone core. If there is just one core in the SOC of the chosen core type, the SOC ID argument can be left out.

Note that slave cores in Intel® platforms are often protected by security features. A suitable security setting is often needed before such a core can be debugged. Please consult your Intel® support to check how to get access to the slave core you are interested in.

JTAG.CONFIG

Electrical characteristics of MIPI-60 debug signals

Using the **JTAG.CONFIG** command group, you can change electrical characteristics of MIPI-60 debug signals to account for target irregularities.

Availability of these commands is dependent on the Debug probe hardware in use.

Many of these commands allow specifying individual whiskers. Multiple whiskers may be selected. Specifying no whiskers indicates that the characteristics of all possible whiskers will be altered.

JTAG.CONFIG.DRiVer

Set slew rate of JTAG signals

Format:	JTAG.CONFIG.DRiVer.<signal> Fast Slow [/<whisker>]
<signal>:	all TCK TCK0 TCK1 TMS TDI nTRST nPREQ
<whisker>:	WhiskerA WhiskerB WhiskerC WhiskerD

Default: Fast.

Selects whether to use a series inductor to slow the slew rate of output signals.

all	Set rate for all relevant signals.
TCK	Set rate only for selected signal.
TCK0	
TCK1	
TMS	
TDI	
nTRST	
nPREQ	
FAST	Use direct drive of selected signals.
SLOW	Insert inductor on drive of selected signals to limit voltage change rate.

Format:	JTAG.CONFIG.PowerDownTriState ON OFF [/<whisker>]
<whisker>:	WhiskerA WhiskerB WhiskerC WhiskerD

Default: ON.

Enables or disables the automatic setting of all signals to tristate when a power down state of the target is detected.

JTAG.CONFIG.TckRun

Free-running TCK mode

[build 143356 - DVD 09/2022]

Format:	command.subcommand OFF TCK0 TCK1 [/<whisker>]
<whisker>:	WhiskerA WhiskerB WhiskerC WhiskerD

Default: OFF.

Enables free-running TCK mode for the respective TCK signal.

JTAG.CONFIG.TDOEdge

Select TCK edge

Format:	JTAG.CONFIG.TDOEdge Rising Falling [/<whisker>]
<whisker>:	WhiskerA WhiskerB WhiskerC WhiskerD

Default: RISING

Selects which edge of TCK signal is used for reading TDO.

Format:	JTAG.CONFIG.Voltage.HookThreshhold.<signal> <source> [/<whisker>] [ON OFF]
<signal>:	all Hook0 Hook6 Hook8 Hook9
<source>:	AUTO <voltage>
<whisker>:	WhiskerA WhiskerB WhiskerC WhiskerD

Default: 600mV

Sets voltage threshold to use for determining active state for selected Hook signals.

all	Set threshold for all Hook input signals.
Hook0 Hook6 Hook8 Hook9	Set threshold for selected Hook input signal only.
AUTO	Use threshold derived from reference voltage.
<voltage>	Value in volts to use as threshold.

Format:	JTAG.CONFIG.Voltage.REFerence <source>
<source>:	AUTO <voltage>

Default: AUTO.

Selects source to use for reference voltage.

- AUTOUse reference voltage supplied from target system.
- <voltage>Use specified value in volts as reference voltage.

JTAG.CONFIG.Voltage.THreshold

Set JTAG threshold voltages

Format:	JTAG.CONFIG.Voltage.THreshold.<signal> <source> [/<whisker>]
<signal>:	all TDO PRDY
<source>:	AUTO <voltage>
<whisker>:	WhiskerA WhiskerB WhiskerC WhiskerD

Default: AUTO.

Sets the voltage threshold to use for determining active state for selected JTAG signals.

- allSet threshold for TDO and PRDY.
- TDO
PRDYSet threshold for only selected signal.
- AUTOUse threshold derived from reference voltage.
- <voltage>Value in volts to use as threshold.

CPU specific SYStem.DETECT Commands

The **SYStem.DETECT** commands detect various configuration parameters of attached target board and apply these parameters to TRACE32.

For information about *architecture-independent* **SYStem.DETECT** commands, refer to “[General Commands Reference Guide S](#)” ([general_ref_s.pdf](#)).

For information about *architecture-specific* **SYStem.DETECT** commands, see command descriptions below.

SYStem.DETECT.CLTapchain

Show SOC IDs of SOC slave cores

Format:	SYStem.DETECT.CLTapchain
---------	---------------------------------

See “[Locating the Slave Core](#)”, page 35.

SYStem.DETECT.CORES

Detect core/thread number

Format:	SYStem.DETECT.CORES
---------	----------------------------

The command **SYStem.DETECT.CORES** detects the core number and the hyper thread status of the target board. The setup of TRACE32 is changed accordingly.

This command requires:

- Topology configuration (with **SYStem.Option.TOPOlogy**)
- Whisker configuration (with **SYStem.Option.MultiCoreWhiskers**)
- CPU configuration (with **SYStem.DETECT.CPU** or **SYStem.CPU**)

Format:	SYStem.DETECT.HyperThreads
---------	-----------------------------------

The command **SYStem.DETECT.HyperThreads** detects the hyper thread status of the CPU. The setup of TRACE32 is changed accordingly. This command is intended for analysis and not for target board setup.

This command requires:

- Topology configuration (with **SYStem.Option.TOPOlogy**)
- Whisker configuration (with **SYStem.Option.MultiCoreWhiskers**)
- CPU configuration (with **SYStem.DETECT.CPU** or **SYStem.CPU**)

Format:	SYStem.DETECT.TARGET [/<option>] [/Verbose]
<option>:	Auto CPUonly PCHonly

The command **SYStem.DETECT.TARGET** detects all required board setup parameters:

- The board topology
- The required whiskers for the detected topology
- The CPU and the PCH
- Merged debug port configuration
- The reset detection method
- The total core number for all CPUs of the target system
- The Hyperthread status

The setup of TRACE32 is changed accordingly.

<option>	For a description of the options, see SYStem.DETECT.TOPOlogy .
----------	--

Example: With **SYStem.DETECT.TARGET** the board setup simplifies to:

```
SYStem.DETECT TARGET
; target-specific configuration e.g.:
; SYStem.Option.RESetWaitTIME <milliseconds>
SYStem.Mode Attach
```

Format:

SYStem.DETECT.TOPOlogy [/<option>] [/Verbose]

<option>:

Auto | CPUonly | PCHonly

The command **SYStem.DETECT.TOPOlogy** detects:

- The board topology
- The required whiskers for the detected topology
- The CPU and the PCH
- Merged debug port configuration
- The reset detection method

The setup of TRACE32 is changed accordingly.

Auto	<ul style="list-style-type: none">• Default option• Detects and configures TRACE32 for CPU debugging.• Configures merged and non-merged debug ports.• Merged debug ports: enables CPU + PCH debugging (see “Systems Using a Merged Debug Port” (debugger_x86.pdf)).
CPUonly	<ul style="list-style-type: none">• Detects and configures TRACE32 for CPU only debugging.• Merged debug ports are <u>not</u> configured.
PCHonly	Detects and configures TRACE32 for PCH only debugging.
Verbose	Prints the detected topologies to the AREA window.

Format:	SYStem.CONFIG.state [/<tab>]
<tab>:	DebugPort Jtag COmponents USB

Opens the **SYStem.CONFIG.state** window, where you can view and modify most of the target configuration settings. The configuration settings tell the debugger how to communicate with the chip on the target board and how to access the on-chip debug and trace facilities in order to accomplish the debugger’s operations.

Alternatively, you can modify the target configuration settings via the [TRACE32 command line](#) with the **SYStem.CONFIG** commands. Note that the command line provides *additional* **SYStem.CONFIG** commands for settings that are *not* included in the **SYStem.CONFIG.state** window.

<tab>	Opens the SYStem.CONFIG.state window on the specified tab. For tab descriptions, see below.
DebugPort	Informs the debugger about the debug connector type and the communication protocol it shall use.
Jtag	Informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip. NOTE: In most cases, you do not need to make any settings on the Jtag tab of the SYStem.CONFIG.state window.
COmponents	Informs the debugger about the existence and interconnection of system trace modules.
USB	Informs the TRACE32 software about the configuration settings required for debugging via a USB cable. In addition, the icons on the USB tab display the configuration status. For descriptions of the commands on the USB tab, see “Debugging via USB User’s Guide” (usbdebug_user.pdf).

Format:	SYStem.CONFIG <parameter> SYStem.MultiCore <parameter> (deprecated)
<parameter>:	IRPRE <bits> IRPOST <bits> DRPRE <bits> DRPOST <bits> TriState [ON OFF] Slave [ON OFF] TAPState <state> TCKLevel <level>

Multicore Settings (daisy chain)

NOTE:	Almost no Intel x86/x64 targets require setting any of the four parameters IRPRE, IRPOST, DRPRE, DRPOST when using TRACE32 The configuration of the JTAG tap chain is handled automatically by the debugger when the appropriate CPU/SoC has been selected.
--------------	--

The four parameters IRPRE, IRPOST, DRPRE, DRPOST are used to inform the debugger of the TAP controller position in the JTAG chain if there is more than one core in the JTAG chain. This information is required for some CPUs before the debugger can be activated, e.g., by **SYStem.Mode.Attach**.

NOTE:	It is possible to use the command SYStem.DETECT.DaisyChain to probe the JTAG chain for the presence and positions of TAP controllers.
--------------	--

TriState has to be used if several debuggers are connected to a common JTAG port at the same time. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to tristate mode. Please note: nTRST must have a pull-up resistor on the target, TCK can have a pull-up or pull-down resistor, other trigger inputs need to be kept in inactive state.

DRPRE	(default: 0) <number> of TAPs in the JTAG chain between the core of interest and the TDO signal of the debugger. If each core in the system contributes only one TAP to the JTAG chain, DRPRE is the number of cores between the core of interest and the TDO signal of the debugger.
DRPOST	(default: 0) <number> of TAPs in the JTAG chain between the TDI signal of the debugger and the core of interest. If each core in the system contributes only one TAP to the JTAG chain, DRPOST is the number of cores between the TDI signal of the debugger and the core of interest.

IRPRE	(default: 0) <i><number></i> of instruction register bits in the JTAG chain between the core of interest and the TDO signal of the debugger. This is the sum of the instruction register length of all TAPs between the core of interest and the TDO signal of the debugger.
IRPOST	(default: 0) <i><number></i> of instruction register bits in the JTAG chain between the TDI signal and the core of interest. This is the sum of the instruction register lengths of all TAPs between the TDI signal of the debugger and the core of interest. See also Daisy-Chain Example .
TriState [ON OFF]	(default: OFF) If several debuggers share the same debug port, this option is required. The debugger switches to tristate mode after each debug port access. Then other debuggers can access the port.
Slave [ON OFF]	(default: OFF) If several debuggers share the same debug port, all except one must have this option active.
TAPState	(default: 7 = Select-DR-Scan) This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.
TCKLevel [0 1]	(default: 0) Level of TCK signal when all debuggers are tristated.

Daisy-Chain Example

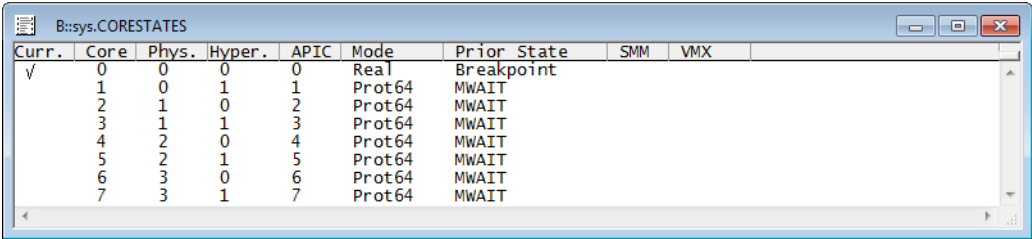
For a daisy-chain example, please refer to [“Daisy-Chain Example”](#) (general_ref_s.pdf).

TapStates

0	Exit2-DR
1	Exit1-DR
2	Shift-DR
3	Pause-DR
4	Select-IR-Scan
5	Update-DR
6	Capture-DR
7	Select-DR-Scan
8	Exit2-IR
9	Exit1-IR
10	Shift-IR
11	Pause-IR
12	Run-Test/Idle
13	Update-IR
14	Capture-IR
15	Test-Logic-Reset

Format:SYSystem.CORESTATES

This command opens an overview window showing mode, state and more for each core/hyperthread of the CPU. This information is updated every time the CPU is stopped.



Column	Description
Curr.	Currently selected core
Core	Core index as numbered by the debugger
Phys.	Index of physical core
Hyper.	Index of hyperthread of physical core
APIC	APIC ID of core
Mode	Current mode of core (see below)
Prior State	Indicates any special state the core is in (see below)
SMM	Shows if a core is in SMM mode (indicated by “Yes”)
VMX	Shows if a core is in VMX mode (indicated by “Host” or “Guest”)

NOTE:By double-clicking a line, the current core can be selected.

Possible values in the **Mode** column

Mode Value	Description
Inactive	The core is not accessible (e.g. due to a hang)
Real	The core is in real mode
Prot16	The core is in protected mode 16 bit
Prot32	The core is in protected mode 32 bit
Prot64	The core is in protected mode 64 bit

Possible values in the **Prior State** column

Prior State Value	Description
Breakpoint	The core stopped due to a breakpoint
Reset	The core stopped at the reset vector
SMM Entry	The core stopped at SMM Entry
SMM Exit	The core stopped at SMM Exit
VM Entry	The core stopped at VM Entry
VM Exit	The core stopped at VM Exit
Machine Check	The core stopped due to a Machine Check
INIT	The core stopped in the INIT state
HLT	The core stopped in the HLT state
Shutdown	The core stopped due to Shutdown
WFS	The core stopped in the Wait-for-SIPI state
MWAIT	The core stopped in an MWAIT state

Format:

SYStem.CPU <cpu> | <soc>

Selects the target CPU/SOC.

If no CPU/SOC name is provided, a window with a list of available names is opened. Note that this is not a full list of all supported CPUs/SOCs, it only contains names of public, already launched products.

After the CPU/SOC has been selected, further target-specific settings and options can be chosen.

SYStem.JtagClock

Define JTAG clock

Format:

SYStem.JtagClock <frequency>
SYStem.BdmClock <frequency> (deprecated)

Default: 5.0 MHz.

Selects the clock frequency of the JTAG debug interface communication.

Use the command **SYStem.DETECT.JtagClock** to experimentally detect the maximum possible JTAG clock frequency for a particular setup.

SYStem.LOCK

Tristate the JTAG port

Format:

SYStem.LOCK [ON | OFF]

Default: OFF.

When the system is locked, no access to the JTAG port will be performed by the debugger. While locked, the JTAG connector of the debugger is tristated. The intention of the **SYStem.LOCK** command is, for example, to give JTAG access to another tool. The process can also be automated, see **SYStem.CONFIG TriState**.

It must be ensured that the state of the JTAG state machine remains unchanged while the system is locked. To ensure correct hand-over, the options **SYStem.CONFIG TAPState** and **SYStem.CONFIG TCKLevel** must be set properly. They define the TAP state and TCK level which is selected when the debugger switches to tristate mode.

Format:	SYStem.MemAccess Denied StopAndGo
---------	--

Default: Denied.

Denied	No x86/x64 targets support non-intrusive real-time memory access.
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.

SYStem.Mode

Establish the communication with the target

Format:	SYStem.Mode <i><mode></i> SYStem.Attach (alias for SYStem.Mode Attach) SYStem.Down (alias for SYStem.Mode Down) SYStem.Up (alias for SYStem.Mode Up)
<i><mode></i> :	Down NoDebug Prepare Go Attach StandBy Up

Down	Down mode. Disconnects the debugger from the target. If the CPU is stopped in debug mode it is forced to leave and start running before the debugger is tristated.
NoDebug	Equivalent to Down .
Prepare	Resets JTAG. This must be used before doing raw JTAG shifting. Not used for normal debugging
Go	Connects the debugger and resets the target.

Attach	Connects the debugger to the running target. The state of the CPU remains unchanged.
StandBy	Standby mode. The debugger must be in this mode to handle scenarios where the target loses power and where the debugger must react when the power returns. The <i>default behavior</i> is to stop at the reset vector, rearm onchip breakpoints and set the CPU running again. The default behavior can be overwritten by using SYStem.Option.STandBYAttach , TrOnchip.Set.ColdRESet or TrOnchip.Set.BootStall .
Up	Connects the debugger, resets the target, enters debug mode and stops the CPU at the reset vector.

NOTE:

Some CPUs are not resettable via the JTAG interface, i.e., **Go** and/or **Up** might not work for all targets.

NOTE:

Standby functionality is not available for all CPUs.

SYStem.Option.Address32

Use 32 bit address display only

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

Default: AUTO.

This option only has an effect when in 64-bit mode. When the option is ON, a

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).
OFF	Display all addresses as 64-bit values.
AUTO	Number of displayed digits depends on address size.
NARROW	32-bit display with extendible address field.

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

SYStem.Option.BIGREALmode

Enable Big Real mode handling

Format:	SYStem.Option.BIGREALmode [ON OFF]
---------	--------------------------------------

Default: OFF.

NOTE:	The command takes effect only if the processor is in real mode.
-------	---

ON	SYStem.Option.BIGREALmode ON switches from the real mode to the Big Real mode. TRACE32 now works with 32-bit addresses (instead of 16 bit real-mode addresses). Opcodes are decoded as 16-bit real mode opcodes. The processor itself continues to be in real mode.
OFF	If the processor is in real mode and SYStem.Option.BIGREALmode is OFF , TRACE32 works only with real-mode addresses and 16-bit offsets.

The Big Real mode makes use of the fact that the hardware address registers in the core and the MMU of x386 and newer CPUs can hold 32 bit addresses, even if the CPU is in real mode. If **SYStem.Option.BIGREALmode** is enabled and the processor is in real mode, TRACE32 uses 32-bit addresses and 16-bit real-mode opcodes. The current PC is reported with the Big Real mode access class QP: instead of the real mode access class RP:

In Big Real mode, as in protected mode, the address extension (the first number in addresses like P:0x00A0:0x8000) specifies a **descriptor** and not a segment offset. This descriptor must be one of the 6 existing segment descriptors shown in the **MMU.view** window for CS, DS, ES, FS, GS or SS. Specifying a segment descriptor other than CS, DS, ES, FS, GS or SS will fail because in real mode there is no descriptor table walk.

In Big Real mode, the TRACE32 debugger address translation will add the code segment base CSB (for program addresses) or data segment base DSB (for data addresses) to the address offset. In contrast, if **SYStem.Option.BIGREAL** is disabled, CSB or DSB will be ignored during the debugger address translation and the segment offset will be multiplied by 16 and added to the address offset instead.

SYStem.Option.BranchSTEP

Enables branch stepping

Format:

SYStem.Option.BranchStep [ON | OFF]

Default: OFF.

If enabled, the debugger changes the behavior of normal single stepping to “single stepping on branches”: Only taken branches are visited when stepping.

SYStem.Option.BreakDELAY

Set max. break delay

Format:

SYStem.Option.BreakDELAY <ms>

Default: 500 ms.

Sets the max. break delay during which the debugger attempts to stop the CPU cores.

Increasing the break delay might help stop the CPU cores in certain power down scenarios, for example.

SYStem.Option.C0Hold

Hold CPU in C0 state

Format:

SYStem.Option.C0Hold [ON | OFF]

Default: OFF.

If enabled, the CPU is being held in the C0 state. In C0 the CPU is fully powered all the time. The CPU does not enter any power-saving modes and no peripherals are powered down.

Format:

SYStem.Option.IGnoreDEbugReDirections [ON | OFF]

Default: OFF.

When enabled, debug redirections are ignored by the debugger. This means that onchip breakpoints and most onchip triggers will not be functional.

This option is available for special handling if the target program needs to react to the debug redirections itself.

NOTE:

SW breakpoints in the debugger are still functional even if this option is enabled.

SYStem.Option.IGnoreSOC

Ignore SoC TAP chain structure

Format:

SYStem.Option.IGnoreSOC [ON | OFF]

Default: OFF.

When enabled, the debugger ignores all other TAPs in the SoC and considers just the plain x86 core. This option is typically used in early design phases to verify the x86 core in FPGA, for example.

To debug a plain x86 core, select an SoC that contains this type of core and enable this option.

SYStem.Option.IGnoreSWBPreDirections

Ignore SW BP redirections

Format:

SYStem.Option.IGnoreSWBPreDirections [ON | OFF]

Default: OFF.

When enabled, SW breakpoint redirections are ignored by the debugger. This means that SW breakpoints will not be functional.

This option is available for special handling if the target program needs to react to the SW breakpoint redirections itself.

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.

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.InstrSUBmitForcePHYSicalPRDY [ON | OFF]

Default: OFF.

If enabled, the debugger forces usage of the physical PRDY pin for checking completion of instruction submissions in Probe Mode even if [SYStem.Option.JTAGOnly](#) ON.

Format:SYStem.Option.InstrSUBmitIgnorePHYSicalPRDY [ON | OFF]

Default: OFF.

If enabled, the debugger ignores the physical PRDY pin for checking completion of instruction submissions in Probe Mode. A fixed delay is used instead, see [SYStem.Option.InstrSUBmitTimeout](#)

Format:	SYStem.Option.InstrSUBmitTimeout <us>
---------	--

Default: 2000 us.

Sets the timeout for instruction submission completion checking in Probe Mode. If no PRDY response has been registered within the time range, an error is issued in TRACE32. Note that a PRDY response can refer to either physical PRDY, virtual PRDY or RCM PRDY.

In the case that a mode is used which does not use PRDY at all (e.g., **SYStem.Option.JTAGDirectCPU ON**), the timeout value is used as a *fixed delay* after each instruction submission instead.

SYStem.Option.IntelSOC

Slave core is part of Intel® SoC

Format:	SYStem.Option.IntelSOC [ON OFF] [<soc_id>]
---------	---

Default: OFF.

Used for AMP multicore debugging to inform the slave debugger that the core is part of an Intel® SoC. When enabled, all IR and DR pre/post settings are handled automatically, no manual configuration is necessary.

The usage requires that the TRACE32 instance is slave in a multicore setup with a TRACE32 x86 master debugger. For more details, see **“Slave Core Debugging”**, page 34.

<soc_id>

An integer ID used by TRACE32 to identify a specific core in an SOC if there is more than one core of the same type. This ID is platform specific. For more details, see **“Slave Core Debugging”**, page 34.
Default: 0.

Format:

SYSystem.Option.JTAGDirectCPU [ON | OFF]

Default: CPU dependent.

NOTE:

This option is only relevant when **SYSystem.Option.JTAGOnly** ON

If enabled, the debugger talks directly to the CPU TAPs. If disabled, the debugger uses other JTAG-only methods to control the CPU (virtual PREQ/PRDY, RCM).

It is recommended to use the default setting unless special cases require otherwise.

Not all targets support **SYSystem.Option.JTAGDirectCPU** OFF

SYSystem.Option.JTAGOnly

Use only JTAG signals

Format:

SYSystem.Option.JTAGOnly [ON | OFF]

Default: CPU dependent.

If enabled, the debugger uses only the five JTAG signals (TCK,TMS,TDI,TDO,TRST) for controlling the target. If disabled, the debugger also uses extra non-JTAG signals (PREQ,PRDY, ...).

It is recommended to use the default setting unless special cases require otherwise.

Not all targets support **SYSystem.Option.JTAGOnly** OFF

SYSystem.Option.MACHINESPACES

Address extension for guest OSeS

Format:

SYSystem.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.MultiCoreWhiskers A0 | A1 | B0 | B1 | C0 | C1 | D0 | D1

Configures the required whiskers for server boards. It can more than one whisker be selected, e.g: **SYStem.Option.MultiCoreWhiskers A0 A1**.

A0	Whisker A, TCK0
A1	Whisker A, TCK1
B0	Whisker B, TCK0
B1	Whisker B, TCK1
C0	Whisker C, TCK0
C1	Whisker C, TCK1
D0	Whisker D, TCK0
D1	Whisker D, TCK1

SYStem.Option.NoDualcoreModule

Disable dualcore module support

Format:

SYStem.Option.NoDualcoreModule [ON | OFF]

Default: OFF if the CPU supports dual core modules. ON if the CPU does not support dual core modules.

If Dual Core Module support is disabled in the CPU, this option must be enabled.

It is required to use this option **before** attaching to the target, that is, before using the **SYStem.Mode.Attach** or **SYStem.Mode.Up** commands.

Format:

SYStem.Option.NoHyperThread [ON | OFF]

Default: OFF if the CPU supports hyper threading. ON if the CPU does not support hyper threading

If HyperThreading is disabled in the CPU, this option must be enabled.

It is required to use this option **before** attaching to the target, that is, before using the **SYStem.Mode.Attach** or **SYStem.Mode.Up** commands.

SYStem.Option.NoIPAdjust

Do not adjust IP at reset vector

Format:

SYStem.Option.NoIPAdjust [ON | OFF]

Default: OFF

Some CPUs, in some scenarios, do not initialize the instruction pointer correctly at the reset/INIT vector. To give a consistent user experience, the debugger by default forces the correct initialization in such cases.

An advanced user can disable the forced adjustment by setting this option to ON.

SYStem.Option.NoReBoot

Disable watchdog causing reboot

Format:

SYStem.Option.NoReBoot [ON | OFF]

Default: ON.

On some targets a watchdog timer causes a power cycle or a warm/cold reset if no forward progress is detected in the FW/BIOS.

To avoid such a target reboot (e.g., when stopping at the reset vector), if this option is enabled, the debugger will disable the watchdog timer if possible.

Format:

SYStem.Option.OSWakeupTIME *<milliseconds>*

Default 20ms.

Sets a wait time after a break, to wake up an operating system from sleep states.

SYStem.Option.PC10MODE

Wake up target from package C10

Format:

SYStem.Option.PC10MODE [ON | OFF]

Default: OFF.

Enables the flow to wake up target system from package C10 low power mode for debugging. This feature is target dependent and is not available for all Intel platforms.

SYStem.Option.PreserveDRX

Preserve DRx resources

Format:

SYStem.Option.PreserveDRX [ON | OFF]

Default: OFF.

If enabled, prevents other software from touching debug resources, including flags, debug registers (DRx), pending debug exceptions.

SYStem.Option.PreserveLBR

Preserve LBR resources

Format:

SYStem.Option.PreserveLBR [ON | OFF]

Default: OFF.

If enabled, prevents other software from touching LBR resources.

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

Default: OFF.

When enabled, the debugger does not carry out the state save/restore flow when entering/existing Probe Mode.

This option is only to be used for initial testing on slow emulation/simulation setups.

Format:	SYStem.Option.ProbeModeONDEmand [ON OFF]
---------	---

Default: OFF.

When enabled, the debugger carries out the state save/restore flow for each thread only on demand when in Probe Mode.

On demand means that only when registers, memory, etc. are being accessed through a given thread, will the state save/restore flow for that thread be carried out. Such a thread is then said to have been *visited*.

An important restriction compared to the normal save/restore handling applies: Onchip breakpoints can only be set for threads that have been visited. Due to this restriction, "Break.IMPLementation.Program SOFT" is automatically executed when enabling this option to enforce the use of SW breakpoints in the default case.

This option is meant for use for servers with many cores to speed up the total Probe Mode entry/exit time.

SYStem.Option.PWRCycleTime

Set power cycle time

Format:	SYStem.Option.PWRCycleTime <milliseconds>
---------	--

Default: CPU dependent (typical: 3000ms).

Sets the time between power off and power on for the command **SYStem.POWER CYCLE**.

SYStem.Option.PWROFFTime

Set power off assertion time

Format:	SYStem.Option.PWROFFTime <milliseconds>
---------	--

Default: CPU dependent (typical: 6000ms).

Sets the maximum assertion time for "Power Button" signal (hook2) to power off the system.

Format:

SYStem.Option.PWRONTime <milliseconds>

Default: CPU dependent (typical: 1000ms).

Sets the maximum assertion time for “Power Button” signal (hook2) to power on the system. If the system has not powered on, it will wait for PWRONWaitTime.

Format:

SYStem.Option.PWRONWaitTime <milliseconds>

Default: CPU dependent (typical: 3000ms).

Sets the maximum wait time after assertion of “Power Button” signal (hook2) to power on the system.

Format:

SYStem.Option.ReArmBreakPoints [ON | OFF]

Default: OFF.

When enabled, if a (warm) reset happens, the debugger attempts to stop at the reset vector, rearm onchip breakpoints and set the CPU running again.

[build 130480 - DVD 09/2021]

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.

Format:	SYStem.Option.RESetDELAY <milliseconds>
---------	--

Default: CPU dependent (typical: 200ms)

Sets the reset delay during which the debugger attempts to stop the CPU cores at the reset vector for the command **SYStem.Up** and the onchip trigger **TrOnchip.Set RESet ON**.

Increasing the break delay might help stop the CPU cores at the reset vector for certain platforms.

SYStem.Option.RESetDetection

Select reset detection source

Format:	SYStem.Option.RESetDetection OFF HOOK PMODE
---------	--

Default: See below

Selects the reset detection source used by TRACE32.

OFF	Ignore reset indications from the platform. This setting should be used if the platform does not have any reset indication or if the reset indication signal is not supported by TRACE32. This is the default if a PCH has been selected which has a reset indication signal not supported by TRACE32.
HOOK	Use the classical HOOK reset pin for reset indication. This is the default if no PCH has been selected.
PMODE	Use the PMODE signal for reset indication. Usage of PMODE as reset detection source requires the selection of a PCH with supported PMODE. This is the default value if a PCH has been selected which has a reset indication signal supported by TRACE32.

For more on selecting a PCH, see **“Platform Controller Hub (PCH)”**, page 32.

Format:

SYSystem.Option.RESetMode WARM | COLD

Default: WARM.

Used to select if a warm or a cold reset should happen when using **SYSystem.Mode Go** or **SYSystem.Up**.

This option does not have an effect for all targets.

Format:

SYSystem.Option.RESetTIME <milliseconds>

Default: CPU dependent (typical: 200ms)

Sets the reset assertion time for the commands **SYSystem.Mode Go** and **SYSystem.Up**.

The diagram shows three signals: **reset_o (hook 7)** (green), **reset_i (hook 6)** (orange), and **peq** (blue). The **reset_o** signal transitions from high to low at the start of the **RESetTIME** interval. The **reset_i** signal transitions from high to low at the start of the **RESetWaitTIME** interval. The **peq** signal transitions from high to low at the start of the **ResetDELAY** interval. The **WatchDogWaitTIME** interval is shown as a period where the **reset_i** signal is low. The **RESetTIME** interval is the time from the start of the **reset_o** signal transition to the start of the **reset_i** signal transition. The **RESetWaitTIME** interval is the time from the start of the **reset_i** signal transition to the start of the **ResetDELAY** interval. The **ResetDELAY** interval is the time from the start of the **ResetDELAY** interval to the start of the **WatchDogWaitTIME** interval. The **WatchDogWaitTIME** interval is the time from the start of the **WatchDogWaitTIME** interval to the end of the **WatchDogWaitTIME** interval.

Format:

SYSystem.Option.RESetWaitTIME <milliseconds>

Default: CPU dependent (typical: 200ms)

Sets the maximum wait time for the reset signal from the target after reset assertion with the commands **SYSystem.Mode Go** and **SYSystem.Up**.

If the target system has a **reset_i** signal (hook 6) and no reset input signal was detected during **RESetTime+RESetWaitTIME**, a warning will be displayed.

For the command **SYStem.Up** RESetWaitTIME controls the preq assertion:

- On systems without reset input (hook 6), preq will be asserted after RESetWaitTIME to halt the target at the reset vector
- On systems with reset input (hook 6), preq will be asserted as soon a reset input assertion from the target is detected. If no reset input assertion is detected, **SYStem.Up** aborts with an error.

See also: **SYStem.Option.RESetTIME** for timing diagram.

SYStem.Option.S0Hold

Hold SoC in S0 state

Format:

SYStem.Option.S0Hold [ON | OFF]

Default: OFF.

If enabled, the SoC is being held in the S0 state. The CPU is still free to use its C states. See also **SYStem.Option.C0Hold**. This option does not have an effect for all targets.

SYStem.Option.SOFTLONG

Use 32-bit access to set SW breakpoint

Format:

SYStem.Option.SOFTLONG [ON | OFF]

Default: OFF.

When enabled, this option forces the debugger to use only 32-bit memory access when patching code with the software breakpoint instruction.

NOTE:

MAP.BUS8 / BUS16 / BUS32 (used for restricting general memory access to the given width) does NOT influence the access width used for patching code with the software breakpoint instruction. So if MAP.BUS32 is used for a code memory range, this option must be enabled for SW breakpoints to work as well.

SYStem.Option.STandBYAttach

In standby mode, only attach to target

Format:

SYStem.Option.STandBYAttach [ON | OFF]

Default: ON.

When enabled, this option changes the behavior of the Standby mode (see [SYStem.Mode](#)): The debugger does not attempt to stop at the reset vector, but instead just attaches to the running CPU.

SYStem.Option.STandBYAttachDELAY

Delay after standby

Format:

SYStem.Option.STandBYAttachDELAY *<milliseconds>*

Default: 20ms.

When power returns in Standby mode (see [SYStem.Mode](#)) and [SYStem.Option.STandBYAttach](#) ON, this options sets the delay before the automatic [SYStem.Mode.Attach](#) is carried out.

SYStem.Option.STepINToEXC

Step into interrupt or exception handler

Format:

SYStem.Option.STepINToEXC [ON | OFF]

Default: OFF.

When enabled, this option allows the debugger to step into interrupt and exception handlers. This is not supported on older CPUs.

SYStem.Option.TOPology

Select server board topology

Format:

SYStem.Option.TOPology 1X1 | 1X2 | 2X1 | 2X2

Selects the server board topology.

1X1	1 CPU
1X2	2 CPUs (1 JTAG chain with 2 CPUs)
2X1	2 CPUs (2 JTAG chains with 1 CPU each)
2X2	4 CPUs (2 JTAG chains with 2 CPUs each)

Format:

SYSystem.Option.WatchDogWaitTIME *<milliseconds>*

Default: CPU dependent (typical: 2ms)

Sets the wait time for disabling the watch dog after a reset break with the command **SYSystem.Up** or the onchip trigger **TrOnchip.Set RESet ON**.

See also: **SYSystem.Option.RESetTIME** for timing diagram.

SYSystem.Option.WFSMemAccess

Allow WFS memory access

Format:

SYSystem.Option.WFSMemAccess [ON | OFF]

Default: OFF

When a core is in the Wait-For-SIPI (WFS) state, the debugger by default inhibits memory access through that particular core. This is because the instruction pointer might not be initialized correctly by the CPU. This could lead to illegal memory accesses which in the worst case could crash the platform.

An advanced user can allow WFS memory access by setting this option to ON.

SYSystem.Option.WHISKER

Select a whisker

Format:

SYSystem.Option.WHISKER **A0 | A1 | B0 | B1 | C0 | C1 | D0 | D1**

Selects a whisker on debug probes, which supports more than one JTAG chain (e.g. QuadProbe). It is mainly intended to temporarily select a whisker in the system mode “Down” for commands like **SYSystem.DETECT.CPU**.

A0	Whisker A, TCK0
A1	Whisker A, TCK1
B0	Whisker B, TCK0
B1	Whisker B, TCK1

C0	Whisker C, TCK0
C1	Whisker C, TCK1
D0	Whisker D, TCK0
D1	Whisker D, TCK1

The selected whisker may be changed by the debugger, when not in system mode “Down”.

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 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
```

SYStem.PCH

Select the target PCH

Format: **SYStem.PCH** <*pch*> | **NONE**

Default: NONE.

Selects the target PCH.

For more information, see “[Platform Controller Hub \(PCH\)](#)”, page 32.

SYStem.POWER

Control target power

Format: **SYStem.POWER** [ON | OFF | CYCLE]

If supported by the target, this command turns the target power ON (if off), OFF (if on), or does a power CYCLE (if on).

Format:

SYSystem.STALLPhase <stall_phase> | **NEXT**

Sets the target system into the selected stall phase (if supported). It should be used only if the system has already entered SoC/PCH Bootstall. Use **SYSystem.Mode.Attach** to leave the stall phases completely.

NEXT

Sets the system to the next stall phase supported by your target.

SYSystem.StuffInstruction

Submit instruction to CPU in probe mode

Format:

SYSystem.StuffInstruction <address> <mnemonic>

This command can be used to submit an assembler instruction (<mnemonic>) to the CPU in Probe Mode.

The <address> is a "dummy" address used only to decide the instruction size. This means that just either "O:0", "N:0" or "X:0" can be used as <address> to determine if 16, 32 or 64 bit instruction size, respectively.

SYSystem.StuffInstructionRead

Submit instruction and read

Format:

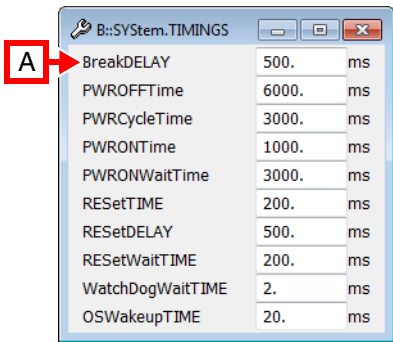
SYSystem.StuffInstructionRead <address> <mnemonic>

This command is like **SYSystem.StuffInstruction** but where PDRL and PDRH are read after issuing the instruction. The PDR values can be retrieved afterwards using the functions **SYSystem.ReadPDRL()** and **SYSystem.ReadPDRH()**.

Format:

SYStem.TIMINGS

Opens the **SYStem.TIMINGS** window, which gives an overview of all configurable timing settings related to target debugging. The values shown below are the default settings.



A To change any value, simply edit the window field directly or use the corresponding command, e.g., **SYStem.TIMINGS.BreakDELAY 1000.** or the equivalent **SYStem.Option.BreakDELAY 1000.**

See the corresponding **SYStem.Option** for descriptions of the timing settings, e.g., [SYStem.Option.BreakDELAY](#).

Command Groups for Special Registers

The command groups for special registers are documented in the `general_ref_<x>.pdf` manuals. For more information, click the blue hyperlinks.

AVX	Command group for the AVX registers (Advanced Vector Extension)
AVX512	Command group for the AVX512 registers (Advanced Vector Extension)
MMX	Command group for the MMX registers (MultiMedia eXtension)
SSE	Command group for the SSE registers (Streaming SIMD Extension)

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>	<p>Limit the address range displayed to either an address range or to addresses larger or equal to <address>.</p> <p>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.</p>
PageTable	<p>Displays the entries of an MMU translation table.</p> <ul style="list-style-type: none">• 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	<p>Displays the MMU translation table of the kernel.</p> <p>If specified with the MMU.FORMAT command, this command reads the MMU translation table of the kernel and displays its table entries.</p>

TaskPageTable <task_magic> <task_id> <task_name> <space_id>:0x0	<p>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.</p> <ul style="list-style-type: none">• 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>	<p>The following options are only available if SYSystem.Option.MACHINESPACES is set to ON.</p> <p>Dumps a page table of a virtual machine. The MACHINE option applies to PageTable and KernelPageTable and some <cpu_specific_tables>.</p> <p>The parameters <machine_magic>, <machine_id> and <machine_name> are displayed in the TASK.List.MACHINES window.</p>
Fulltranslation	<p>For page tables of guest machines both the intermediate address and the physical address is displayed in the MMU.DUMP window.</p> <p>The physical address is derived from a table walk using the guest's intermediate page table.</p>

CPU specific Tables in MMU.DUMP <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	<p>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.</p> <p>If the CPU's VMX mode is enabled, the IPT is identical to the EPT.</p> <p>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</p>

Examples for Page Tables in Virtualized Systems

Example 1:

```
System.Option.MACHINESPACES ON

; your code to load Hypervisor Awareness and define guest machine setup.

;                                     <machine_id>
MMU.DUMP.PageTable /MACHINE          2.

;                                     <machine_name>
MMU.DUMP.PageTable /MACHINE          "Dom0 "
```

Example 2:

```
SYStem.Option.MACHINESPACES ON

; your code to load Hypervisor Awareness and define guest machine setup.

;                                     <machine_name>:::<task_name>
MMU.DUMP.TaskPageTable                "Dom0::swapper"
```

Example 3:

```
SYStem.Option.MACHINESPACES ON

;your code to load Hypervisor Awareness and define guest machine setup.

;a) dumps the current guest page table of the current machine, showing
;   the intermediate addresses.
;   Without the option /Fulltranslation the column "physical" is hidden.
MMU.DUMP.PageTable 0x400000

;b) With the option /Fulltranslation the intermediate addresses
;   are translated to physical addresses and shown in column "physical"
MMU.DUMP.PageTable 0x400000 /Fulltranslation

;c) dumps the current page table of machine 2
;                                     <machine_id>
MMU.DUMP.PageTable  /MACHINE      2.          /Fulltranslation
```

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

<i><root></i>	The <i><root></i> 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.
<i><range></i> <i><address></i>	Limit the address range displayed to either an address range or to addresses larger or equal to <i><address></i> . For most table types, the arguments <i><range></i> or <i><address></i> 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. <ul style="list-style-type: none">• if <i><range></i> or <i><address></i> 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. <ul style="list-style-type: none">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. <ul style="list-style-type: none">• 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. <ul style="list-style-type: none">• 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 .
<i><option></i>	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	<p>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.</p> <p>If the CPU's VMX mode is enabled, the IPT is identical to the EPT.</p> <p>When the VMX mode is not enabled or not available on a CPU, an IPT can be specified using command MMU.FORMAT <i><ipt_base_address></i> /Intermediate</p>

MMU.Set

Set MMU register

Format:	MMU.Set <i><register></i> <i><value></i>
---------	---

Assigns *<value>* to an MMU *<register>*.

TrOnchip.PrintList

Print possible onchip triggers

Format:

TrOnchip.PrintList

Prints a list of Onchip Triggers available for this architecture. These are the legal values for use in the **TrOnchip.IsSet()** and **TrOnchip.IsAvailable()** functions.

TrOnchip.RESet

Reset settings to defaults

Format:

TrOnchip.RESet

Resets the TrOnchip settings to their default values.

TrOnchip.Set

Break on event

NOTE:

TRACE32 cannot activate the selected settings while the program execution is running.

TrOnchip.Set.BootStall

Enter bootstall

Format:

TrOnchip.Set.BootStall [ON | OFF]

Default: OFF.

If enabled, this trigger changes the default behavior of the Standby mode (see **SYSystem.Mode**) as follows:
After a power cycle, the debugger enters SoC/PCH Bootstall (if supported by the SoC/PCH).

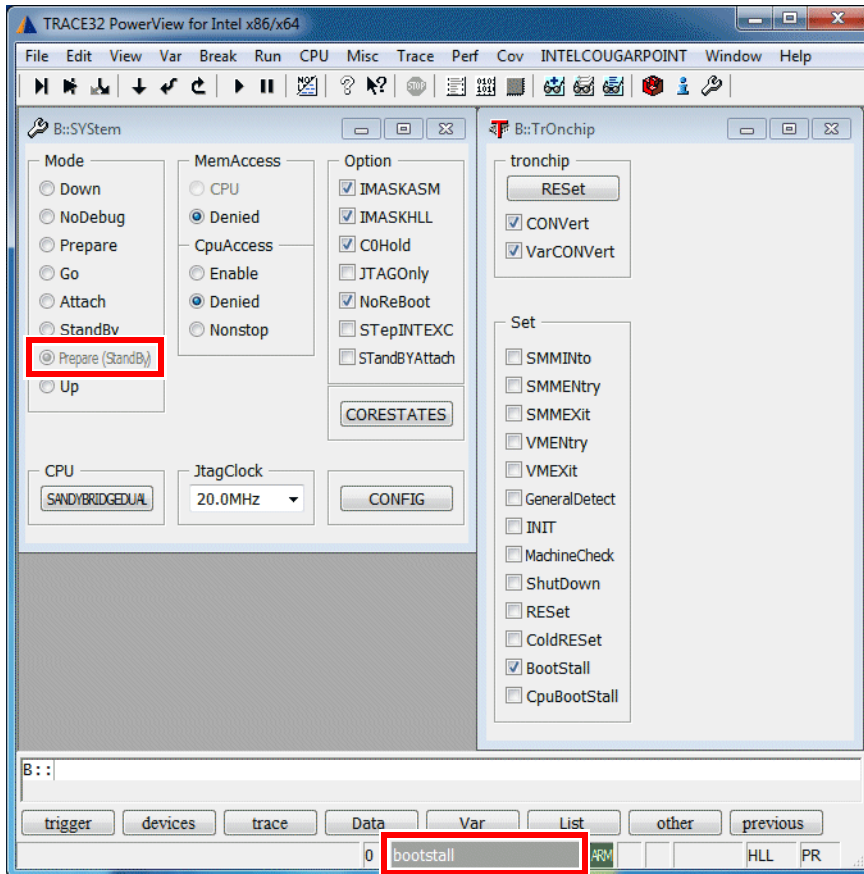
Example:

```
SYStem.Mode StandBy          ; Prepare TRACE32 to enter bootstall
                               ; mode on the next power cycle

TrOnchip.Set.BootStall ON

                               ; power off SoC/PCH

                               ; power on SoC/PCH
```



TRACE32 indicates successful Bootstall mode entry as follows:

- **bootstall** is displayed in the Debug field of the TRACE32 state line.
- The current mode of the debugger is **Prepare (StandBy)**.

After finishing operations in Bootstall mode, it can be left by one of the following commands (thereby returning the debugger to normal debug mode operation):

- **SYStem.Attach**: Leave bootstall and let the target boot normally from reset.
- **SYStem.Mode Go**: Leave bootstall and let the target boot normally from reset.
- **SYStem.Up**: Leave bootstall and stop the CPU at the reset vector.

Format:TrOnchip.Set.C6Exit [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped when a C6 Exit happens.

TrOnchip.Set.ColdRESet

Break on cold reset

Format:TrOnchip.Set.ColdRESet [ON | OFF]

Default: OFF.

If enabled, this trigger changes the default behavior of the Standby mode (see [SYStem.Mode](#)) as follows: After a power cycle, the debugger stops the CPU at the reset vector (if supported by the SoC/PCH).

Example:

```
SYStem.Mode StandBy          ; Prepare TRACE32 to stop the CPU
                             ; at the reset vector after a power
TrOnchip.Set ColdRESet ON    ; cycle
```

TrOnchip.Set.CpuBootStall

Enter CPU bootstall

Format:TrOnchip.Set.CpuBootStall [ON | OFF]

Default: OFF.

If enabled, this trigger changes the default behavior of the Standby mode (see [SYStem.Mode](#)) as follows: After a power cycle, the debugger enters CPU Bootstall (if supported by the CPU).

An example of how to use this feature is given in the description of the command [TrOnchip.Set.BootStall](#).

Format:TrOnchip.Set.ENCLU [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped when an ENCLU event happens.

Format:TrOnchip.Set.GeneralDetect [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped when a General Detect exception happens.

Format:TrOnchip.Set.INIT [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped when a processor INIT happens.

Format:TrOnchip.Set.MachineCheck [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped when a Machine Check exception happens.

Format:

TrOnchip.Set.RESet [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped at the reset vector when a target reset happens.

Format:

TrOnchip.Set.ShutDown [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped when a Shutdown occurs.

Format:

TrOnchip.Set.SMMENtry [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped each time SMM is entered.

Format:

TrOnchip.Set.SMMEXit [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped each time SMM is exited.

Format:

TrOnchip.Set.SMMINto [ON | OFF]

Default: OFF.

If enabled, if during an assembler single step an SMM interrupt happens, the debugger steps into the SMM handler. If disabled, the debugger steps over the SMM handler.

TrOnchip.Set.TraceHub

Enter/leave trace hub break

Format:

TrOnchip.Set.TraceHub [ON | OFF]

Default: OFF.

If enabled, the debugger enters Trace Hub Break next time a target reset happens. To leave Trace Hub Break, set to OFF.

Example:

```
TrOnchip.Set.TraceHub ON           ; prepare for Trace Hub Break
                                   ; cause target reset
                                   ; target enters Trace Hub Break
TrOnchip.Set.TraceHub OFF          ; leave Trace Hub Break
```

It is also possible to enter Trace Hub Break directly after leaving Bootstall mode (see [TrOnchip.Set.BootStall](#)). Simply set to ON before.

TrOnchip.Set.VMENtry

Break on VM entry

Format:

TrOnchip.Set.VMENtry [ON | OFF]

Default: OFF.

If enabled, the program execution is stopped each time a virtual machine VM entry happens.

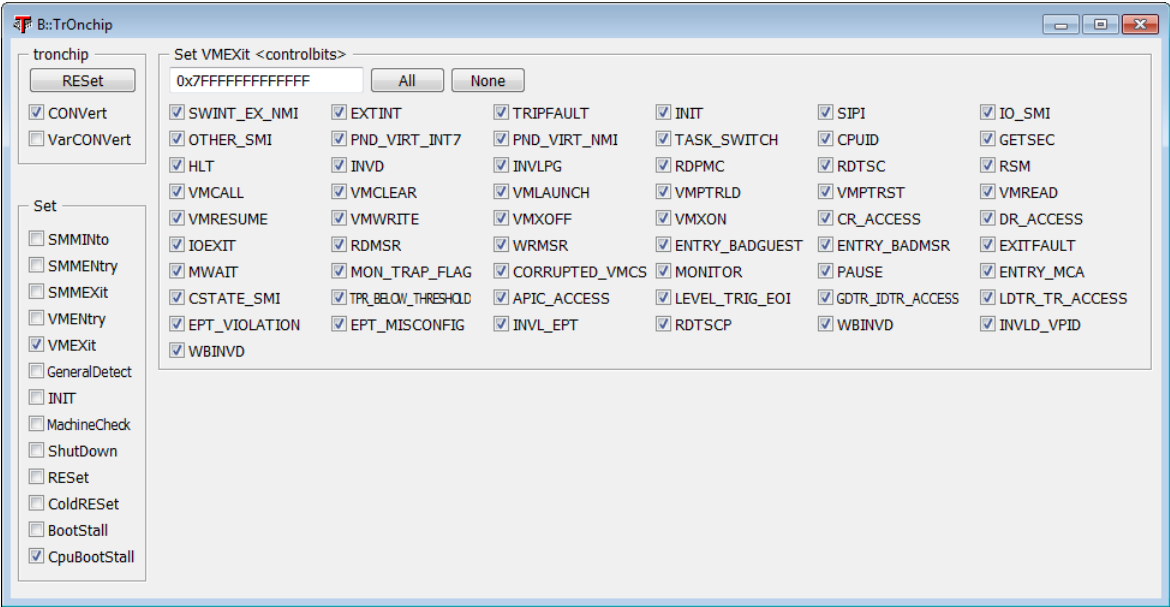
Format:	TrOnchip.Set.VMEXit [ON OFF All None <value> <controlbits>]
<value>:	<hexadecimal> <integer> <binary>
<controlbits>:	{<controlbit>}
<controlbit>:	SWINT_EXCEPTION_NMI EXTERNAL_INTERRUPT TRIPLE_FAULT INIT SIPI IO_SMI OTHER_SMI PND_VIRT_INTERRUPT PND_VIRT_NMI TASK_SWITCH CPUID GETSEC HLT INVD INVLPG RDPMC RDTSC RSM VMCALL VMCLEAR VMLAUNCH VMPTRLD VMPTRST VMREAD VMRESUME VMWRITE VMXOFF VMXON CR_ACCESS DR_ACCESS IOEXIT RDMSR WRMSR ENTRY_BADGUEST ENTRY_BADMSR EXITFAULT MWAIT MONITOR_TRAP_FLAG CORRUPTED_VMCS MONITOR PAUSE ENTRY_MCA CSTATE_SMI TPR_BELOW_THRESHOLD APIC_ACCESS LEVEL_TRIG_EOI GDTR_IDTR_ACCESS LDTR_TR_ACCESS EPT_VIOLATION EPT_MISCONFIG INVL_EPT RDTSCP VMXTIMER INVLD_VPID WBINVD

Default: OFF with all control bits disabled.

If enabled, the program execution is stopped each time a virtual machine VM exit event happens for the enabled control bits.

The **TrOnchip** control window is extended by the control bits:

- When VMEXit is set to ON.
- When control bits are set via the command line.



VMEXit check box:

ON The VM exit event is enabled and the program execution is stopped on a VM exit. If previously no control bit was enabled then all control bits are enabled.

OFF The VM exit event is disabled. The control bit remain unchanged.

All/None button:

All The VM exit event is enabled and all control bits are enabled.

None The VM exit event is disabled and all control bits are disabled.

Examples:

```
; Trigger VM exit event on signals TRIPLE_FAULT, VMWRITE and INIT
TrOnchip.Set.VMExit TRIPLE_FAULT VMWRITE INIT
; TrOnchip.Set.VMExit 0x200000C
; TrOnchip.Set.VMExit 0y1000000000000000000000001100

; Enable VM exit event on all control bit signals
TrOnchip.Set.VMExit All
;TrOnchip.Set.VMExit 0x7FFFFFFFFFFFFFFF

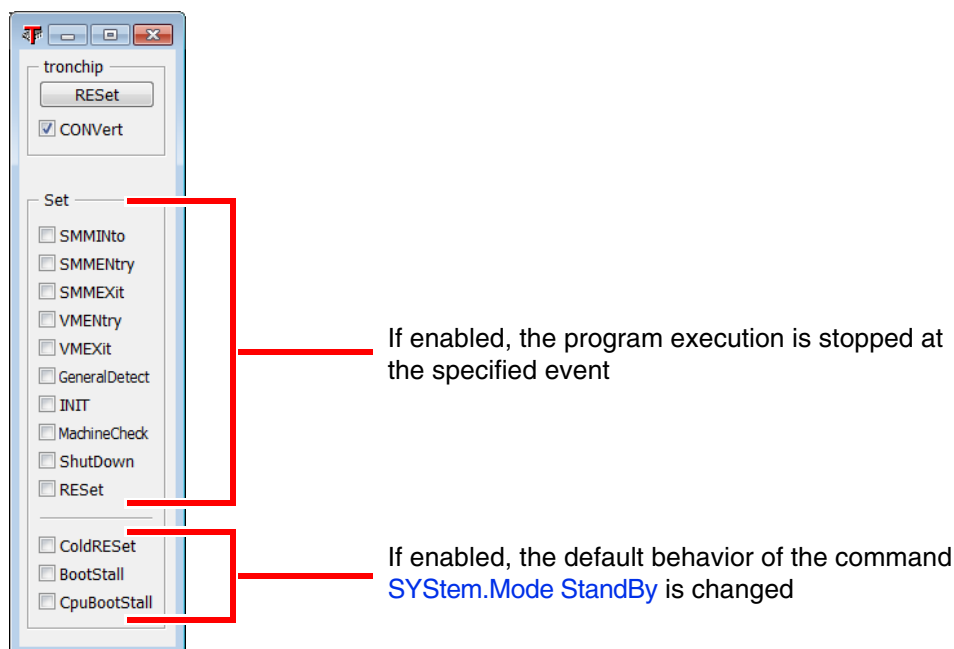
; Disable VM exit event and clear all control bits
TrOnchip.Set.VMExit None
```

TrOnchip.state

Display onchip trigger window

Format:	TrOnchip.state
---------	-----------------------

Displays the **TrOnchip** control window.



CPU specific Events for the ON and GLOBALON Command

TRACE32 can be programmed to detect CPU specific events and execute a user-defined *<action>* in response to the detected *<event>*. The user-defined action is a PRACTICE script (*.cmm).

The following commands and CPU specific events are available:

GLOBALON <i><event></i> [<i><action></i>]	Global event-controlled PRACTICE script execution. The event is detectable during an entire TRACE32 session.
ON <i><event></i> [<i><action></i>]	Event-controlled PRACTICE script execution. The event is detectable only by a particular PRACTICE script.

CPU specific <i><event></i>	Description
BOOTSTALL	The target entered Bootstall.
CPUBOOTSTALL	The target entered CPU Bootstall.
TRACEHUBBREAK	The target entered Trace Hub Break.
PBREAKRESET	The CPU stopped at the reset vector.
PBREAKVMENTRY	The CPU stopped due to a VM Entry event.
PBREAKVMEXIT	The CPU stopped due to a VM Exit event.
PBREAKSMMENTRY	The CPU stopped due to an SMM Entry event.
PBREAKSMMEXIT	The CPU stopped due to an SMM Exit event.
PBREAKGENERALDETECT	The CPU stopped due to a General Detect event.
PBREAKINIT	The CPU stopped due to an Init event.
PBREAKMACHINECHECK	The CPU stopped due to a Machine Check event.
PBREAKSHUTDOWN	The CPU stopped due to a Shutdown event.
PBREAKC6EXIT	The CPU stopped due to a C6 Exit event.
PBREAKENCLU	The CPU stopped due to an ENCLU event.

CPU specific BenchmarkCounter Commands

The **BMC** (**BenchMark Counter**) commands provide control and usage of the x86 performance monitoring capabilities. The benchmark counters can only be read while the target application is halted. Currently only the pre-defined architectural performance events are supported.

For information about *architecture-independent* **BMC** commands, refer to “**BMC**” (general_ref_b.pdf).

For information about *architecture-specific* **BMC** commands, see command descriptions below.

BMC.<counter>

Select BMC event to count

Format:	BMC.PMC0 <event> BMC.PMC1 <event>
<event>:	OFF UCC URC IR LLCR LLCM BIR BMR

Currently only the two generic benchmark counters PMC0 and PMC1 are supported. Each of these two counters can count one of the seven pre-defined architectural performance events. Please see the chapter on “Performance Monitoring” in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3” for details.

BMC.<counter>.COUNT

Select count mode for BMC

Format:	BMC.<counter>.COUNT <mode>
<mode>:	DUR EDGE

Default: DUR.

Selects the count mode for <counter>. In DURation mode, all cycles, where the selected event is enabled, are counted. In EDGE mode, only rising edges of the selected event are counted.

CPU specific Onchip Trace Commands

For information about *architecture-independent* **Onchip** commands, refer to “[Onchip Trace Commands](#)” (general_ref_o.pdf).

For information about *architecture-specific* **Onchip** commands, see command descriptions below.

Onchip.Buffer

Configure onchip trace source

Format:	Onchip.Buffer <i><item></i>
<i><item></i> :	LBR BTS IPT ITH BASE <i><base></i> SIZE <i><size></i> TOPA [ON OFF]

Provides control of the architectural x86 execution trace capabilities:

- LBR (Last Branch Records)
- BTS (Branch Trace Store)
- IPT (Intel® Processor Trace)

LBR	Chooses LBR as the trace source. LBR uses onchip registers to store the last 4, 8, or 16 (CPU dependent) taken branches/interrupts for each HW thread/core. The LBR feature is always available.
BTS	Chooses BTS as the trace source. BTS stores BTMs (Branch Trace Messages) in a user-defined area in target RAM for all HW threads/cores. <ul style="list-style-type: none">• BTS/BTM is not always available.• As a rule of thumb, BTS is usually available on iCore CPUs, whereas it is not functional on Atom CPUs.
IPT	Chooses IPT (Intel® Processor Trace) as the trace source. See “ Training Intel® Processor Tracing ” (training_ipt_trace.pdf).
ITH	Chooses ITH (Intel® Trace Hub) as trace source.
BASE <i><base></i>	Sets the base of the trace buffer to the linear address <i><base></i> .

SIZE <size>	<p>Sets the size of the trace buffer to <size> bytes.</p> <p>In the case of BTS, note that the buffer must be big enough to hold BTMs for all HW threads/cores.</p>
TOPA	<p>Enables/Disables Table Of Physical Addresses:</p> <ul style="list-style-type: none"> OFF: Trace data will be written to default memory location or address defined by Onchip.Buffer.Base. ON: Onchip.Buffer.Base points to a structure which defines the memory output area. Please consult Intel® Processor Trace documentation for more information. <p>Only applicable if Onchip.Buffer.IPT is selected!</p>

NOTE:	BTMs may not be observable on Intel Atom processor family processors that do not provide an externally visible system bus.
--------------	--

NOTE:	BTMs visibility is implementation specific and limited to systems with a front side bus (FSB). BTMs may not be visible to newer system link interfaces or a system bus that deviates from a traditional FSB.
--------------	--

Please see the chapter on “Debugging, Profiling Branches and Timestamp Counter” in the public “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3” for more details on LBR, BTM and BTS.

SYStem.CoreStates.APIC()

Syntax: **SYStem.CoreStates.APIC(<core>)**

Returns the APIC ID of the specified “virtual” core index <core>. This corresponds to the **APIC** column in the **SYStem.CORESTATES** window.

Parameter Type: [Decimal value](#).

Return Value Type: [Decimal value](#).

SYStem.CoreStates.HYPER()

Syntax: **SYStem.CoreStates.HYPER(<core>)**

Returns the hyper thread index of the specified “virtual” core index <core>. This corresponds to the **Hyper.** column in the **SYStem.CORESTATES** window.

Parameter Type: [Decimal value](#).

Return Value Type: [Decimal value](#).

SYStem.CoreStates.MODE()

Syntax: **SYStem.CoreStates.MODE(<core>)**

Returns the core mode of the specified “virtual” core index <core>. This corresponds to the **Mode** column in the **SYStem.CORESTATES** window.

Parameter Type: [Decimal value](#).

Return Value Type: [String](#).

SYStem.CoreStates.PHYS()

Syntax: **SYStem.CoreStates.PHYS(<core>)**

Returns the physical core index of the specified “virtual” core index <core>. This corresponds to the **Phys.** column in the **SYStem.CORESTATES** window.

Parameter Type: [Decimal value](#).

Return Value Type: [Decimal value](#).

SYStem.CoreStates.PRIOR()

Syntax: **SYStem.CoreStates.PRIOR(<core>)**

Returns the prior state of the specified “virtual” core index <core>. This corresponds to the **Prior State** column in the **SYStem.CORESTATES** window.

Parameter Type: [Decimal value](#).

Return Value Type: [String](#).

SYStem.CoreStates.SMM()

Syntax: **SYStem.CoreStates.SMM(<core>)**

Returns the SMM state of the specified “virtual” core index <core>. This corresponds to the **SMM** column in the **SYStem.CORESTATES** window.

Parameter Type: [Decimal value](#).

Return Value Type: [String](#).

SYStem.CoreStates.VMX()

Syntax: **SYStem.CoreStates.VMX(<core>)**

Returns the VMX mode of the specified “virtual” core index <core>. This corresponds to the **VMX** column in the **SYStem.CORESTATES** window.

Parameter Type: [Decimal value](#).

Return Value Type: [String](#).

SYStem.Option.MEMoryMODEL()

Syntax: **SYStem.Option.MEMoryMODEL()**

Returns the name of the currently enabled memory model.

Return Value Type: [String](#).

SYStem.Option.TOPOlogy()

Syntax: **SYStem.Option.TOPOlogy()**

Returns the name of the currently selected topology (e.g. “1X2”).

Return Value Type: [String](#).

SYStem.Option.TOPOlogy.SOCKETS()

Syntax: **SYStem.Option.TOPOlogy.SOCKETS()**

Returns the total number of CPU sockets for the currently selected topology.

Return Value Type: [Decimal value](#).

SYStem.ReadPDRH()

Syntax: **SYStem.ReadPDRH()**

Returns the PDRH value previously read with the command **SYStem.StuffInstructionRead**.

Return Value Type: [Hex value](#).

SYStem.ReadPDRL()

Syntax: **SYStem.ReadPDRL()**

Returns the PDRL value previously read with the command **SYStem.StuffInstructionRead**.

Return Value Type: [Hex value](#).

TrOnchip.IsAvailable()

[build 73501 - DVD 09/2016]

Syntax: **TrOnchip.IsAvailable("<trigger_name>")**

Returns TRUE if the named Onchip trigger is available for this architecture. A list of potential values for <trigger_name> can be generated with the **TrOnchip.PrintList** command.

Parameter Type: [String](#).

Return Value Type: [Boolean](#).

Examples:

```
PRINT TrOnchipIsAvailable("ColdRESet")
PRINT TrOnchipIsAvailable("CRES")
```

TrOnchip.IsSet()

[build 73501 - DVD 09/2016]

Syntax: **TrOnchip.IsSet("<trigger_name>")**

Returns TRUE if the named Onchip trigger is set. A list of potential values for <trigger_name> can be generated with the **TrOnchip.PrintList** command.

Parameter Type: [String](#).

Return Value Type: [Boolean](#).

Examples:

```
PRINT TrOnchip.IsSet ( "TraceHub" )  
PRINT TrOnchip.IsSet ( "TH" )
```

VMX()

[build 42354 - DVD 02/2013]

Syntax: **VMX()**

Returns TRUE if in VMX mode, FALSE otherwise.

Return Value Type: [Boolean](#).

VMX.Guest()

[build 42354 - DVD 02/2013]

Syntax: **VMX.Guest()**

Returns TRUE if in VMX guest mode, FALSE otherwise. This function is only applicable if **VMX()** returns TRUE.

Return Value Type: [Boolean](#).

SYStem Trace Settings

For information, see “[System Trace User’s Guide](#)” (trace_stm.pdf).

JTAG Connector

This JTAG connector is a 60-pin XDP connector.

Signal	Pin	Pin	Signal
GND	1	2	GND
PREQ-	3	4	N/C
PRDY-	5	6	N/C
GND	7	8	GND
N/C	9	10	N/C
N/C	11	12	N/C
GND	13	14	GND
N/C	15	16	N/C
N/C	17	18	N/C
GND	19	20	GND
N/C	21	22	N/C
N/C	23	24	N/C
GND	25	26	GND
N/C	27	28	N/C
N/C	29	30	N/C
GND	31	32	GND
N/C	33	34	N/C
N/C	35	36	N/C
GND	37	38	GND
PWRGOOD	39	40	N/C
N/C	41	42	N/C
VTREF	43	44	N/C
N/C	45	46	RESET-
N/C	47	48	DBR-
GND	49	50	GND
N/C	51	52	TDO
N/C	53	54	TRST-
N/C	55	56	TDI
TCK	57	58	TMS
GND	59	60	GND

Signal	Pin	Pin	Signal
VTREF DEBUG	1	2	TMS
GND	3	4	TCK
GND	5	6	TDO
N/C (KEY)	-	8	TDI
GND	9	10	N/C
GND	11	12	N/C
GND	13	14	N/C
GND	15	16	TRST-
GND	17	18	PREQ-
GND	19	20	PRDY-
GND	21	22	PTI 0 CLK
GND	23	24	PTI 0 DATA[0]
GND	25	26	PTI 0 DATA[1]
GND	27	28	PTI 0 DATA[2]
GND	29	30	PTI 0 DATA[3]
GND	31	32	N/C
GND	33	34	VTREF TRACE

MIPI60-C Connector

MIPI60 target pinout specified by Intel®.

Signal	Pin	Pin	Signal
VREF_DEBUG	1	2	TMS
TCK0	3	4	TDO
TDI	5	6	No Connect
HOOK[6]=Reset In	7	8	10 kOhm to GND
TRST_N	9	10	PREQ_N
PRDY_N	11	12	VTREF_TRACE
PTI_0_CLK	13	14	PTI_1_CLK
POD_PRESENT1_N	15	16	GND
POD_PRESENT2_N	17	18	PTI_1_DATA[0]
PTI_0_DATA[0]	19	20	PTI_1_DATA[1]
PTI_0_DATA[1]	21	22	PTI_1_DATA[2]
PTI_0_DATA[2]	23	24	PTI_1_DATA[3]
PTI_0_DATA[3]	25	26	PTI_1_DATA[4]/PTI_2_DATA[0]
PTI_0_DATA[4]	27	28	PTI_1_DATA[5]/PTI_2_DATA[1]
PTI_0_DATA[5]	29	30	PTI_1_DATA[6]/PTI_2_DATA[2]
PTI_0_DATA[6]	31	32	PTI_1_DATA[7]/PTI_2_DATA[3]
PTI_0_DATA[7]	33	34	HOOK[7]=Reset Out
PTI_0_DATA[8]/PTI_3_DATA[0]	35	36	HOOK[3]=Boot Stall
PTI_0_DATA[9]/PTI_3_DATA[1]	37	38	HOOK[2]=CPU Boot Stall
PTI_0_DATA[10]/PTI_3_DATA[2]	39	40	HOOK[1]=Power Button
PTI_0_DATA[11]/PTI_3_DATA[3]	41	42	HOOK[0]=PWRGOOD
PTI_0_DATA[12]/PTI_3_DATA[4]	43	44	HOOK[5]
PTI_0_DATA[13]/PTI_3_DATA[5]	45	46	HOOK[4]
PTI_0_DATA[14]/PTI_3_DATA[6]	47	48	I2C_SCL
PTI_0_DATA[15]/PTI_3_DATA[7]	49	50	I2C_SDA
TCK1	51	52	GND
TRIG_INOUT	53	54	DBG_UART_TX
TRIG_IN	55	56	DBG_UART_RX
GND	57	58	GND
PTI_3_CLK	59	60	PTI_2_CLK

Not all pins of the Intel® MIPI60 connector are connected to the CombiProbe Intel x86/x64 MIPI60-C. The connected pins are displayed with their name on a gray background in the picture below.

Signal	Pin	Pin	Signal
VREF_DEBUG	1	2	TMS
TCK0	3	4	TDO
TDI	5	6	Open Drain Reset Out
Reset In	7	8	No Connect
TRST_N	9	10	PREQ_N
PRDY_N	11	12	VREF_TRACE
PTI_0_CLK	13	14	PTI_1_CLK
GND	15	16	GND
No Connect	17	18	PTI_1_DATA[0]
PTI_0_DATA[0]	19	20	PTI_1_DATA[1]
PTI_0_DATA[1]	21	22	PTI_1_DATA[2]
PTI_0_DATA[2]	23	24	PTI_1_DATA[3]
PTI_0_DATA[3]	25	26	No Connect
PTI_0_DATA[4]	27	28	No Connect
PTI_0_DATA[5]	29	30	No Connect
PTI_0_DATA[6]	31	32	No Connect
PTI_0_DATA[7]	33	34	Reset Out
No Connect	35	36	Boot Stall
No Connect	37	38	CPU Boot Stall
No Connect	39	40	Power Button
No Connect	41	42	PWRGOOD
No Connect	43	44	No Connect
No Connect	45	46	No Connect
No Connect	47	48	I2C_SCL
No Connect	49	50	I2C_SDA
No Connect	51	52	No Connect
No Connect	53	54	DBG_UART_TX
No Connect	55	56	DBG_UART_RX
GND	57	58	GND
No Connect	59	60	No Connect

MIPI60-Cv2 Connector

Converged MIPI60 target pinout specified by Intel®.

Signal	Pin	Pin	Signal
VREF_DEBUG	1	2	TMS
TCK0	3	4	TDO
TDI	5	6	HOOK[7]=Reset Out
HOOK[6]=PMODE/Reset In	7	8	10 kOHM to GND
TRST_N	9	10	PREQ_N
PRDY_N	11	12	VTREF_TRACE
PTI_0_CLK	13	14	PTI_1_CLK
POD_PRESENT1_N	15	16	GND
POD_PRESENT2_N	17	18	PTI_1_DATA[0]
PTI_0_DATA[0]	19	20	PTI_1_DATA[1]
PTI_0_DATA[1]	21	22	PTI_1_DATA[2]
PTI_0_DATA[2]	23	24	PTI_1_DATA[3]
PTI_0_DATA[3]	25	26	PTI_1_DATA[4]/PTI_2_DATA[0]
PTI_0_DATA[4]	27	28	PTI_1_DATA[5]/PTI_2_DATA[1]
PTI_0_DATA[5]	29	30	PTI_1_DATA[6]/PTI_2_DATA[2]
PTI_0_DATA[6]	31	32	PTI_1_DATA[7]/PTI_2_DATA[3]
PTI_0_DATA[7]	33	34	No Connect
PTI_0_DATA[8]/PTI_3_DATA[0]	35	36	HOOK[3]=Boot Stall
PTI_0_DATA[9]/PTI_3_DATA[1]	37	38	HOOK[2]=CPU Boot Stall
PTI_0_DATA[10]/PTI_3_DATA[2]	39	40	HOOK[1]=Power Button
PTI_0_DATA[11]/PTI_3_DATA[3]	41	42	HOOK[0]=PWRGOOD
PTI_0_DATA[12]/PTI_3_DATA[4]	43	44	No Connect
PTI_0_DATA[13]/PTI_3_DATA[5]	45	46	No Connect
PTI_0_DATA[14]/PTI_3_DATA[6]	47	48	I2C_SCL
PTI_0_DATA[15]/PTI_3_DATA[7]	49	50	I2C_SDA
TCK1	51	52	No Connect
HOOK[9]	53	54	DBG_UART_TX
HOOK[8]	55	56	DBG_UART_RX
GND	57	58	GND
PTI_3_CLK	59	60	PTI_2_CLK

Not all pins of the Converged MIPI60 connector are connected to the CombiProbe Intel x86/x64 MIPI60-Cv2. The connected pins are displayed with their name on a gray background in the picture below.

Signal	Pin	Pin	Signal
VREF_DEBUG	1	2	TMS
TCK0	3	4	TDO
TDI	5	6	Reset Out
PMODE/Reset In	7	8	No Connect
TRST_N	9	10	PREQ_N
PRDY_N	11	12	VREF_TRACE
PTI_0_CLK	13	14	PTI_1_CLK
GND	15	16	GND
GND	17	18	PTI_1_DATA[0]
PTI_0_DATA[0]	19	20	PTI_1_DATA[1]
PTI_0_DATA[1]	21	22	PTI_1_DATA[2]
PTI_0_DATA[2]	23	24	PTI_1_DATA[3]
PTI_0_DATA[3]	25	26	No Connect
PTI_0_DATA[4]	27	28	No Connect
PTI_0_DATA[5]	29	30	No Connect
PTI_0_DATA[6]	31	32	No Connect
PTI_0_DATA[7]	33	34	No Connect
No Connect	35	36	Boot Stall
No Connect	37	38	CPU Boot Stall
No Connect	39	40	Power Button
No Connect	41	42	PWRGOOD
No Connect	43	44	No Connect
No Connect	45	46	No Connect
No Connect	47	48	I2C_SCL
No Connect	49	50	I2C_SDA
TCK1	51	52	reserved by TRACE32
HOOK[9]	53	54	DBG_UART_TX
HOOK[8]	55	56	DBG_UART_RX
GND	57	58	GND
No Connect	59	60	No Connect

MIPI60-Q Connector

Converged MIPI60 target pinout specified by Intel®.

Signal	Pin	Pin	Signal
VREF_DEBUG	1	2	TMS
TCK0	3	4	TDO
TDI	5	6	HOOK[7]=Reset Out
HOOK[6]=PMODE/Reset In	7	8	10 kOHM to GND
TRST_N	9	10	PREQ_N
PRDY_N	11	12	VTREF_TRACE
PTI_0_CLK	13	14	PTI_1_CLK
POD_PRESENT1_N	15	16	GND
POD_PRESENT2_N	17	18	PTI_1_DATA[0]
PTI_0_DATA[0]	19	20	PTI_1_DATA[1]
PTI_0_DATA[1]	21	22	PTI_1_DATA[2]
PTI_0_DATA[2]	23	24	PTI_1_DATA[3]
PTI_0_DATA[3]	25	26	PTI_1_DATA[4]/PTI_2_DATA[0]
PTI_0_DATA[4]	27	28	PTI_1_DATA[5]/PTI_2_DATA[1]
PTI_0_DATA[5]	29	30	PTI_1_DATA[6]/PTI_2_DATA[2]
PTI_0_DATA[6]	31	32	PTI_1_DATA[7]/PTI_2_DATA[3]
PTI_0_DATA[7]	33	34	No Connect
PTI_0_DATA[8]/PTI_3_DATA[0]	35	36	HOOK[3]=Boot Stall
PTI_0_DATA[9]/PTI_3_DATA[1]	37	38	HOOK[2]=CPU Boot Stall
PTI_0_DATA[10]/PTI_3_DATA[2]	39	40	HOOK[1]=Power Button
PTI_0_DATA[11]/PTI_3_DATA[3]	41	42	HOOK[0]=PWRGOOD
PTI_0_DATA[12]/PTI_3_DATA[4]	43	44	No Connect
PTI_0_DATA[13]/PTI_3_DATA[5]	45	46	No Connect
PTI_0_DATA[14]/PTI_3_DATA[6]	47	48	I2C_SCL
PTI_0_DATA[15]/PTI_3_DATA[7]	49	50	I2C_SDA
TCK1	51	52	No Connect
HOOK[9]	53	54	DBG_UART_TX
HOOK[8]	55	56	DBG_UART_RX
GND	57	58	GND
PTI_3_CLK	59	60	PTI_2_CLK

Not all pins of the Converged Intel® MIPI60 connector are connected to the Whisker MIPI60-Q for Quad-Probe x86/x64. The connected pins are displayed with their name on a gray background in the picture below.

Signal	Pin	Pin	Signal
VREF_DEBUG	1	2	TMS
TCK0	3	4	TDO
TDI	5	6	Reset Out
PMODE/Reset In	7	8	No Connect
TRST_N	9	10	PREQ_N
PRDY_N	11	12	VREF_TRACE
No Connect	13	14	No Connect
GND	15	16	GND
GND	17	18	No Connect
No Connect	19	20	No Connect
No Connect	21	22	No Connect
No Connect	23	24	No Connect
No Connect	25	26	No Connect
No Connect	27	28	No Connect
No Connect	29	30	No Connect
No Connect	31	32	No Connect
No Connect	33	34	No Connect
No Connect	35	36	Boot Stall
No Connect	37	38	CPU Boot Stall
No Connect	39	40	Power Button
No Connect	41	42	PWRGOOD
No Connect	43	44	No Connect
No Connect	45	46	No Connect
No Connect	47	48	I2C_SCL
No Connect	49	50	I2C_SDA
TCK1	51	52	reserved by TRACE32
HOOK[9]	53	54	DBG_UART_TX
HOOK[8]	55	56	DBG_UART_RX
GND	57	58	GND
No Connect	59	60	No Connect