LAUTERBACH
DEVELOPMENT TOOLS

General Commands Reference Guide R

MANUAL

General Commands Reference Guide R

General Commands Reference Guide R

Version 06-Jun-2024

History

24-Apr-23	The RunTime command group has been reworked. The new command is now a trace sink.
22-Aug-22	New command RunTime.WAIT.

Register — Processor registers

The **Register** command group is used to control and view the processor registers. In case of subprocessors (e.g., **FPU**) extra commands are provided.

Register values can be returned by the function **Register(***<register_name>***)**:

```
PRINT Register(D0)                    ; print the contents of the
                                      ; register D0 in the message line

Register.Set D1 Register(D0)          ; set register D1 to the contents
                                      ; of register D0
```

See also

- Register.Init
- Register.LOG
- Register.REFRESH
- Register.RELOAD
- Register.Set
- Register.StackTop
- Register.view
- Register()
- Register.LIST()
- Register.Valid()

▲ 'Release Information' in 'Legacy Release History'

Register.Init — Initialize the processor registers

Format:	**Register.Init** [**/***<option>*] **Register.RESet** (deprecated)
<option>:	**ForeGroundSet** | **BackGroundSet** | **SystemSet** | **TemporarySet** **CORE** *<number>* **REGSET** *<number>* | **Current** | **Previous** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* *<other_options>*

Sets the registers to the same state as after the processor reset. Registers which are undefined after **RESET** are set to zero.

<option>	For information about the options, see **Register.view**.

```
B::                             ; example for the debugger

…

SYStem.Up                       ; establish the communication between the
                                ; processor and the debugger

Register.Init                   ; initialize the general purpose registers

…
```

CPU	Behavior of **Register.Init**

ARC	`STATUS  <= 0x02000000`
	`STATUS32  <= 0x00000001`
	`DEBUG  <= 0x11000000`
	`IRQ_LV12  <= 0x00000002` (Resets any interrupt flags)
	`IENABLE  <= 0xffffffff`
	`SEMAPHORE <= 0x00000000`
	
	All other registers are set to zero.
	
	If **SYStem.Option.ResetDetection** is used with a semaphore bit (**Sem0**...**Sem3**), **Register.Init** sets the corresponding semaphore bit in the SEMAPHORE register.

ARM, Cortex, XScale	**ARM7/9/10/11, Cortex-A/R, XScale:** Rx = 0 SPSRx = 0x10 CPSR = 0xd3 (ARM7/9/10, XScale), 0x1d3 (ARM11, Cortex-A/R) R15 (PC) = 0, 0xffff0000 if high exception vectors selected **Cortex-M:** Rx = 0 R15 (PC) = [vector table base + 4] xPSR = 0x01000000 MSP = [vector table base + 0] PSP = 0 R13 (SP) = MSP or PSP depending on the mode
C166	The CP is set to 0xFC00. The sixteen registers R0 - R15 are set to 0x0. DPP0 = 0x0, DPP1 = 0x1, DPP2 = 0x2 and DPP3 = 0x3. Stack registers STKUN is set to 0xFC00 and STKOV is set to 0xFA00. The Stack Pointer SP is set to 0xFC00 The Instruction Pointer IP is set to zero. The Code Segment Pointer CSP and the VECSEG are set to the initial value after **SYStem.Mode Up**. All other registers are set to zero.
CEVA-X	MODA and MODA shadow register are set to 0x1E. All other registers are set to zero.

CPU	Behavior of **Register.Init**
DSP56K	**Family 56000 and 56100** The eight 16-bit modifier registers M[0-7] are set to 0xFFFF. This specifies linear arithmetic as the default type for address register update calculations.The Operating Mode Register (OMR) is set to the initial value after **SYStem.Mode Up**. Values of bits MA, MB and MC of the OMR register are preserved.The program counter is set to zero. All interrupts are masked by setting the Status Register (SR) to 0x300. **Family 56300 and 56720 Dualcore** The eight 24-bit modifier registers M[0-7] are set to 0xFFFFFF. This specifies linear arithmetic as the default type for address register update calculations. The Operating Mode Register (OMR) is set to the initial value after **SYStem.Mode Up**. Values of bits MA, MB, MC and MD of the OMR register are preserved. All interrupts are masked by setting Status Register (SR) to 0x300. The program counter is set to zero. **Family 56800 and 56800E** The eight 16-bit modifier registers M[0-7] are set to 0xFFFF. This specifies linear arithmetic as the default type for address register update calculations. The Operating Mode Register (OMR) is set to the initial value after **SYStem.Mode Up**. Values of bits MA and MB of the OMR register are preserved. All interrupts are masked by setting Status Register (SR) to 0x300. The program counter is set to zero.
HCS08	The Program Counter is set to the value read at 0xFFFE. The Stack Pointer SP is set to 0xFF and the CCR is set to 0x68. All other registers are set to zero.
HC11	The Program Counter is set to the value read at 0xFFFE. The Stack Pointer SP is set to a default value dependent on the derivative. The CCR is set to 0xD8. All other registers are set to zero.
HC12/S12/S12X	The Program Counter is set to the value read at 0xFFFE. The CCR is set to 0xD8. All other registers are set to zero.
Microblaze	All registers are set to zero.
MIPS32/MIPS64/NEC-VR	Program Counter, Status register and Config register are set to their initial values after reset (read during **SYStem.Mode Up**). PRID and Cause register are updated, all other registers are set to zero.
MMDSP	Sets all registers to their initial value after a reset. This is done via a **soft reset** of the core. **NOTE**: This may have effects besides updating the contents of architectural registers.
PowerPC	Program counter and MSR are set to their initial values after reset (read during **SYStem.Mode Up**). GPRs and SPR appearing in the Register window are set to zero.
PCP	All registers are set to zero.

CPU	Behavior of **Register.Init**
RISC-V	All registers are set to zero, with the following exceptions: The initial values for registers PC, and PRV are read from the CPU during **SYStem.Mode Up**. If this is not possible the following default values are assumed: PC = 0 PRV = M
Teak	MOD0 and MOD0S registers SATA bit is set. MOD1 and MOD1S registers CMD bit is set. MOD3 and MOD3S registers CREP, CPC and CCNTA bits are set. All other registers are set to zero.
Teaklite/Teaklite-II/Oak	All registers are set to zero.
Teaklite-III	MOD2 register SATA and SATP bits are set. All other registers are set to zero.
x86	EDX is set to a cpu specific value defining the family/model/stepping of the core if a **SYStem.Mode Up** has been executed at some point before, otherwise EDX is set to 0. EAX,EBX,ECX,ESI,EDI,ESP,EBP are set to 0. EIP is set to 0xFFF0 and EFLAGS to 2. CR0 is set to 0x60000010 and CR2-4 to 0. DR0-3 are set to 0. DR6 to 0xFFFF0FF0 and DR7 to 0x400. IDT and GDT: Base = 0 and Limit = 0xFFFF. LDT and TR: Selector = 0, Base = 0, Limit = 0xFFFF, Access = 0x82. CS: Selector = 0xF000, Base = 0xFFFF0000, Limit = 0xFFFF, Access = 0x93. DS,ES,FS,GS,SS: Selector = 0, Base = 0, Limit = 0xFFFF, Access = 0x93. **NOTE**: In a multicore system the above holds for the main bootstrap processor. For the other processors the following differences apply: EIP is set to 0x10000 and CR0 to 0x10.
TMS320	All registers except SSR, IER and TSR are set to zero.
TriCore	All registers are set to zero with the following exceptions: The initial values for registers PC, PSW, ISP and BTV are read from the CPU at **SYStem.Mode Up**. If this is not possible the following default values are assumed: PC=0xA0000020 (AURIX and later), 0xA0000000 (otherwise) PSW=0x00000B80 BTV=0xA0000100 ISP=0x00000100
XTENSA	All registers are set to zero.
ZSP	All registers are set to zero.

See also

- Register
- Register.view

Format:	**Register.LOG** [*<set>*] …[*/<option>*]
<set>:	**ALL**
<option>:	**AREA** *<name>*

Writes the selected registers to the **AREA** window whenever the program execution stops. The output can be redirected to any named **AREA** window. The output of any **AREA** windows can also be redirected to a file.

Example:

```
AREA.Create REG_LOG                     ; set up an AREA window named
                                        ; REG_LOG

AREA.OPEN REG_LOG regfile.log           ; write all outputs to the
                                        ; AREA window REG_LOG also to the
                                        ; file regfile.log

AREA.view REG_LOG                       ; display the AREA window REG_LOG
                                        ; in TRACE32

Register.LOG ALL /AREA REG_LOG          ; log the contents of all registers
                                        ; at every program stop to the
                                        ; AREA window REG_LOG

…

Register.LOG                            ; end the register logging

AREA.CLOSE REG_LOG                      ; close the file regfile.log
```

See also

- Register
- Register.view

Format:	**Register.REFRESH**

Forces the debugger to re-read all processor registers from the target and refresh the **Register.view** window.

Use this command, if your registers might have changed. The time base registers of the PowerPC processors for example change permanently even when the program execution is stopped.

NOTE:	Whenever the program execution is stopped, the **Register.view** window is refreshed automatically.

See also

■ Register ■ Register.view

Register.RELOAD Reload the compiler register settings

Format:	**Register.RELOAD**

Re-writes the initialization values of the last **Data.LOAD** command into the appropriate processor registers.

See also

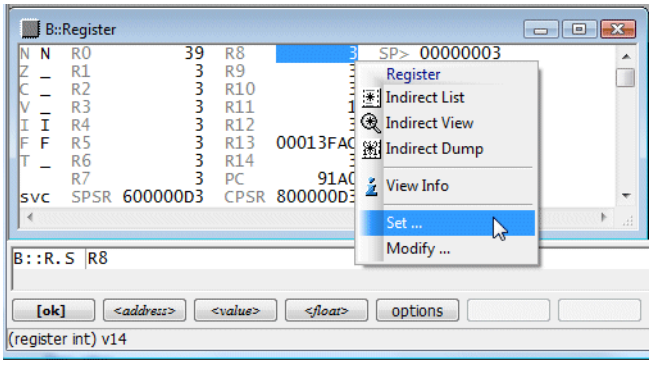
■ Register ■ Register.view

▲ 'Release Information' in 'Legacy Release History'

Format:	**Register.Set** *<register>* [*<value>*] [*/<option>*]
<register>:	**D0** | **D1** | **D2** | **D3** | …
<option>:	**TASK** *<task_magic>* | *<task_id>* | *<task_name>* *<other_options>*

Sets *<register>* to the specified *<value>*.

The **Register.Set** command is also invoked by a double-click to the register contents or by choosing **Set …** in the **Register** popup menu.



<option>	For information about the options, see **Register.view**.

Examples:

```
Register.Set PC start             ; set the Program Counter to the label
                                  ; start

Register.Set D0 Register(D0)+1    ; increment register contents
```

See also

■ Register ■ Register.view ❑ Register()

▲ 'Release Information' in 'Legacy Release History'

Format: **Register.StackTop** *<address>*

Limits the display of the stack in the register window. When the stack is below or equal to this address its contents will not be displayed.

Example:

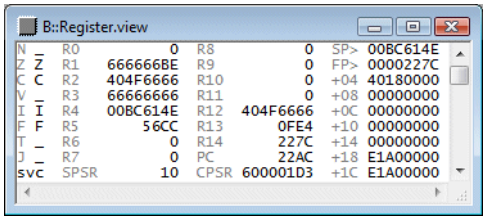
```
Register.StackTop 0x14000              ; limit the stack display in the
                                       ; register window to address
                                       ; 0x14000
```

See also

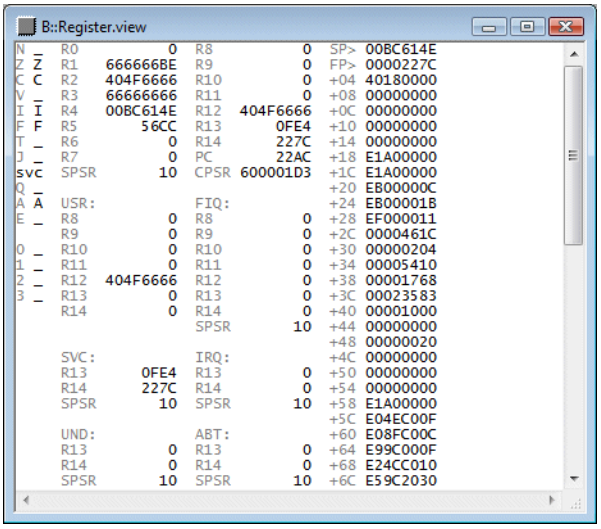
■ Register ■ Register.view ❑ Register()

Format:	**Register.view** [**/***<option>*]
<display_option>:	**SpotLight** **Stack**
<context_option>:	**CORE** *<number>* **REGSET** *<number>* | **Current** | **Previous** **SystemSet** | **TemporarySet** **TASK** *<task_magic>* | *<task_id>***.** | "*<task_name>*" **MACHINE** *<machine_id>* [**/VCPU** *<vcpu_id>*]

Display the general purpose registers. For some CPUs additional information is displayed if the **Register.view** window is dragged to its full size.



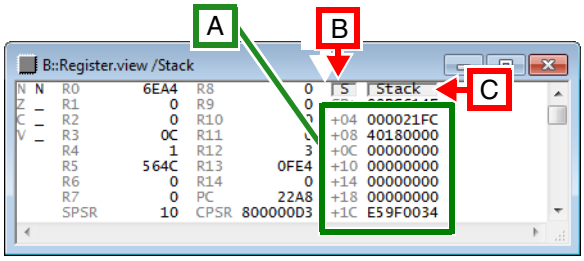
Default display



Full display

SpotLight	Highlight changed registers. Registers changed by the last program run/single step are marked in dark red. Registers changed by the second to last program run/single step are marked a little bit lighter. This works up to a level of 4.

```
Register.view /SpotLight
```

Stack	With **Stack**: The stack display [**A**] is shown. Without **Stack**: The stack display is hidden.  [**B**] Toggle between: • S = Addresses relative to the stack pointer • F = Addresses relative to the frame pointer • C = Addresses relative to the canonical frame address [**C**] Clicking the **Stack** button in the **Register.view** window lets you show/hide the stack display on the fly.
SystemSet	The TRACE32-internal register set is a temporary buffer within TRACE32. This buffer is used to hold a copy of the CPU registers. Commands like **Register.Copy**, **Register.SWAP** are using this TRACE32-internal register set.

```
Register.COPY                   ; copy general purpose registers into the
                                ; TRACE32-internal register set

Go                              ; start program execution

Break                           ; stop program execution

Register.view /SystemSet        ; display TRACE32-internal register set
```

TemporarySet	(diagnosis purpose only)

CORE *<number>*	(SMP debugging only) The TRACE32 PowerView GUI displays the context (registers, cache, memory …) of the currently selected core if SMP debugging is performed. The option **CORE** allows to display the register set of another core.

```
CORE.select CORE 1                      ; advise TRACE32 to display the
                                        ; context of core 1

Register.view                           ; display the registers of the
                                        ; current context

Register.view /CORE 0                   ; display the registers for core 0
```

REGSET *<number>*	(processor-specific) **MIPS architecture:** Display the specified shadow register set. **SH2A architecture:** Display the specified register bank.

TASK *<task_magic>*, etc.	Display the register set of the specified task. See also **"What to know about the Task Parameters"** (general_ref_t.pdf).

```
Register.view /TASK 0x41498             ; <task_magic>

                                        ; <task_id>

Register.view /TASK "thread 0"          ; <task_name>
```

MACHINE *<machine_id>*	Display the current register set for the specified machine (only available with **SYStem.Option.MACHINESPACES ON**). See also **"What to know about the Machine Parameters"** (general_ref_t.pdf).
VCPU *<vcpu_id>*	Display the current register set for the specified VCPU (only available with **SYStem.Option.MACHINESPACES ON**).

See also

- Register
- Register.Init
- Register.LOG
- Register.REFRESH
- Register.RELOAD
- Register.Set
- Register.StackTop
- Go.direct
- Register()
- Register.LIST()
- Register.Valid()

▲ 'Release Information' in 'Legacy Release History'

Resets all commands.

RESet — Reset all commands

Format:	**RESet**

All commands of the debugger are reset to their default state.

See also

- SYStem.RESet

RTP.CLEAR — Clear tracebuffer

See command **RTP.CLEAR** in 'RAM Trace Port' (trace_rtp.pdf, page 6).

RTP.DirectDataMode — Simple trace mode

See command **RTP.DirectDataMode** in 'RAM Trace Port' (trace_rtp.pdf, page 7).

RTP.DirectDataMode.Mode — Direct data mode read/write

See command **RTP.DirectDataMode.Mode** in 'RAM Trace Port' (trace_rtp.pdf, page 7).

RTP.HaltOnOverflow — Halt system on RTP FIFO overflow

See command **RTP.HaltOnOverflow** in 'RAM Trace Port' (trace_rtp.pdf, page 8).

RTP.Mode — Select the trace mode

See command **RTP.Mode** in 'RAM Trace Port' (trace_rtp.pdf, page 8).

RTP.OFF — Disables the RTP module

See command **RTP.OFF** in 'RAM Trace Port' (trace_rtp.pdf, page 8).

RTP.ON — Activates the RTP module

See command **RTP.ON** in 'RAM Trace Port' (trace_rtp.pdf, page 9).

RTP.PortClock — Configure RTPCLK

See command **RTP.PortClock** in 'RAM Trace Port' (trace_rtp.pdf, page 9).

RTP.PortSize — Size of RTP data port

See command **RTP.PortSize** in 'RAM Trace Port' (trace_rtp.pdf, page 9).

RTP.RESet — Resets RTP settings

See command **RTP.RESet** in 'RAM Trace Port' (trace_rtp.pdf, page 10).

RTP.state — Display RTP setup

See command **RTP.state** in 'RAM Trace Port' (trace_rtp.pdf, page 10).

RTP.TraceMode — Complex trace mode

See command **RTP.TraceMode** in 'RAM Trace Port' (trace_rtp.pdf, page 11).

RTP.TraceMode.RAM<x>.SECTion<y> — Configures a trace region

See command **RTP.TraceMode.RAM<x>.SECTion<y>** in 'RAM Trace Port' (trace_rtp.pdf, page 11).

RTP.TraceMode.TraceExclude — Invert all trace regions

See command **RTP.TraceMode.TraceExclude** in 'RAM Trace Port' (trace_rtp.pdf, page 12).

RTS

Real-time profiling (RTS)

See also

- RTS.COMMAND
- RTS.LOAD
- RTS.ON
- RTS.RESet
- RTS.StopOnBadaddress
- RTS.StopOnFifofull
- RTS.StopOnUnknowntask
- RTS.TrackData
- RTS.TriggerConnect
- RTS.TriggerOnRead
- RTS.TriggerOnWTM
- RTS.TriggerWaitForAck
- RTS.UNLOAD
- ISTATistic
- RTS.Init
- RTS.OFF
- RTS.PROfile
- RTS.state
- RTS.StopOnError
- RTS.StopOnNoaccesstocode
- RTS.TImeMode
- RTS.TRIGGERACK
- RTS.TriggerOnExecute
- RTS.TriggerOnWrite
- RTS.TriggerSlave
- RTS.UnknownData
- COVerage

▲ 'RTS Functions' in 'General Function Reference'
▲ 'Release Information' in 'Legacy Release History'

Overview RTS

Real-time profiling (RTS) is a trace mode that allows to process the trace data **while the trace information is being recorded**. RTS requires that the program code that is needed to decode the trace raw data is located in the TRACE32 virtual memory (VM:). RTS can therefore be used only for static program code. However, it supports code overlays and zone spaces.

Direct processing is possible for the following command groups:

- **COVerage.List***
- **ISTATistic.List***
- **RTS.PROfile**

By default the trace data is discarded after processing, but enabling the **Trace.Mode STREAM** allows to obtain the trace data for a later analysis.

RTS is currently supported for the following processor architecture/trace protocols:

- ARM ETMv3 and ARM ETMv4
- Nexus for MPC5xxx and QorIQ
- TriCore MCDS

RTS can only be used if the average data rate at the trace port does not exceed the maximum transmission rate of the host interface in use. Peak loads at the trace port are intercepted by the trace memory, which can be considered to be operating as a large FIFO.

Not all **chip timestamp modes** are decodable by RTS.

Example 1: Live update of code coverage results for Nexus MPC5xxx, trace information is discarded after processing.

```
…

; enable Nexus Indirect Branch History Messages to get compact
; trace data
NEXUS.HTM ON

; load executable to target and to TRACE32 Virtual Memory
Data.LOAD.Elf demo.x /PlusVM

; switch RTS processing to on
RTS.ON

; specify stops on errors if required (just as example)
; RTS.StopOnNoaccesstocode ON

; select the required coverage metric (just as example)
; COVerage.Option.SourceMetric Decision

; display code coverage for HLL functions
COVerage.ListFunc

; display source code with coverage tagging
List E:0x8D04 /COVerage

Go

Break

…

; switch RTS processing to off
RTS.OFF

; save coverage results to file
COVerage.SAVE demo.acd

…
```

Example 2: Live update of code coverage results for Nexus MPC5xxx, trace information is obtained for a later analysis:

```
…

; enable Nexus Indirect Branch History Messages to get compact
; trace data
NEXUS.HTM ON

; select trace mode STREAM to obtain trace data
Trace.Mode STREAM

; suppress generation of TRACE32 tool timestamps
Trace.PortFilter MAX

; load executable to target and to TRACE32 Virtual Memory
Data.LOAD.Elf demo.x /PlusVM

; switch RTS processing to on
RTS.ON

; specify stops on errors if required (just as example)
; RTS.StopOnNoaccesstocode ON

; select the required coverage metric (just as example)
; COVerage.Option.SourceMetric Decision

; display code coverage for HLL functions
COVerage.ListFunc

; display source code with coverage tagging
List E:0x8D04 /COVerage

Go

Break

…

; save coverage results to file
COVerage.SAVE demo.acd

; save trace recording
Trace.SAVE demo.ad

; switch RTS processing to off
RTS.OFF

; select trace mode Fifo
Trace.Mode Fifo

…
```

RTS.COMMAND — Issue command to RTS API model

Format: **RTS.COMMAND** *<cmd>* [*<string>*] [*<address>*] [*<time>*] [*<value>*]

Issues a command to all loaded RTS API models. The parameters are interpreted by the loaded models.

See also

- RTS
- RTS.LOAD
- RTS.state
- RTS.UNLOAD

RTS.Init — Initialize RTS

Format: **RTS.Init**

Initializes RTS by clearing the already processed RTS data. The trace buffer is also initialized.

See also

- RTS
- RTS.state

RTS.LOAD — Load RTS API module

Format: **RTS.LOAD** *<file>* [*<parameter>* …]

Loads RTS API library. The parameters are specific for the loaded library.

See also

- RTS
- RTS.COMMAND
- RTS.state
- RTS.UNLOAD

RTS.OFF — Deactivate real-time profiling

Format: **RTS.OFF**

Disable trace processing while trace information is recorded.

The **Trace.Mode** is switched to STREAM, if **Trace.Mode STREAM** was selected when RTS was enabled and to **Fifo** otherwise.

The **COVerage.state** and the **ISTATistic.state** window return to the state they had before **RTS.ON** was done. The **COVerage** and **ISTATistic** results retain in their database.

See also

- RTS
- RTS.state
- COVerage.state
- ISTATistic.state

RTS.ON — Activate real-time profiling

Format: **RTS.ON**

Enables trace processing while trace information is recorded. The **COVerage** and the **ISTATistic** system are cleared.

The trace raw data without tool timestamps are conveyed to the host computer for processing. The trace data are discarded after processing if another **Trace.Mode** than STREAM was selected when RTS was enabled.

The trace raw data and the tool timestamps are conveyed to the host computer for processing if **Trace.Mode STREAM** was selected when RTS was enabled. The trace data are retained in the streaming file. The generation of the tool timestamp can be suppressed by using the command **Trace.PortFilter MAX**.

See also

- RTS
- RTS.state
- <trace>.PortFilter
- COVerage.METHOD

RTS.PROfile — Display performance characteristics charts

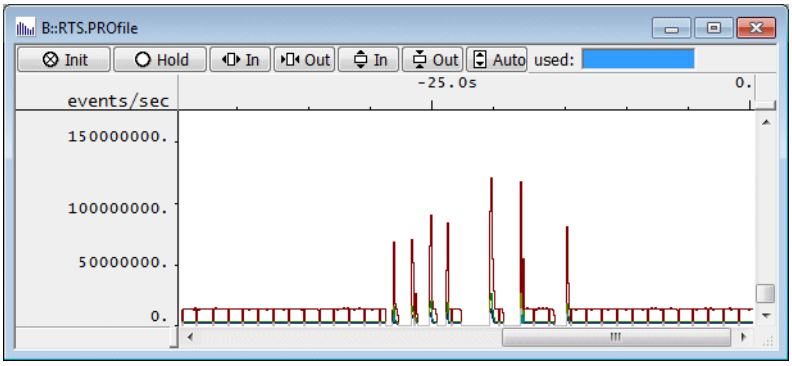
Format: **RTS.PROfile** [{*<event_option>*}] [*<time>*]

<event_option>: **MIPS** | **READS** | **WRITES** | **TaskSwitches**

<time>: **0.1s** | **1.0s** | **10.s**

Currently the results are not conclusive. **RTS.PROFile** is intended to prove that RTS is alive and working.

Displays a window charting the occurrence of events on the vertical axis versus the elapsed time on the horizontal axis. If no *<event_option>* is specified the following events are displayed: MIPS, READS, WRITES. The default update time is 0.1s.



The events are assigned colors according to their position:

- 1st parameter : red
- 2nd parameter : green
- 3rd parameter : blue

The following table explains the event options:

MIPS	Number of instructions executed.
READS	Number of memory read operations executed by the core.
WRITES	Number of memory write operations executed by the core.
TaskSwitches	Number of task switches performed.

Examples:

```
RTS.PROfile 1.s                          ; update chart every second

RTS.PROfile READS 10.s                   ; display chart only for reads
                                         ; update chart every 10 seconds
```

See also

- RTS
- RTS.state

Format: **RTS.RESet**

Restores the default settings and similar to **RTS.Init** clears the already processed RTS data.

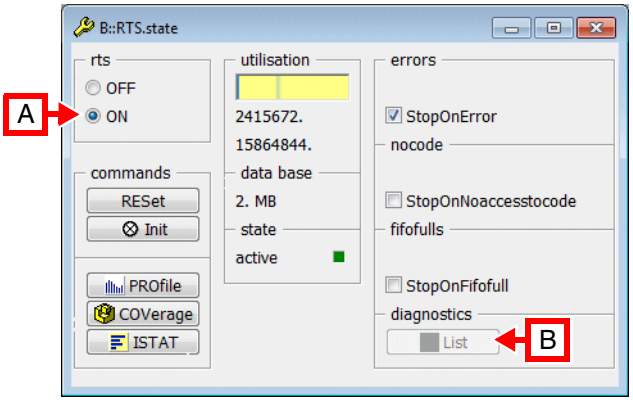
See also

- RTS
- RTS.state

RTS.state Open status and control window

Format: **RTS.state**

Opens the RTS status and control window. This window displays the current status of the RTS system, allows to configure the most important options, and gives access to various analysis options through the buttons at the left side.



A For descriptions of the commands in the **RTS.state** window, please refer to the **RTS.*** commands in this chapter. **Example**: For information about **ON**, see **RTS.ON**.

Exceptions:
- **COVerage** opens the **COVerage.ListModule** window.
- **ISTAT** opens the **ISTATistic.ListModule** window.

B See **RTS.StopOnNoaccesstocode** and **RTS.StopOnFifofull**.

utilisation	Shows the number of trace records transferred to the host. The bottom line shows the number of bytes already processed by RTS.
database	Indicates how much memory on the host is currently used by RTS.
state	Shows the current RTS status e.g. ready, active, no tracing, catching up etc.

See also

- RTS
- RTS.Init
- RTS.OFF
- RTS.PROfile
- RTS.StopOnBadaddress
- RTS.StopOnFifofull
- RTS.StopOnUnknowntask
- RTS.TrackData
- RTS.TriggerConnect
- RTS.TriggerOnRead
- RTS.TriggerOnWTM
- RTS.TriggerWaitForAck
- RTS.UNLOAD
- RTS.RECORDS()
- RTS.COMMAND
- RTS.LOAD
- RTS.ON
- RTS.RESet
- RTS.StopOnError
- RTS.StopOnNoaccesstocode
- RTS.TImeMode
- RTS.TRIGGERACK
- RTS.TriggerOnExecute
- RTS.TriggerOnWrite
- RTS.TriggerSlave
- RTS.UnknownData
- RTS.ERROR()

▲ 'Release Information' in 'Legacy Release History'

RTS.StopOnBadaddress — Stop RTS on VM errors

Format: **RTS.StopOnBadaddress** [**ON** | **OFF**]

Default: OFF.

ON	If **RTS.StopOnBadaddress** is **ON**, the trace recording is stopped when a bad data address is detected. The **diagnostics List** button in the **RTS.state** window provides access to a diagnostic trace listing.
OFF	If **RTS.StopOnBadaddress** is **OFF**, bad data addresses are counted.

See also

- RTS
- RTS.state

Format: **RTS.StopOnError** [**ON** | **OFF**]

Default: ON.

ON	If **RTS.StopOnError** is **ON**, the trace recording is stopped when a **Flow Error** is detected. The **diagnostics List** button in the **RTS.state** window provides access to a diagnostic trace listing.
OFF	If **RTS.StopOnError** is **OFF**, flow errors are counted.

See also

- RTS
- RTS.state

RTS.StopOnFifofull Stop RTS on FIFOFULL

Format: **RTS.StopOnFifofull** [**ON** | **OFF**]

Default: OFF.

ON	If **RTS.StopOnFifoFull** is **ON**, the trace recording is stopped when a FIFOFULL is detected. The **diagnostics List** button in the **RTS.state** window provides access to a diagnostic trace listing.
OFF	If **RTS.StopOnFifoFull** is **OFF**, **FIFOFULLs** are counted.

See also

- RTS
- RTS.state

Format: **RTS.StopOnNoaccesstocode** [**ON** | **OFF**]

RTS requires that the program code that is needed to decode the trace raw data is located in **TRACE32 Virtual Memory**. If TRACE32 cannot find the program code to decode the raw trace data, a Noaccesstocode error occurs.

ON	If **RTS.StopOnNoaccesstocode** is **ON**, the trace recording is stopped when a Noaccesstocode error is detected. The **diagnostics List** button in the **RTS.state** window provides access to a diagnostic trace listing.
OFF (default)	If **RTS.StopOnNoaccesstocode** is **OFF**, Noaccesstocode errors are counted.

See also

- RTS
- RTS.state

RTS.StopOnUnknowntask Stop RTS on unknown task

Format: **RTS.StopOnUnknowntask** [**ON** | **OFF**]

Default: OFF.

ON	If **RTS.StopOnUnknowntask** is **ON**, the trace recording is stopped when an unknown task is detected in the trace.
OFF	If **RTS.StopOnUnknowntask** is **OFF**, unknown tasks are counted.

See also

- RTS
- RTS.state

Format:	**RTS.TImeMode** [**OFF** | **External**]

Default: OFF.

Configures RTS to enable processing of time information. As this increases the RTS processing workload, only enable if required.

OFF	Disables the processing of time information.
External	Enables RTS processing with time information.

See also

- RTS
- RTS.state

RTS.TrackData Enable RTS data tracking

Format:	**RTS.TrackData** [**ON** | **OFF**]

If **ON**, RTS decodes all received data trace messages and writes the data value to the corresponding address into the TRACE32 virtual memory. The data in VM can then be used for further processing without target access.

Currently not supported for ETMv4.

See also

- RTS
- RTS.state

Format: **RTS.TRIGGERACK**

Acknowledges the RTS trigger to continue RTS processing if **RTS.TriggerWaitForAck** is **ON**.

See also

- RTS
- RTS.state

RTS.TriggerConnect Propagate RTS triggers to RTS trigger slaves

Format: **RTS.TriggerConnect** { [*<host>*;]*<port>* }

Sends RTS triggers to other TRACE32 PowerView instances via **InterCom**. InterCom must be enabled in this instance, as well as in the slave instances. The slave instances must set **RTS.TriggerSlave** to receive the master's RTS triggers.

See also

- RTS
- RTS.state

RTS.TriggerOnExecute Generate RTS trigger on execution

Format: **RTS.TriggerOnExecute** {*<address>*}

Selects program addresses that generate an RTS trigger when one of the addresses was found as executed in the program flow.

See also

- RTS
- RTS.state

RTS.TriggerOnRead — Generate RTS trigger on read event

Format: **RTS.TriggerOnRead** {*<address>* [*<data>* | *<bitmask>*] }

Selects data addresses with optional value or bitmask that generate an RTS trigger when a matching read access occurred.

Currently not supported for ETMv4.

See also

- RTS
- RTS.state

RTS.TriggerOnWrite — Generate RTS trigger on write event

Format: **RTS.TriggerOnWrite** {*<address>* [*<data>* | *<bitmask>*] }

Selects data addresses with optional value or bit mask that generate an RTS trigger when a matching write access occurred.

Currently not supported for ETMv4.

See also

- RTS
- RTS.state

RTS.TriggerOnWTM — Generate RTS trigger on watchpoint event

Only supported for PowerPC NEXUS

Format: **RTS.TriggerOnWTM** *<event>* [**ON** | **OFF**]

<event>: **IAC1** | **IAC2** | **IAC3** | **IAC4** | **IAC5** | **IAC6** | **IAC7** | **IAC8**
DAC1 | **DAC2** | **DAC3** | **DAC4**

Selects watchpoint hit events that generate an RTS trigger when a matching watchpoint hit message (WHM) occurred. Remote API clients programs can register for RTS trigger events to react on RTS triggers.

See also

- RTS
- RTS.state

Format: **RTS.TriggerSlave** [**ON** | **OFF**]

Enables receiving and processing of RTS triggers.

See also

- RTS
- RTS.state

RTS.TriggerWaitForAck Stall RTS processing until trigger acknowledged

Format: **RTS.TriggerWaitForAck** [**ON** | **OFF**]

If **ON** and an RTS trigger occurred, RTS processing is suspended until the RTS trigger is acknowledged using **RTS.TRIGGERACK** or through Remote API.

See also

- RTS
- RTS.state

RTS.UnknownData HTM unknown data

Format: **RTS.UnknownData** *<data>*

This command is only needed in some rare cases when RTS is used together with **HTM** and data is lost because of trace FIFO overflows.

See also

- RTS
- RTS.state

Format: **RTS.UNLOAD** [<*file*>]

Unloads an RTS API library.

See also

- RTS
- RTS.COMMAND
- RTS.LOAD
- RTS.state

RunTime

Runtime measurement

See also

- <trace>.Arm
- <trace>.AutoArm
- <trace>.AutoInit
- <trace>.BookMark
- <trace>.Chart
- <trace>.CLOCK
- <trace>.ComPare
- <trace>.DISable
- <trace>.EXPORT
- <trace>.FILE
- <trace>.Find
- <trace>.FindAll
- <trace>.FindChange
- <trace>.GOTO
- <trace>.Init
- <trace>.LOAD
- <trace>.OFF
- <trace>.PROfileChart
- <trace>.REF
- <trace>.RESet
- <trace>.SAVE
- <trace>.SIZE
- <trace>.STATistic
- <trace>.Timing
- <trace>.TRACK
- <trace>.View
- <trace>.ZERO
- RunTime.List
- RunTime.Mode
- RunTime.refA
- RunTime.refB
- RunTime.SHOW
- RunTime.state
- RunTime.WAIT
- RunTime.ACCURACY()
- RunTime.ACTUAL()
- RunTime.LAST()
- RunTime.LASTRUN()
- RunTime.REFA()
- RunTime.REFB()

▲ 'RunTime Functions' in 'General Function Reference'
▲ 'Release Information' in 'Legacy Release History'

Overview RunTime

The **RunTime** command group and the **RunTime()** functions are used to measure the time the target has been executing code.

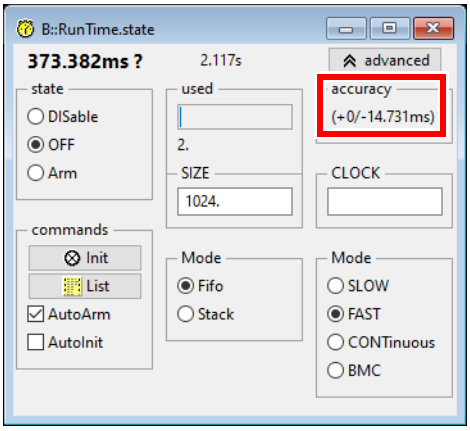
The runtime is calculated from the target reset and increments while the target executes program code. The target runtime is frozen when the target is stopped.

There are different methods of measuring the runtime with different accuracies. The available methods depend on the target hardware and the debug interface (debugger-based vs. trace-based).

On some cores, the runtime can also be based on onchip counters, which reduce the time lag of the external time.

Runtime Measurements Using the Debugger

The RunTime counter allows to measure the program execution times between two breakpoints. The accuracy of the measurement depends on the features provided by the debug interface. The measurement error is displayed in the extended view of **RunTime.state** or by the function **RunTime.ACCURACY()**.



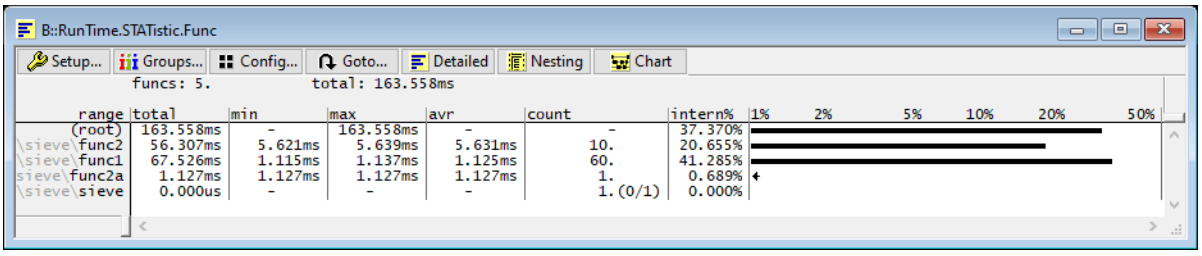
The polling rate can be influenced by selecting one of the mode **RunTime.Mode SLOW** | **FAST** | **CONTinuous**.

Nested Function Analysis

The RunTime command group can be used with SPOT breakpoints set on function entries and exits to get the time of execution of nested functions.

Example:

```
RunTime.RESet
RunTime.OFF
Break.SetFunc func1 /SPOT
Break.SetFunc func2 /SPOT
Break.SetFunc func2a /SPOT
Go.direct sieve
RunTime.STATistic.Func
```



The following functions can be used to obtain various runtime-related values.

- **RunTime.ACTUAL()** returns the total measured program runtime. The total program runtime is reset by the following commands **SYStem.Mode Up**, **RunTime.Init, RunTime.Mode** *<mode>*.
- **RunTime.LASTRUN()** returns the last measured runtime, i.e. the time of a single step or the time between the last **Go** and **Break**.
- **RunTime.LAST()** the return value is calculated by:

```
&difftime=CONVert.TIMEUSTOINT(RunTime.ACTUAL()-RunTime.LAST())
```

- **RunTime.ACCURACY()** returns the inaccuracy of the runtime measurement. It depends on the **RunTime.Mode**.

The following RunTime functions are deprecated:

- **RunTime.REFA()** returns the reference value A. There is a homonymous command to set the value.
- **RunTime.REFB()** returns the reference value B. There is a homonymous command to set the value.

RunTime.List

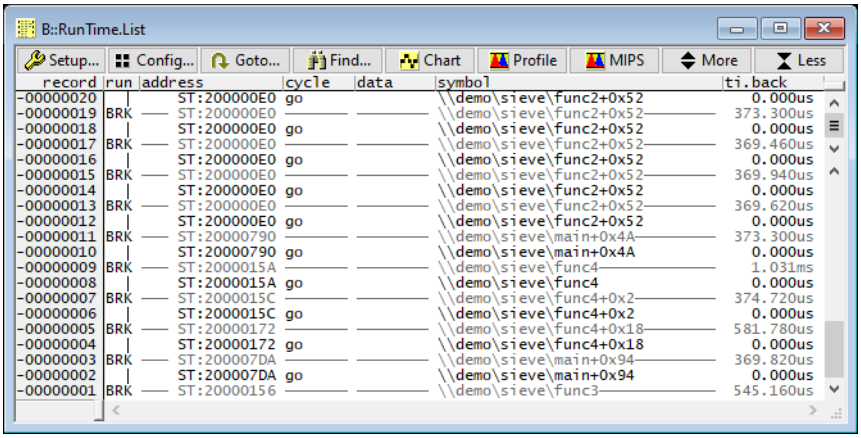
List runtime logs

[build 149833 - TRACE32 Release 02/2023]

Format:	**RunTime.List** [*<parameter>*]
<parameter>:	[*<record>* | *<record_range>* | *<time>* | *<time_range>*] [*<items>* …] [/*<options>*]

Opens a window displaying the runtime logs.

<parameter>	For a detailed description of the parameters and options, refer to **Trace.List** command.



The address at which the program execution was started (go, break or implicitly after a spot break point) is recorded in the runtime trace with the cycle type **go** and a zero timestamp.

The address where the program execution was stopped is tagged with **BRK** and shows the program execution time since the last start.

See also

- RunTime
- RunTime.state

Format:	**RunTime.Mode** *<mode1>* | *<mode2>*
<mode1>:	**SLOW** | **FAST** | **CONTinuous** **BMC** **TB** (PowerPC only)
<mode2>:	**Fifo** | **Stack**

- **Target hardware:** If TRACE32 is in “running” state, it is constantly polling whether the program execution is still running or has already been stopped. There are different polling rates. The selected polling rate determines the inaccuracy of the measurement. The inaccuracy is displayed in the **RunTime configuration window**.
- **Virtual target:** If TRACE32 is in „running“ state, it is constantly polling whether the program execution is still running or has already been stopped. With each polling TRACE32 also queries the timestamp of the virtual target/simulation. If a timestamp is supplied, it is accurate.

SLOW	The polling interval depends on the setting of the **SETUP.UpdateRATE** command and can thus be selected by the user (target hardware). The timestamp is queried with the poll. The result is very accurate (default for virtual targets).
FAST	The polling interval is 1 ms (default for target hardware).
CONTinous	The polling is performed as often as possible. This leads to a high load on the (JTAG) debug interface and collides with other activities over this interface such as run-time memory access (target hardware only).

Very accurate measurement results are obtained when using an onchip cycle counter (target hardware only). This is not supported by all core architectures.

BMC	An onchip counter is used for the runtime measurement. The core clock needs to be set using the **BMC.CLOCK** command. This mode can only be used: • if the core clock remains constant during the program runtime. • if all cores in an SMP system run at the same clock frequency.
TB	Returns the value of the runtime based on the time base registers (TBU and TBL). This option also depends on the value of the clock.

The individual measurements can be logged in the RunTime trace. The runtime trace is located on the host computer. Its default size is 1024 entries. Its size can be changed with the **RunTime.Size** command. Logging can be switched on with the **RunTime.OFF** command. The log can be displayed with the **RunTime.List** command.

The RunTime Trace can operate in two modes.

Fifo	If the trace is full, new records will overwrite older ones. The runtime trace includes always the last logs.
Stack	If the trace is full, the logging will be stopped. The runtime trace includes always the first log.

See also

- RunTime
- RunTime.state

RunTime.refA — Set reference

Format: **RunTime.refA** (deprecated)

The RunTime command group has been completely revised with build 155615/TRACE32 Release 02/2023. Since then this command is set to deprecated. If you still have scripts that use this command, please use the **RunTime.Show** command to display the results of the runtime measurement.

This command sets the reference value A to the current runtime. Typically the feature is used to record the moment of an “important event” like entering an interrupt handler etc.

See also

- RunTime
- RunTime.state

RunTime.refB — Set reference

Format: **RunTime.refB** (deprecated)

The RunTime command group has been completely revised with build 155615/TRACE32 Release 02/2023. Since then this command is set to deprecated. If you still have scripts that use this command, please use the **RunTime.Show** command to display the results of the runtime measurement.

This command sets the reference value B to the current runtime.

See also

- RunTime
- RunTime.state

RunTime.SHOW — Display results

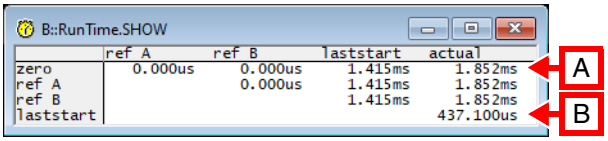
Format: **RunTime.SHOW** (deprecated)

This command was introduced with build 155615/TRACE32 Release 02/2023 to be able to display the results of the RunTime measurements even if scripts with deprecated commands are used.

This command displays the RunTime counter window with a matrix of values related to runtime measurements.

Each cell of the matrix shows the difference between the value denoted by the *column header* and the value denoted by the *row header*. As the first line's row header is "zero", the line shows the effective values e.g. (refA - 0), (refB - 0), ... The cell at (column refB / row ref A) shows refB - refA i.e. the runtime between both values.

Reference values can be set to the current runtime ("actual") by double-clicking the appropriate element.



A Total time since the last **SYStem.Mode Up** or **RunTime.Init**

B Time since the latest program start

In the deprecated version of the RunTime command group, the measurement method of TRACE32 was set automatically. The following measurement methods were set depending on the core architecture.

Feature	RunTime counter is started...	Runtime counter is stopped...	Accuracy
"CPU running" signal	...after detecting the "CPU running" signal	... after deassertion of the "CPU running" signal	High
NEXUS Debug Status Message	... after receiving the "Debug Mode Left" message	... after receiving the "Debug Mode Entered" message	High

Feature	RunTime counter is started...	Runtime counter is stopped...	Accuracy
“CPU stopped” signal	...by TRACE32 after starting the CPU	...with the activation of the “CPU stopped” signal	RunTime counter start is imprecise
Polling the PC	...by TRACE32 after starting the CPU	...by TRACE32 after CPU is stopped	RunTime counter start and stop are imprecise

“CPU running” signal

Some processor architectures provide a “CPU running” signal within the debug interface (e.g. DBACK for ARM7/ARM9). This feature allows an exact measurement by the RunTime counter.

Debug Status Messages

Most NEXUS interfaces provide Debug Status Messages which indicate the start/stop of the program execution. The maximum measurement error is calculated as follows:

$(SizeOfMessageFifo \times 2)/(MCKOFactor)$ clock cycles

While MCKOFactor is the value entered via the command **SYStem.Option.MCKO** *<factor>*.

Hardware signal indicating “CPU stopped”

Some processor architectures provide a “CPU stopped” signal within the debug interface (e.g. DE for the DSP56K). This feature allows an exact stop of the RunTime counter, but the start of the RunTime counter can´t be synchronized exactly with the start of the program execution.

Polling

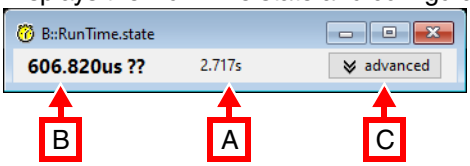
For most processor architectures the RunTime counter is started/stopped by TRACE32. Thus the measurement can’t be exactly synchronized with the CPU start/stop.

See also

- RunTime
- RunTime.state

Format:	**RunTime.state**

Displays the RunTime state and configuration window.

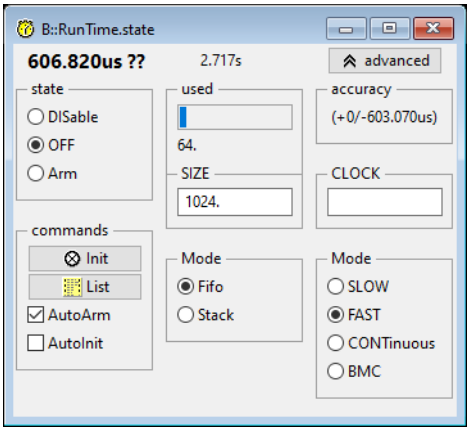


A Total time since the last **SYStem.Mode Up** or **RunTime.Init**

B Time since the latest program start. Question marks may be shown if the accuracy of the error is not exact:

- More than 10% error shows “??”
- More than 1% error shows “?”

C Advanced button can be used to display RunTime settings.



See also

- <trace>.Arm
- <trace>.AutoArm
- <trace>.AutoInit
- <trace>.BookMark
- <trace>.Chart
- <trace>.CLOCK
- <trace>.ComPare
- <trace>.DISable
- <trace>.EXPORT
- <trace>.FILE
- <trace>.Find
- <trace>.FindAll
- <trace>.FindChange
- <trace>.GOTO
- <trace>.Init
- <trace>.LOAD
- <trace>.OFF
- <trace>.PROfileChart
- <trace>.REF
- <trace>.RESet
- <trace>.SAVE
- <trace>.SIZE
- <trace>.STATistic
- <trace>.Timing
- <trace>.TRACK
- <trace>.View
- <trace>.ZERO
- RunTime
- RunTime.List
- RunTime.Mode
- RunTime.refA
- RunTime.refB
- RunTime.SHOW
- RunTime.WAIT
- RunTime.ACCURACY()
- RunTime.ACTUAL()
- RunTime.LAST()
- RunTime.LASTRUN()
- RunTime.REFA()
- RunTime.REFB()

▲ ’Release Information’ in ’Legacy Release History’

Format:	**RunTime.WAIT** [*<condition>*] [*<period>*]

If a virtual target or a simulation system is debugged, the time behavior can be faster or slower than that of the real target hardware. The specified <period> here refers to the simulated time. Same as **WAIT** /**RunTime**.

<condition>	PRACTICE functions that return the boolean values TRUE or FALSE.
<period>	Min.: 1ms Max.: 100000s Without unit of measurement, the specified value will be interpreted as seconds and must be an integer. See below.

Example 1: Run program execution for a second.

```
Go
RunTime.WAIT 3.s
Break
```

Example 2: Start program execution and wait until core stops at a breakpoint, with 3.s timeout.

```
Go main
RunTime.WAIT !STATE.RUN() 3.s

IF STATE.RUN()
(
     PRINT %ERROR "function main not reached!"
     ENDDO
)
```

See also

- RunTime
- RunTime.state

RunTime.Arm — Arm the trace

See command **<trace>.Arm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 134).

RunTime.AutoArm — Arm automatically

See command **<trace>.AutoArm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 135).

RunTime.AutoInit — Automatic initialization

See command **<trace>.AutoInit** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 140).

RunTime.BookMark — Set a bookmark in trace listing

See command **<trace>.BookMark** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 140).

RunTime.Chart — Display trace contents graphically

See command **<trace>.Chart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 144).

RunTime.CLOCK — Clock to calculate time out of cycle count information

See command **<trace>.CLOCK** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 191).

RunTime.ComPare — Compare trace contents

See command **<trace>.ComPare** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 192).

RunTime.DISable — Disable the trace

See command **<trace>.DISable** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 197).

RunTime.EXPORT — Export trace data for processing in other applications

See command **<trace>.EXPORT** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 212).

RunTime.FILE — Load a file into the file trace buffer

See command **<trace>.FILE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 233).

RunTime.Find — Find specified entry in trace

See command **<trace>.Find** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 235).

RunTime.FindAll — Find all specified entries in trace

See command **<trace>.FindAll** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 237).

RunTime.FindChange — Search for changes in trace flow

See command **<trace>.FindChange** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 238).

RunTime.GOTO — Move cursor to specified trace record

See command **<trace>.GOTO** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 244).

RunTime.Init — Initialize trace

See command **<trace>.Init** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 246).

RunTime.LOAD — Load trace file for offline processing

See command **<trace>.LOAD** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 270).

RunTime.OFF — Switch off

See command **<trace>.OFF** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 278).

RunTime.PROfileChart — Profile charts

See command **<trace>.PROfileChart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 283).

RunTime.REF — Set reference point for time measurement

See command **<trace>.REF** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 357).

RunTime.RESet — Reset command

See command **<trace>.RESet** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 357).

RunTime.SAVE — Save trace for postprocessing in TRACE32

See command **<trace>.SAVE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 358).

RunTime.SIZE — Define buffer size

See command **<trace>.SIZE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 373).

RunTime.STATistic — Statistic analysis

See command **<trace>.STATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 378).

RunTime.Timing — Waveform of trace buffer

See command **<trace>.Timing** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 499).

RunTime.TRACK — Set tracking record

See command **<trace>.TRACK** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 502).

RunTime.View — Display single record

See command **<trace>.View** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 504).

RunTime.ZERO — Align timestamps of trace and timing analyzers

See command **<trace>.ZERO** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 505).