<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TASK</strong></td>
<td>OS Awareness for TRACE32</td>
</tr>
<tr>
<td>TASK</td>
<td>Control memory access</td>
</tr>
<tr>
<td>TASK.ACCESS</td>
<td>Deprecated</td>
</tr>
<tr>
<td>TASK.Break</td>
<td>Reread task list</td>
</tr>
<tr>
<td>TASK.CACHEFLUSH</td>
<td>Configure OS Awareness</td>
</tr>
<tr>
<td>TASK.COPYDOWN</td>
<td>Copy file from host into target</td>
</tr>
<tr>
<td>TASK.COPYUP</td>
<td>Copy file from target into host</td>
</tr>
<tr>
<td>TASK.CreateExtraID</td>
<td>Create a virtual task</td>
</tr>
<tr>
<td>TASK.CreateID</td>
<td>Create virtual task</td>
</tr>
<tr>
<td>TASK.Debug</td>
<td>Deprecated</td>
</tr>
<tr>
<td>TASK.DeleteID</td>
<td>Delete virtual task</td>
</tr>
<tr>
<td>TASK.DETACH</td>
<td>Detach from task</td>
</tr>
<tr>
<td>TASK.INSTALL</td>
<td>Deprecated</td>
</tr>
<tr>
<td>TASK.KILL</td>
<td>End task</td>
</tr>
<tr>
<td>TASK.List</td>
<td>Information about tasks</td>
</tr>
<tr>
<td>TASK.List.MACHINES</td>
<td>List machines</td>
</tr>
<tr>
<td>TASK.List.SPACES</td>
<td>List MMU spaces</td>
</tr>
<tr>
<td>TASK.List.tasks</td>
<td>List all running tasks</td>
</tr>
<tr>
<td>TASK.List.TREE</td>
<td>Display tasks in a tree structure</td>
</tr>
<tr>
<td>TASK.ListID</td>
<td>List virtual tasks</td>
</tr>
<tr>
<td>TASK.MTXTRC</td>
<td>Internal</td>
</tr>
<tr>
<td>TASK.NAME</td>
<td>Translation of task magic number to task name</td>
</tr>
<tr>
<td>TASK.NAME.DELETE</td>
<td>Delete a task name table entry</td>
</tr>
<tr>
<td>TASK.NAME.RESET</td>
<td>Reset task name table</td>
</tr>
</tbody>
</table>

©1989-2020 Lauterbach GmbH
General Commands Reference Guide

**TASK.NAME.Set** Set a task name table entry 41
**TASK.NAME.view** Show task name translation table 41
**TASK.NoBreak** Deprecated 41
**TASK.NoDebug** Deprecated 42
**TASK.NoStop** Deprecated 42
**TASK.OFF** Deprecated 42
**TASK.ON** Deprecated 43
**TASK.ORTI** AUTOSAR/OSEK support 44
**TASK.ORTI.CPU** Set OSEK SMP CPU number 44
**TASK.ORTI.load** Configure OS Awareness for OSEK/ORTI 44
**TASK.ORTI.NOSTACK** Exclude an ORTI task from stack evaluation 45
**TASK.ORTI.SPLITSTACK** Split stack analysis of idle ORTI task to cores 46
**TASK.RESet** Reset OS Awareness 47
**TASK.RUN** Load task 47
**TASK.select** Display context of specified task 48
**TASK.SETDIR** Set the awareness directory 49
**TASK.STacK** Stack usage coverage 50
**TASK.STacK.ADD** Add stack space coverage 51
**TASK.STacK.DIRection** Define stack growth direction 53
**TASK.STacK.Init** Initialize unused stack space 53
**TASK.STacK.PATtern** Define stack check pattern 53
**TASK.STacK.PATternGAP** Define check pattern gap 55
**TASK.STacK.ReMove** Remove stack space coverage 55
**TASK.STacK.RESet** Reset stack coverage 56
**TASK.STacK.view** Open stack space coverage 57
**TASK.Stop** Deprecated 58

**TCB** Trace control block 59
**TCB.AllBranches** Broadcast all branches 60
**TCB.CPU** Broadcast information for specified CPU only 60
**TCB.CycleAccurate** Cycle accurate tracing 61
**TCB.DataTrace** Broadcast specified address and data information 62
**TCB.EX** Broadcast exception level information 63
**TCB.FCR** Broadcast function call-return information 63
**TCB.IM** Broadcast instruction cache miss information 63
**TCB.InstructionCompletionSizeBits** Specify size of completion message 64
**TCB.KE** Broadcast kernel mode information 64
**TCB.LSM** Broadcast load store data cache information 65
**TCB.OFF** Switch TCB off 65
**TCB.ON** Switch TCB on 65
**TCB.PCTrace** Broadcast program counter trace 66
**TCB.PortFilter** Disable port filter 66
**TCB.PortMode** Specify trace clock ratio 67

©1989-2020 Lauterbach GmbH
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCB.PortWidth</td>
<td>Specify trace port width</td>
<td>67</td>
</tr>
<tr>
<td>TCB.Register</td>
<td>Display TCB control register</td>
<td>68</td>
</tr>
<tr>
<td>TCB.RESet</td>
<td>Reset TCB setup to default</td>
<td>69</td>
</tr>
<tr>
<td>TCB.SourceSizeBits</td>
<td>Specify number of bit for core information in trace</td>
<td>69</td>
</tr>
<tr>
<td>TCB.STALL</td>
<td>Stall CPU for complete trace</td>
<td>69</td>
</tr>
<tr>
<td>TCB.state</td>
<td>Display TCB setup</td>
<td>70</td>
</tr>
<tr>
<td>TCB.SV</td>
<td>Broadcast supervisor mode information</td>
<td>71</td>
</tr>
<tr>
<td>TCB.SyncPeriod</td>
<td>Specify TCB sync period</td>
<td>71</td>
</tr>
<tr>
<td>TCB.TC</td>
<td>Broadcast information for specified HW thread</td>
<td>72</td>
</tr>
<tr>
<td>TCB.ThreadSizeBits</td>
<td>Specify number of bit for thread information in trace</td>
<td>72</td>
</tr>
<tr>
<td>TCB.Type</td>
<td>Specify TCB type</td>
<td>73</td>
</tr>
<tr>
<td>TCB.UM</td>
<td>Broadcast user mode information</td>
<td>73</td>
</tr>
<tr>
<td>TCB.Version</td>
<td>Specify trace cell version</td>
<td>74</td>
</tr>
<tr>
<td>TERM.CLEAR</td>
<td>Clear terminal window</td>
<td>79</td>
</tr>
<tr>
<td>TERM.CLOSE</td>
<td>Close files</td>
<td>79</td>
</tr>
<tr>
<td>TERM.CMDLINE</td>
<td>Specify a command line</td>
<td>79</td>
</tr>
<tr>
<td>TERM.FastWRITE</td>
<td>Fast data write</td>
<td>79</td>
</tr>
<tr>
<td>TERM.GATE</td>
<td>Terminal with virtual hosting</td>
<td>81</td>
</tr>
<tr>
<td>TERM.HARDCOPY</td>
<td>Print terminal window contents</td>
<td>81</td>
</tr>
<tr>
<td>TERM.HEAPINFO</td>
<td>Define memory heap parameters</td>
<td>82</td>
</tr>
<tr>
<td>TERM.LocalEcho</td>
<td>Enables/disables local echo for new terminal windows</td>
<td>82</td>
</tr>
<tr>
<td>TERM.METHOD</td>
<td>Select terminal protocol</td>
<td>83</td>
</tr>
<tr>
<td>TERM.Mode</td>
<td>Define terminal type</td>
<td>86</td>
</tr>
<tr>
<td>TERM.Out</td>
<td>Send data to virtual terminal</td>
<td>87</td>
</tr>
<tr>
<td>TERM.PIPE</td>
<td>Connect terminal to named pipe</td>
<td>87</td>
</tr>
<tr>
<td>TERM.PipeREAD</td>
<td>Connect terminal input to named pipe</td>
<td>88</td>
</tr>
<tr>
<td>TERM.PipeWRITE</td>
<td>Connect terminal output to named pipe</td>
<td>88</td>
</tr>
<tr>
<td>TERM.Protocol</td>
<td>Select terminal protocol</td>
<td>88</td>
</tr>
<tr>
<td>TERM.PULSE</td>
<td>Enable pulse generator for transfers</td>
<td>89</td>
</tr>
<tr>
<td>TERM.Rate</td>
<td>Define polling rate</td>
<td>89</td>
</tr>
<tr>
<td>TERM.READ</td>
<td>Get terminal input from file</td>
<td>90</td>
</tr>
</tbody>
</table>
TERM.RESet  Reset terminal parameters
TERM.SCROLL  Enable automatic scrolling for terminal window
TERM.SIZE  Define size of terminal window
TERM.TCP  Route terminal input/output to TCP port
TERM.TELNET  Open TELNET terminal window
TERM.TRIGGER  Trigger on string in terminal window
TERM.Vector  Define interrupt vectors
TERM.view  Terminal display
TERM.WRITE  Write terminal output to file

TPIU  Trace Port Interface Unit (TPIU)

Overview TPIU
TPIU.CLEAR  Re-write the TPIU registers
TPIU.IGNOREZEROS  Workaround for a special chip
TPIU.NOFLUSH  Workaround for a chip bug affecting TPIU flush
TPIU.PortClock  Inform debugger about HSSTP trace frequency
TPIU.PortMode  Select the operation mode of the TPIU
TPIU.PortSize  Select interface type and port size of the TPIU
TPIU.RefClock  Set up reference clock for HSSTP
TPIU.Register  Display TPIU registers
TPIU.RESet  Reset TPIU settings
TPIU.state  Display TPIU configuration window
TPIU.SWVPrescaler  Set up SWV prescaler
TPIU.SWVZEROS  Workaround for a chip bug
TPIU.SyncPeriod  Set period of sync packet injection

TPU  Base address

TPU.BASE  Base address
TPU.Break  Break TPU
TPU.Dump  Memory display
TPU.Go  Start TPU
TPU.Register.ALL  Register operation mode
TPU.Register.NEWSTEP  New debugging mode
TPU.Register.Set  Register modification
TPU.Register.view  Register display
TPU.RESet  Disable TPU debugger
TPU.SCAN  Scanning TPU
TPU.SELect  Select TPU for debugging
TPU.Step  Single step TPU
TPU.view  View TPU channels

Trace  Trace configuration and display
About the Command Placeholder <trace> 111
What to know about the TRACE32 default settings for Trace.METHOD 111
Types of Replacements for <trace> 113
Replacing <trace> with a Trace Method - Examples 113
Replacing <trace> with a Trace Evaluation - Example 114
Replacing <trace> with RTS for Real-time Profiling - Example 115
Replacing <trace> with Trace Source and Trace Method - Examples 116
How to access the trace sources in TRACE32 118
List of <trace> Command Groups consisting of <trace_source><trace_method> 119
Related Trace Command Groups 120
<trace>.ABCDEF Sampling configuration for probes ABCDEF 121
<trace>.ACCESS Define access path to program code for trace decoding 122
<trace>.ADDRESS Software trace address 124
<trace>.Arm Arm the trace 125
<trace>.AutoArm Arm automatically 126
<trace>.AutoFocus Calibrate AUTOFOCUS preprocessor 126
Preprocessor with AUTOFOCUS Technology 129
<trace>.AutoInit Automatic initialization 131
<trace>.AutoStart Automatic start 131
<trace>.AutoTEST Continuous measurement 132
<trace>.BookMark Set a bookmark in trace listing 134
<trace>.BookMarkToggle Toggles a single trace bookmark 136
<trace>.Break Stop trace 137
<trace>.Chart Display trace contents graphically 138
Parameters 138
General Options 139
Drag and Drop 141
<trace>.Chart.Address Time between program events as a chart 146
<trace>.Chart.ChildTREE Display callee context of a function as chart 146
<trace>.Chart.DatasYmbol Analyze pointer contents graphically 148
<trace>.Chart.DistriB Distribution display graphically 150
<trace>.Chart.Func Function activity chart 152
<trace>.Chart.GROUP Group activity chart 153
<trace>.Chart.INTERRUPTTREE Display interrupt nesting 154
<trace>.Chart.LINE Graphical HLL lines analysis 155
<trace>.Chart.MODULE Code execution broken down by module as chart 156
<trace>.Chart.Nesting Show function nesting at cursor position 157
<trace>.Chart.PAddress Which instructions accessed data address 158
<trace>.Chart.PROGRAM Code execution broken down by program as chart 159
<trace>.Chart.PsYmbol Shows which functions accessed data address 159
<trace>.Chart.sYmbol Symbol analysis 160
<trace>.Chart.TASK Task activity chart 164
<trace>.Chart.TASKFunc Task related function run-time analysis (legacy) 165
<trace>.Chart.TASKINTR
Display ISR2 time chart (ORTI) 165

<trace>.Chart.TASKKernel
Task run-time chart with kernel markers (flat) 166

<trace>.Chart.TASKSRV
Service routine run-time analysis 167

<trace>.Chart.TASKState
Task state analysis 168

<trace>.Chart.TASKVSINTERRUPT
Time chart 169

<trace>.Chart.TASKVSINTR
Time chart 170

<trace>.Chart.TREE
Display function chart as tree view 171

<trace>.Chart.Var
Variable chart 171

<trace>.Chart.VarState
Variable activity chart 172

<trace>.CLEAR
Clear FDX communication buffers 175

<trace>.CLOCK
Clock to calculate time out of cycle count information 175

<trace>.CLOSE
Close FDX files 176

<trace>.ComPare
Compare trace contents 176

<trace>.CSELect
Select signal for counter 178

<trace>.CustomTrace
Custom trace 179

<trace>.CustomTrace.<label>.COMMAND
Send command to specific DLL 179

<trace>.CustomTrace.<label>.UNLOAD
Unload a single DLL 179

<trace>.CustomTraceLoad
Load a DLL for trace analysis/Unload all DLLs 180

<trace>.DISable
Disable the trace 181

<trace>.DISableChannel
Disable FDX communication 181

<trace>.DisConfig
Trace disassembler configuration 182

<trace>.DisConfig.CYcle
Trace disassemble setting 182

<trace>.DisConfig.FlowMode
Enable FlowTrace analysis 183

<trace>.DisConfig.RESet
Reset trace disassemble setting 184

<trace>.DisConfig.view
Trace disassemble setting 184

<trace>.DRAW
Plot trace data against time 185

<trace>.DRAW.channel
Plot no-data values against time 185

<trace>.DRAW.Data
Plot data values against time 187

<trace>.DRAW.Var
Plot variable values against time 191

<trace>.Enable
Operation mode 195

<trace>.ENableChannel
Enable FDX communication 195

<trace>.EXPORT
Export trace data for processing in other applications 196

<trace>.EXPORT.Ascii
Export trace data as ASCII 197

<trace>.EXPORT.Bin
Export trace data as binary file 198

<trace>.EXPORT.BRANCHFLOW
Export branch events from trace data 199

<trace>.EXPORT.CSVFunc
Export the function nesting to a CSV file 200

<trace>.EXPORT.TracePort
Export trace packets as recorded at trace port 207

<trace>.EXPORT.VCD
Export trace data in VCD format 208

<trace>.EXPORT.VERILOG
Export trace data in VERILOG format 209
<trace>.EXPORT.VHDL
Export trace data in VHDL format
209

<trace>.FILE
Load a file into the file trace buffer
210

<trace>.Find
Find specified entry in trace
212

<trace>.FindAll
Find all specified entries in trace
216

<trace>.FindChange
Search for changes in trace flow
217

<trace>.FLOWPROCESS
Process flowtrace
218

<trace>.FLOWSTART
Restart flowtrace processing
218

<trace>.Get
Display input level
219

<trace>.GOTO
Move cursor to specified trace record
221

<trace>.IMPORT
Import trace information
223

<trace>.IMPORT.CoreByteStream
Import pure single core trace data
225

<trace>.IMPORT.ETB
Import on-chip trace data
226

<trace>.IMPORT.flow
Import bus trace data
226

<trace>.IMPORT.GUESSWRAP
Guess wrap pointer
227

<trace>.IMPORT.STP
Import STP recording from file (nibble)
227

<trace>.IMPORT.STPByteStream
Import STP recording from file (byte)
228

<trace>.IMPORT.TraceFile
Import trace data where processing has failed
228

<trace>.IMPORT.TracePort
Import off-chip trace data
229

<trace>.IMPORT.VCD
Import recorded signals in VCD file format
229

<trace>.IMPORT.WRAP
Define wrap pointer
230

<trace>.InChannel
Inchannel state display
231

<trace>.Init
Initialize trace
231

<trace>.JKLMNO
Sampling configuration for probes JKLMNO
232

<trace>.JOINFILE
Concatenate several trace recordings
232

<trace>.LEVEL
Select trigger level manually
232

<trace>.List
List trace contents
235

<trace>.ListNesting
Analyze function nesting
247

<trace>.ListVar
List variable recorded to trace
250

<trace>.LOAD
Load trace file for offline processing
253

<trace>.MERGEFILE
Combine two trace files into one
255

Trace.METHOD
Select trace method
256

<trace>.Mode
Set the trace operation mode
258

<trace>.MUX
Select channels
262

<trace>.OFF
Switch off
262

<trace>.PC
Display PC in real-time
262

<trace>.PipePROTO
Unload all DLLs
263

<trace>.PipePROTO.COMMAND
Send command to DLLs
263

<trace>.PipePROTO.load
Define a user-supplied DLL as trace sink
263

<trace>.PipeWRITE
Connect to a named pipe to stream trace data
264

<trace>.PortFilter
Specify utilization of trace memory
264

<trace>.PortSize
Set external port size
266

<trace>.PortType
Specify trace interface
266

<trace>.PROfile
Display counter profile
268
<trace>.PROfileChart
Profile charts 269
<trace>.PROfileChart.COUNTER
Display a profile chart 270
<trace>.PROfileChart.DISTance
Time interval for a single event 271
<trace>.PROfileChart.DISTRIBUTion
Distribution display in timeslices 273
<trace>.PROfileChart.DURATION
Time between two events 274
TRACE32 PowerTrace 274
TRACE32-ICE and TRACE32-FIRE 277
<trace>.PROfileChart.GROUP
Group profile chart 278
<trace>.PROfileChart.MODULE
HLL-line profile chart 279
<trace>.PROfileChart.MODULE
Module profile chart 280
<trace>.PROfileChart.PROGRAM
Program profile chart 281
<trace>.PROfileChart.Rate
Event frequency 282
<trace>.PROfileChart.SYMBOL
Dynamic program behavior graphically (flat) 284
<trace>.PROfileChart.TASK
Dynamic task behavior graphically (flat) 288
<trace>.PROfileSTATistic
Statistical analysis in a table versus time 290
<trace>.PROfileSTATistic.LINE
Statistical analysis for HLL lines 290
<trace>.Program
Program trigger unit 290
<trace>.PROTOcol
Protocol analysis 292
<trace>.PROTOcol.CHART
Graphic display for user-defined protocol 292
<trace>.PROTOcol.DRAW
Graphic display for user-defined protocol 294
<trace>.PROTOcol.EXPORT
Export trace buffer for user-defined protocol 295
<trace>.PROTOcol.FIND
Find in trace buffer for user-defined protocol 296
<trace>.PROTOcol.LIST
Display trace buffer for user-defined protocol 297
<trace>.PROTOcol.PROFILECHART
Profile chart for user-defined protocol 300
<trace>.PROTOcol.STATISTIC
Display statistics for user-defined protocol 302
Protocol specific Options 304
  Options for ASYNC 304
  Options for CAN 304
  Options for I²C 305
  Options for JTAG 305
  Options for USB 306
<trace>.Rate
Select sampling rate 307
<trace>.REF
Set reference point for time measurement 308
<trace>.ReProgram
Program trigger unit 309
<trace>.RESet
Reset command 309
<trace>.SAMPLE
Set AutoFocus sample time offset 309
<trace>.SAVE
Save trace for postprocessing in TRACE32 311
  Parameters 311
  Options 312
<trace>.Select
Select trigger/counter line 315
<trace>.SelfArm
Automatic restart of trace recording 315
<trace>.SET
Select line for recording 317
<trace>.ShowFocus
Display data eye for AUTOFOCUS preprocessor 317
<trace>.ShowFocusClockEye Display clock eye 321
<trace>.ShowFocusEye Display data eye 323
<trace>.SIZE Define buffer size 326
<trace>.SLAVE Select slave mode 327
<trace>.SnapShot Restart trace capturing once 328
<trace>.SPY Adaptive stream and analysis 328
<trace>.state Display trace configuration window 330
<trace>.STATistic Statistic analysis 333
<trace>.STATistic.Address Time between up to 8 program events 334
<trace>.STATistic.AddressDISTance Time interval for single program event 335
<trace>.STATistic.AddressDURATION Time interval for two program events 336
<trace>.STATistic.BondOut Bondout mode 338
<trace>.STATistic.ChildTREE Show callee context of a function 338
<trace>.STATistic.COLOR Assign colors to function for colored graphics 339
<trace>.STATistic.CYcle Analyze cycle types 339
<trace>.STATistic.DatasYmbol Analyze pointer context numerically 342
<trace>.STATistic.DISTance Time interval for a single event 344
<trace>.STATistic.Distrib Distribution analysis 347
<trace>.STATistic.DURATION Time between two events 350
<trace>.STATistic.FIRST Start point for statistic analysis 353
<trace>.STATistic.Func Nesting function runtime analysis 355
<trace>.STATistic.FuncDURATION Statistic analysis of single function 380
<trace>.STATistic.FuncDURATIONInternal Statistic analysis of single function 381
<trace>.STATistic.GROUP Group run-time analysis 382
<trace>.STATistic.Ignore Ignore false records in statistic 383
<trace>.STATistic.INTERRUPT Interrupt statistic 384
<trace>.STATistic.INTERRUPTFunction Statistics interrupt processing 385
<trace>.STATistic.INTERRUPTTREE. Display interrupt nesting 386
<trace>.STATistic.LAST End point for statistic analysis 387
<trace>.STATistic.LINE HLL-line analysis 389
<trace>.STATistic.LINKage Per caller statistic of function 392
<trace>.STATistic.Measure Analyze the performance of a signal 394
<trace>.STATistic.MODULe Code execution broken down by module 396
<trace>.STATistic.PADDRESS Which instructions accessed data address 396
<trace>.STATistic.PARENTTREE Show the call context of a function 397
<trace>.STATistic.PREFetch Prefetch detection 398
<trace>.STATistic.PROGRAM Code execution broken down by program 399
<trace>.STATistic.PSYMBol Shows which functions accessed data address 399
<trace>.STATistic.SORT Specify sorting criterion for statistic commands 401
<trace>.STATistic.SYMBol Flat run-time analysis 409
<trace>.STATistic.TASK Task activity statistic 414

PowerTrace

Procedure for Measurement for TRACE32-ICE and TRACE32-FIRE 375
<trace>.STATistic.FUNCTION Statistic analysis of single function 380
<trace>.STATistic.FUNCTIONINTERNAL Statistic analysis of single function 381
<trace>.STATistic.GROUP Group run-time analysis 382
<trace>.STATistic.IGNORE Ignore false records in statistic 383
<trace>.STATistic.INTERRUPT Interrupt statistic 384
<trace>.STATistic.INTERRUPTFUNCTION Statistics interrupt processing 385
<trace>.STATistic.INTERRUPTTREE Display interrupt nesting 386
<trace>.STATistic.LAST End point for statistic analysis 387
<trace>.STATistic.LINE HLL-line analysis 389
<trace>.STATistic.LINKAGE Per caller statistic of function 392
<trace>.STATistic.MEASURE Analyze the performance of a signal 394
<trace>.STATistic.MODULe Code execution broken down by module 396
<trace>.STATistic.PADDRESS Which instructions accessed data address 396
<trace>.STATistic.PARENTTREE Show the call context of a function 397
<trace>.STATistic.PREFETCH Prefetch detection 398
<trace>.STATistic.PROGRAM Code execution broken down by program 399
<trace>.STATistic.PSYMBOL Shows which functions accessed data address 399
<trace>.STATistic.SORT Specify sorting criterion for statistic commands 401
<trace>.STATistic.SYMBOL Flat run-time analysis 409
<trace>.STATistic.TASK Task activity statistic 414
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;trace&gt;.STATistic.TASKFunc</td>
<td>Task specific function run-time analysis</td>
</tr>
<tr>
<td>Procedure for Measurement for TRACE32-ICD and PowerTrace</td>
<td>423</td>
</tr>
<tr>
<td>Procedure for Measurement for TRACE32-ICE and TRACE32-FIRE</td>
<td>424</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.TASKINTR</td>
<td>ISR2 statistic (ORTI)</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.TASKKernel</td>
<td>Task analysis with kernel markers (flat)</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.TASKORINTERRUPT</td>
<td>Statistic of interrupts and tasks</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.TASKSr</td>
<td>Analysis of time in OS service routines</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.TASKState</td>
<td>Performance analysis</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.TASKTREE</td>
<td>Tree display of task specific functions</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.TASKVSINTR</td>
<td>Statistic of interrupts, task-related</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.TREE</td>
<td>Tree display of nesting function run-time analysis</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.Use</td>
<td>Use records</td>
</tr>
<tr>
<td>&lt;trace&gt;.STATistic.Var</td>
<td>Statistic of variable accesses</td>
</tr>
<tr>
<td>&lt;trace&gt;.STREAMCompression</td>
<td>Select compression mode for streaming</td>
</tr>
<tr>
<td>&lt;trace&gt;.STREAMFILE</td>
<td>Specify temporary streaming file path</td>
</tr>
<tr>
<td>&lt;trace&gt;.STREAMFileLimit</td>
<td>Set size limit for streaming file</td>
</tr>
<tr>
<td>&lt;trace&gt;.STREAMLOAD</td>
<td>Load streaming file from disk</td>
</tr>
<tr>
<td>&lt;trace&gt;.STREAMSAVE</td>
<td>Save streaming file to disk</td>
</tr>
<tr>
<td>&lt;trace&gt;.TCount</td>
<td>Set trigger counter</td>
</tr>
<tr>
<td>&lt;trace&gt;.TDelay</td>
<td>Trigger delay</td>
</tr>
<tr>
<td>&lt;trace&gt;.TERMINation</td>
<td>Use trace line termination of preprocessor</td>
</tr>
<tr>
<td>&lt;trace&gt;.TestFocus</td>
<td>Test trace port recording</td>
</tr>
<tr>
<td>&lt;trace&gt;.TestFocusClockEye</td>
<td>Scan clock eye</td>
</tr>
<tr>
<td>&lt;trace&gt;.TestFocusEye</td>
<td>Check signal integrity</td>
</tr>
<tr>
<td>&lt;trace&gt;.THreshold</td>
<td>Optimize threshold for trace lines</td>
</tr>
<tr>
<td>&lt;trace&gt;.TimeStamp</td>
<td>Configure timestamp usage of LOGGER trace</td>
</tr>
<tr>
<td>&lt;trace&gt;.Timing</td>
<td>Waveform of trace buffer</td>
</tr>
<tr>
<td>&lt;trace&gt;.TMode</td>
<td>Select trigger mode</td>
</tr>
<tr>
<td>&lt;trace&gt;.TOT</td>
<td>Enable trigger output line (PowerIntegrator)</td>
</tr>
<tr>
<td>&lt;trace&gt;.TPreDelay</td>
<td>Pre-trigger delay</td>
</tr>
<tr>
<td>&lt;trace&gt;.TraceCLOCK</td>
<td>Set capture rate for single-wire-trace</td>
</tr>
<tr>
<td>&lt;trace&gt;.TraceCONNECT</td>
<td>Select on-chip peripheral sink</td>
</tr>
<tr>
<td>&lt;trace&gt;.TRACK</td>
<td>Set tracking record</td>
</tr>
<tr>
<td>&lt;trace&gt;.TRIGGER</td>
<td>Trigger the trace</td>
</tr>
<tr>
<td>&lt;trace&gt;.TSElect</td>
<td>Select trigger source</td>
</tr>
<tr>
<td>&lt;trace&gt;.SYNC</td>
<td>Select trigger line and mode</td>
</tr>
<tr>
<td>&lt;trace&gt;.TVIEW</td>
<td>Display trigger settings</td>
</tr>
<tr>
<td>&lt;trace&gt;.TWidth</td>
<td>Set trigger filter</td>
</tr>
<tr>
<td>&lt;trace&gt;.View</td>
<td>Display single record</td>
</tr>
<tr>
<td>&lt;trace&gt;.XTrack</td>
<td>Cross system tracking</td>
</tr>
<tr>
<td>&lt;trace&gt;.ZERO</td>
<td>Align timestamps of trace and timing analyzers</td>
</tr>
</tbody>
</table>

TRACEPORT

©1989-2020 Lauterbach GmbH
TRACEPORT Configure trace hardware 469
TRACEPORT.EndsKiP Define number of bytes skipped at the end of frame 470
TRACEPORT.LaneCount Select port size of the trace port 471
TRACEPORT.LaneSpeed Inform debugger about trace port frequency 471
TRACEPORT.MsgBitEndian Change bit-order within each byte 472
TRACEPORT.MsgByteEndian Change byte-order within each word 473
TRACEPORT.MsgLongEndian Change dword-order within each qword 473
TRACEPORT.MsgWordEndian Change word-order within each dword 474
TRACEPORT.PinReMap Adapt the lane order of the trace port 475
TRACEPORT.RefClock Set up reference clock for trace port 476
TRACEPORT.Reset Reset trace port configuration 476
TRACEPORT.StartsKiP Define number of bytes skipped at the start of frame 477
TRACEPORT.state Display trace port configuration window 478

TrAddress ........................................................................................................... 479
TrAddress Select output signal 479

TRANSlation ......................................................................................................... 482
TRANSlation Debugger address translation 482
Overview TRANSlation 482
TRANSlation.AutoEnable Auto-enable debugger MMU translation 485
TRANSlation.AutoSCAN Autoscan feature for debugger MMU 485
TRANSlation.CacheFlush Flush TRACE32 address translation cache 486
TRANSlation.CLEANUP Clean up MMU table 486
TRANSlation.COMMON Common address ranges for kernel and tasks 487
TRANSlation.COMMON.ADD Add another common address range 489
TRANSlation.COMMON.CLEAR Clear all common logical address ranges 489
TRANSlation.Create Create translation 490
TRANSlation.CreateID Add entry to MMU space ID table 491
TRANSlation.CreateTab Create multiple translations 491
TRANSlation.Delete Delete translation 492
TRANSlation.DeleteID Remove entry from MMU space ID table 492
TRANSlation.List List MMU translation table 493
TRANSlation.ListID List MMU space ID table 494
TRANSlation.NoProtect Unprotect memory 494
TRANSlation.OFF Deactivate debugger address translation 495
TRANSlation.ON Activate debugger address translation 495
TRANSlation.PAGER Allow paged breakpoints for Linux 496
TRANSlation.Protect Protect memory 497
TRANSlation.Protect.ADD Add range to protected memory ranges 497
TRANSlation.Protect.OFF Switch protection of target memory off 498
TRANSlation.Protect.ON Protect entire target memory 499
TRANSlation.RESET Reset MMU configuration 500
TRANSlation.SCANall Scan MMU address space tables from kernel 500
TRANSlation.ScanID Scan MMU tables 500
TRANSlation.SHADOW

Enable shadow access to target memory

TRANSlation.state

Overview of translation settings

TRANSlation.TableWalk

Automatic MMU page table walk

TRANSlation.TlbAutoScan

Allow automatic TLB scans during table walk

TRANSlation.TRANSparent

Transparent banking area

TrBus ................................................................................................................................. 507

TrBus

Trigger bus

Overview TrBus

Trigger Bus on the PowerTrace

Interaction Between Independent PODBUS Devices

TrBus.Arm

Arm the trigger bus

TrBus.Connect

Configure TRIGGER as input or output

TrBus.Mode

Define polarity/edge for the trigger signal

TrBus.OFF

Switch trigger bus off

TrBus.Out

Define source for the external trigger pulse

TrBus.RESet

Reset setting for trigger bus

TrBus.Set

Define the target for the incoming trigger

TrBus.state

Display settings for the trigger bus

TrBus.Trigger

Stimulate a trigger on the trigger bus

TrEvent .................................................................................................................. 517

TrEvent

Event trigger system

Overview TrEvent

TrEvent.Delay

Define delay

TrEvent.Enable

Select time windows

TrEvent.Init

Initialization

TrEvent.MinInit

Initialization

TrEvent.Mode

Select operation mode

TrEvent.OFF

Switch off

TrEvent.ON

Switch on

TrEvent.RESet

Reset command

TrEvent.Select

Select event source

TrEvent.view

State display

TrIn .................................................................................................................. 528

TrIn

Internal trigger logic

Overview TrIn

TrIn.Clock

Define clock

TrIn.Data

Define data

TrIn.Mask

Define bits

TrIn.Normal

Level operation

TrIn.RESet

Reset command

TrIn.state

State display

TrIn.Transient

Transient operation
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrMain</td>
<td>Trigger system of TRACE32-ICE</td>
<td>537</td>
</tr>
<tr>
<td>Overview</td>
<td>TrMain</td>
<td>537</td>
</tr>
<tr>
<td>TrMain.ALways</td>
<td>Constant triggering</td>
<td>540</td>
</tr>
<tr>
<td>TrMain.Arm</td>
<td>Release and activate trigger system</td>
<td>540</td>
</tr>
<tr>
<td>TrMain.AutoInit</td>
<td>Automatic trigger initialization</td>
<td>540</td>
</tr>
<tr>
<td>TrMain.AutoStart</td>
<td>Automatic trigger initialization</td>
<td>541</td>
</tr>
<tr>
<td>TrMain.Break</td>
<td>Break</td>
<td>541</td>
</tr>
<tr>
<td>TrMain.Count</td>
<td>Set trigger counter</td>
<td>542</td>
</tr>
<tr>
<td>TrMain.Delay</td>
<td>Set trigger delay</td>
<td>542</td>
</tr>
<tr>
<td>TrMain.Init</td>
<td>Initialize trigger system</td>
<td>543</td>
</tr>
<tr>
<td>TrMain.Mode</td>
<td>Select mode</td>
<td>543</td>
</tr>
<tr>
<td>TrMain.OFF</td>
<td>Switch off trigger system</td>
<td>544</td>
</tr>
<tr>
<td>TrMain.Out</td>
<td>Output trigger pulse</td>
<td>544</td>
</tr>
<tr>
<td>TrMain.RESet</td>
<td>Reset trigger system</td>
<td>545</td>
</tr>
<tr>
<td>TrMain.Set</td>
<td>Select trigger sources</td>
<td>546</td>
</tr>
<tr>
<td>TrMain.state</td>
<td>Trigger state display</td>
<td>551</td>
</tr>
<tr>
<td>TrMain.Trigger</td>
<td>Trigger state display</td>
<td>551</td>
</tr>
<tr>
<td>TrOnchip</td>
<td>Onchip triggers</td>
<td>552</td>
</tr>
<tr>
<td>TrOnchip.CONVert</td>
<td>Adjust range breakpoint in onchip registers</td>
<td>552</td>
</tr>
<tr>
<td>TrOnchip.RESet</td>
<td>Reset settings to defaults</td>
<td>553</td>
</tr>
<tr>
<td>TrOnchip.Set</td>
<td>Break on event</td>
<td>554</td>
</tr>
<tr>
<td>TrOnchip.state</td>
<td>Display onchip trigger window</td>
<td>555</td>
</tr>
<tr>
<td>TrPOD</td>
<td>Trigger probe</td>
<td>557</td>
</tr>
<tr>
<td>TrPOD.Clock</td>
<td>Defines data mask</td>
<td>557</td>
</tr>
<tr>
<td>TrPOD.ClockPOL</td>
<td>Defines data polarity</td>
<td>557</td>
</tr>
<tr>
<td>TrPOD.Data</td>
<td>Defines data mask</td>
<td>558</td>
</tr>
<tr>
<td>TrPOD.DataPOL</td>
<td>Defines data polarity</td>
<td>558</td>
</tr>
<tr>
<td>TrPOD.Mode</td>
<td>Defines data polarity</td>
<td>559</td>
</tr>
<tr>
<td>TrPOD.OFF</td>
<td>Switch off</td>
<td>560</td>
</tr>
<tr>
<td>TrPOD.ON</td>
<td>Switch on</td>
<td>560</td>
</tr>
<tr>
<td>TrPOD.RESet</td>
<td>Reset command</td>
<td>560</td>
</tr>
<tr>
<td>TrPOD.state</td>
<td>State display</td>
<td>561</td>
</tr>
<tr>
<td>TrPOD.Time</td>
<td>Defines the time for the pulse width trigger</td>
<td>562</td>
</tr>
</tbody>
</table>
History

20-Dec-19  Description of the commands <trace>.Chart.Var and <trace>.STATistic.Var.

11-Dec-19  Description of the commands <trace>.Chart.ChildTREE, <trace>.Chart.MODULE, <trace>.Chart.PROGRAM and <trace>.PROfileChart.MODULE.

25-Jun-19  Description of the new trace method NONE in <trace>.METHOD NONE and “What to know about the TRACE32 default settings for Trace.METHOD”.

18-Mar-19  New command TASK.ORTI.SPLITSTACK. New section “What to know about the Machine Parameters”.

27-Feb-19  Updated the commands TRANSlation.ON, TRANSlation.OFF, TRANSlation.TableWalk, and TRANSlation.NoProtect.

22-Feb-19  Description of the command group TRANSlation.Protect.

13-Aug-18  Description of the new command TargetSystem.NewInstance.

10-Aug-18  The channel number was added to the command syntax of most TERM commands. The channel number allows the concurrent use of several terminal windows.

03-Aug-18  Link to “Application Note for the Trace.DRAW Command” (app_trace_draw.pdf) added to Trace.DRAW.Data and Trace.DRAW.Var command. Related to this, a clean-up for both commands was performed.

10-Jul-18  Description of the command TRANSlation.state.

08-Mar-18  Description for command Trace.ListVar updated.

25-Jan-18  Syntax description and examples for commands Trace.DRAW.Data and Trace.DRAW.Var updated.
Using the command group **TargetSystem**, you can start new TRACE32 PowerView instances from within a running instance and keep an overview of these instances.

The instances started with **TargetSystem.NewInstance** are automatically connected to the same PowerDebug hardware module or to the same MCI Server as the instance that initiated the start process. (In case of the MCI Server, the setting in the config file is: \( \text{PBI}=\text{MCISERVER} \)).

**NOTE:** The **TargetSystem.NewInstance** command is not available for:
- The TRACE32 Instruction Set Simulator (\( \text{PBI}=\text{SIM} \) in the config file)
- The debuggers connected to the target via the GDI interface (\( \text{PBI}=\text{GDI} \))
- The debuggers connected to the target via the MCD interface (\( \text{PBI}=\text{MCD} \))

The **TargetSystem.state** window provides an overview of the status of the cores assigned to the various TRACE32 instances. The window also helps you keep an overview of the synchronization mechanism between the TRACE32 instances, which is set up with the **SYnch** command group.

In addition, the **TargetSystem.state** window displays the InterCom names and UDP port numbers used by the instances for communication with each other via the **InterCom** system.

See also
- **TargetSystem.NewInstance**
- **TargetSystem.state**
- **SYnch**
- **InterCom**
TargetSystem.NewInstance  
Start new TRACE32 PowerView instance

<table>
<thead>
<tr>
<th>Format:</th>
<th>TargetSystem.NewInstance &lt;intercom_name&gt; [/&lt;option&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;:</td>
<td>ARCHitecture &lt;arch&gt;</td>
</tr>
<tr>
<td>&lt;arch&gt;:</td>
<td>8051</td>
</tr>
<tr>
<td>&lt;index&gt;:</td>
<td>1. ... 254.</td>
</tr>
<tr>
<td>&lt;index_min&gt;:</td>
<td>1. ... 254.</td>
</tr>
</tbody>
</table>

Allows a TRACE32 PowerView instance to start new TRACE32 PowerView instances (max. 15 new instances) for debugging AMP systems. In AMP (asynchronous multiprocessing) systems, each TRACE32 PowerView instance is responsible for an SMP subsystem or single core. For more information, see CORE.ASSIGN.

All instances started with TargetSystem.NewInstance are automatically connected to the same PowerDebug hardware module or the same MCI Server (PBI=MCISERVER in the config.t32 file) as the instance that initiated the start process.

The instance that starts another instance clones the current config file (by default config.t32) and extends the cloned file for the new instance.

**NOTE:** The TargetSystem.NewInstance command is not available for:
- The TRACE32 Instruction Set Simulator (PBI=SIM in the config file)
- The debuggers connected to the target via the GDI interface (PBI=GDI)
- The debuggers connected to the target via the MCD interface (PBI=MCD)

<table>
<thead>
<tr>
<th>&lt;intercom_name&gt;</th>
<th>Assigns a user-defined InterCom name to the new TRACE32 instance.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHitecture &lt;arch&gt;</td>
<td>Selects the architecture of the new TRACE32 instance. If the ARCHitecture option is omitted, then a TRACE32 instance of the same architecture will be started.</td>
</tr>
<tr>
<td>Parameter Type:</td>
<td>Decimal value.</td>
</tr>
<tr>
<td>APIPORT &lt;port_number&gt;</td>
<td>Passes a UDP remote API &lt;port_number&gt; to the new TRACE32 instance.</td>
</tr>
</tbody>
</table>
Examples

Example 1: This script shows how to start a second TRACE32 instance named `mySecondInstance` from within the current TRACE32 instance.

```plaintext
TargetSystem.NewInstance mySecondInstance /ARCHitecture ARM64
InterCom.execute mySecondInstance PRINT "started by the first instance"
```

Example 2: Let's assume you have started a number of instances and now want to quit a particular instance. This script shows how to quit a TRACE32 instance named `mySecondInstance` in a set of TRACE32 instances.

```plaintext
InterCom.execute mySecondInstance QUIT
```

See also
- TargetSystem
- TargetSystem.state
- InterCom.Enable

▲ 'Release Information' in 'Release History'
Opens the TargetSystem.state window, providing an overview of the multicore system configuration and state across multiple TRACE32 instances sharing one PowerDebug hardware module or MCI Server. The indices on the first and second level are configured using SYStem.CONFIG.CORE <chip> <core>. The indices on the third level indicate the thread index of the SMP system that can be defined by CORE.ASSIGN or CORE.NUMber.

The TargetSystem window is not available for front-end debuggers.

To illustrate the TargetSystem.state command, the following use cases are provided:

- **Use case 1**: Diagnostic tool for the target system structure
- **Use case 2**: TRACE32 instance selector
- **Use case 3**: Manage the SYnch settings for all TRACE32 instances
### <columns> - Description of Columns in the TargetSystem.state Window

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DEFAult</strong></td>
<td>Adds TargetSystem, CoreType and CoreState column. If no column is passed DEFAult is used automatically.</td>
</tr>
<tr>
<td><strong>ALL</strong></td>
<td>Displays all available columns in the TargetSystem.state window.</td>
</tr>
<tr>
<td><strong>TargetSystem</strong></td>
<td>Adds the TargetSystem column to show a hierarchical view on the system. If the column is left out, it will be added automatically. The parameter is used to tell the dialog that the DEFAult option is not active and only the TargetSystem column shall be shown.</td>
</tr>
<tr>
<td><strong>CoreType</strong></td>
<td>Adds a column to show the target architecture of a core and core family name if available.</td>
</tr>
<tr>
<td><strong>CoreState</strong></td>
<td>Shows the state of the core. The state can be system down (gray color), power down (red color), reset (red color), stopped (bold) or running. The running state can be extended by an attribute that indicates a run mode e.g. “no core clock”.</td>
</tr>
<tr>
<td><strong>Title</strong></td>
<td>Adds a column with the corresponding window title. The title can be set by the configuration file before start-up or by the TITLE command.</td>
</tr>
</tbody>
</table>
| **InterComPort** | Adds a column with the InterCom UDP port numbers of TRACE32 instances. The InterCom port numbers are used by the InterCom commands and the SYnch commands.  
You can assign a new port number by double-clicking a port number in the ic port column. For an illustrated example, see InterCom.PORT. |
| **InterComName** | Adds a column with the InterCom names of TRACE32 instances. Names are created with the commands InterCom.NAME or InterCom.ENable. The names can then be used as arguments in InterCom and SYnch commands.  
You can rename an instance by double-clicking a name in the ic name column. For an illustrated example, see InterCom.NAME. |
| **INSTance** | Adds a column, showing the value of INSTANCE= from the config file.  
If INSTANCE= is missing in the config file, then 1 is displayed by default. That is, in this case the display value is equivalent to the explicit setting INSTANCE=1 in the config file. |
| **UseCore** | Adds a column, showing the value of CORE= from the config file.  
If CORE= is missing in the config file, then 1 is displayed by default. That is, in this case the display value is equivalent to the explicit setting CORE=1 in the config file.  
See also SYStem.USECORE(). |
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYnch.All</td>
<td>Adds the columns <code>SYnch.Go</code>, <code>SYnch.Step</code>, <code>SYnch.Break</code> and <code>SYnch.SystemMode</code>.</td>
</tr>
<tr>
<td>SYnch.Go</td>
<td>Adds the column to indicate and edit the <code>SYnch.MasterGo</code> and <code>SYnch.SlaveGo</code> setting. The header of the column is named SG.</td>
</tr>
<tr>
<td>SYnch.Step</td>
<td>Adds the column to indicate and edit the <code>SYnch.MasterStep</code> and <code>SYnch.SlaveStep</code> setting. The header of the column is named SS.</td>
</tr>
<tr>
<td>SYnch.Break</td>
<td>Adds the column to indicate and edit the <code>SYnch.MasterBreak</code> and <code>SYnch.SlaveBreak</code> setting. The header of the column is named SB.</td>
</tr>
<tr>
<td>SYnch.System-Mode</td>
<td>Adds the column to indicate and edit the <code>SYnch.MasterSystemMode</code> and <code>SYnch.SlaveSystemMode</code> setting. The header of the column is named SM.</td>
</tr>
</tbody>
</table>

<options> - Options for the TargetSystem.state Window

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Don’t highlight specific information for the TRACE32 instance from where the dialog was opened. The dialog can be moved outside of the main window and used to act as an independent window to bring a certain instance to foreground by a double click to of an entry of the TargetSystem tree column.</td>
</tr>
<tr>
<td>UseTitle</td>
<td>Use the TRACE32 window title as name for an SMP Subsystem or Core. The title can be set by the configuration file before start-up or by the PRACTICE command TITLE.</td>
</tr>
<tr>
<td>UseICName</td>
<td>Use the TRACE32 InterCom name as window title for an SMP subsystem or core. The InterCom name can be set with the InterCom.NAME command.</td>
</tr>
</tbody>
</table>
Use case 1: Diagnostic tool for the target system structure

The command opens the window showing the overall system. Nodes that belong to this TRACE32 instance are displayed in bold. A double-click to a thread selects this thread to be active.

TargetSystem.state CoreType /UseTitle

Use case 2: TRACE32 instance selector

The command opens the window showing the overall system and the state of the particular cores. The window can be moved outside of the TRACE32 instance where the command was executed. A double-click at an SMP system node or core will bring the assigned instance to foreground.

TargetSystem.state CoreState /UseTitle /Global
Use case 3: Manage the SYNch settings for all TRACE32 instances

The command opens the window showing the overall system and the SYNch settings.

```
TargetSystem.state SYNch.All /UseTitle /Global
```

A single click at an entry in one of the columns will change the setting in the SYNch dialog and set the connection ports.

<table>
<thead>
<tr>
<th>default</th>
<th>Neither master nor slave option is set.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st click M</td>
<td>master option set.</td>
</tr>
<tr>
<td>2nd click S</td>
<td>slave option is set.</td>
</tr>
<tr>
<td>3rd click MS</td>
<td>master and slave option is set.</td>
</tr>
</tbody>
</table>

See also

- TargetSystem
- SYNch
- SYNch.MasterSystemMode
- SYNch.SlaveSystemMode
- InterCom.execute
- TargetSystem.NewInstance
- SYNch.Connect
- SYNch.SlaveBreak
- SYNch.state
- CORE.ASSIGN
- SYNch.MasterGo
- SYNch.SlaveGo
- SYNch.SlaveStep
- SYNch.SlaveSystemMode
- SYNch.MasterSystemMode
- SYNch.state
- SYStem.CONFIG.CORE
- InterCom
Overview TASK

This chapter describes the OS Awareness features (aka kernel awareness), generic to all processors and kernels. Kernel specific features are described in additional manuals, see OS Awareness Manuals.

The OS Awareness may support the following main features:

- Display of kernel resources (e.g. tasks, queues, semaphores, messages).
- Task stack coverage.
- Task related breakpoints.
- Task context display.
- Manual execution of system calls.
- Operating system's MMU support.
- Dynamic task performance measurement
- Task runtime statistics and flowchart display out of the trace buffer. Display of task switches in the trace listing.
- Task state statistics and time chart out of the trace buffer, i.e. show how long each task is in a certain state (running, ready, etc.).
- Task-related function runtime statistics, flowchart display and function nesting display out of the trace buffer.
- Display system calls with parameters in the trace listing.
- Fast access to the features through dedicated menus.
Not all features are implemented for all processors and kernels. Please see the kernel specific manual for a detailed description of the supported features.

**OS Awareness Configurations**

The OS Awareness is configured by the `TASK.CONFIG` command. The command loads a configuration file that tells the debugger all kernel-related information. It can be adopted to any (RT)OS kernel. Lauterbach provides ready-to-start configuration files for a wide range of real-time operating systems. If you want to adapt it to your own proprietary kernel, ask Lauterbach for assistance.

**What to know about the Task Parameters**

In TRACE32, operating system tasks (short: tasks) can be identified based on one of these values:

- Task magic number
- Task ID
- Task name

For OS-aware debugging and tracing, these three values are displayed in the `TASK.List.tasks` window and can be returned with the functions `TASK.MAGIC()`, `TASK.ID()`, and `TASK.NAME()`. In addition, the three values can be passed as parameters to task-related TRACE32 commands and options.

<table>
<thead>
<tr>
<th>NOTE:</th>
<th>In case of the <code>TASK.CONFIG</code> command, you will encounter the parameter <code>&lt;magic_address&gt;</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <code>&lt;task_magic&gt;</code> and <code>&lt;magic_address&gt;</code> are <strong>not</strong> the same.</td>
</tr>
<tr>
<td></td>
<td>• For information about <code>&lt;magic_address&gt;</code>, see <code>TASK.CONFIG</code> command.</td>
</tr>
</tbody>
</table>

**Task Magic Number**

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

<table>
<thead>
<tr>
<th><code>&lt;task_magic&gt;</code></th>
<th>Parameter Type: Hex value.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Example: <code>TASK.select 0xFFFFF7B040</code></td>
</tr>
</tbody>
</table>

**Task ID**

This value refers to the numeric task ID as given by the operating system. If the OS does not provide a task ID, this option may not be available.

<table>
<thead>
<tr>
<th><code>&lt;task_id&gt;</code></th>
<th>Parameter Type: Decimal value.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Example: <code>TASK.select 1546</code></td>
</tr>
</tbody>
</table>
**Task Name**

This string refers to the task name as given by the operating system. If the OS does not provide a task name, this option may not be available.

If the task runs in a system involving virtualization, then the task name can be preceded with the machine name.

<table>
<thead>
<tr>
<th>&lt;task_name&gt;</th>
<th>Parameter Type: String.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 1: TASK.select &quot;adbd:1546&quot;</td>
<td></td>
</tr>
<tr>
<td>Example 2: TASK.select &quot;FreeRTOS:::SieveDemo&quot;</td>
<td></td>
</tr>
</tbody>
</table>

FreeRTOS is the name of the machine. The three colons ::: serve as the separator between the machine name and the task name SieveDemo.

---

### What to know about the Machine Parameters

In hypervisor-based environments, TRACE32 identifies machines based on one of these values:

- Machine magic number
- Machine ID
- Machine name

For hypervisor debugging and tracing, these three values are displayed in the TASK.List.MACHINES window. In addition, the three values can be passed as parameters to machine-related TRACE32 commands and options.

#### Machine Magic Number

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

<table>
<thead>
<tr>
<th>&lt;machine_magic&gt;</th>
<th>Parameter Type: Hex value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: machine magic number &gt; 0xFF</td>
<td></td>
</tr>
</tbody>
</table>

Machine magic numbers are displayed, for example, in the magic column of the TASK.List.MACHINES window as hex values.
Machine ID

A **machine ID** is a numeric identifier which extends a logical address and intermediate physical address in TRACE32 or can be used together with the option **MACHINE** in some TRACE32 commands. The purpose of a machine ID is to identify guest machines within a system that is using a hypervisor to run multiple virtual machines.

| <machine_id> | Parameter Type: **Decimal** or **hex value**.  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Range: 0x0 &lt;= machine ID &lt; 0x1F</td>
</tr>
<tr>
<td></td>
<td>Machine IDs are displayed, for example, in the <strong>mid</strong> column of the <strong>TASK.List.MACHINES</strong> window as decimal values (1., 2., etc.)</td>
</tr>
</tbody>
</table>

In TRACE32, the machine ID clearly specifies which virtual machine (a guest machine or the host machine) an address belongs to:

- The machine ID 0 (zero) is always associated with the host machine running the hypervisor.
- All the other machine IDs >= 1 are associated with the guest machines.

**Format of addresses with machine IDs:**

In the TRACE32 address format, the machine ID is always in the leading position, directly after the access class specifier. The machine ID is followed a triple colon (:::) to separate the machine ID from the remaining parts of an address. The format of a TRACE32 address containing a machine ID looks like this:

- **Without space ID:**
  <access_class>:<machine_id>::<address_offset>

- **With space ID:**
  <access_class>:<machine_id>::<space_id>::<address_offset>

**Examples:**

- **Without space ID:**
  - G:0x1::0x80000000
  - 0x2::0xA0000000

- **With space ID:**
  - G:0x3::0x020A::0x80000000
  - G:0x0::0x0::0x4000C000
  - 0x2::0x170::0x1F000000

**Notes:**

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

A machine name is a meaningful string that allows users to identify a host or guest machine in a hypervisor-based environment. The machine name is given by the Hypervisor Awareness. If the Hypervisor Awareness does not provide a machine name, you can assign a name to a machine by using the NAME option of the EXTension.LOAD command. Without the NAME option, the base name of the extension definition file will be used.

In a hypervisor-based environment, the machine name precedes the task name.

<table>
<thead>
<tr>
<th>&lt;machine_name&gt;</th>
<th>Parameter Type: String.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example: TASK.select &quot;FreeRTOS:::SieveDemo&quot;</td>
<td></td>
</tr>
<tr>
<td>FreeRTOS is the name of the machine. The three colons ::: serve as the separator between the machine name and the task name SieveDemo.</td>
<td></td>
</tr>
</tbody>
</table>

Glossary

For important OS Awareness and Hypervisor Awareness terms, such as task, thread, process, machine, kernel, MMU space, and virtual machine, refer to the “TRACE32 Glossary” (glossary.pdf).
**TASK.ACCESS**

Control memory access

```
Format: TASK.ACCESS [<class>]
```

Defines the memory access class used by TASK related windows.

TASK related windows may access the target memory (e.g. when reading task control blocks). If the access class is set to E:, the debugger uses emulation memory access to read the memory (e.g. emulation memory, shadow memory or pseudo-dual-port access). If set to C:, the debugger uses CPU access. If the appropriate access is not possible, the window is temporarily frozen.

**TASK.ACCESS** without parameter enables the default mode, which uses E: if the application is running, and C: if the application is stopped.

Please see also the manuals for a description of E: and C: to your processor.

- Processor Architecture Manuals
- Target Guides FIRE
- Target Guides ICE

---

**TASK.Break**

*ICE only*

```
Format: TASK.Break <task> (deprecated)
```

It was used in 68k ICE only for task selective debugging. **TASK.Break** marks a task of the selective list to be breaked at the next entry.

---

©1989-2020 Lauterbach GmbH
TASK.CACHEFLUSH

ICE only

Format: TASK.CACHEFLUSH

Usually not needed. Use only if advised to do so.

The debugger reads out the task list of the target at each single step or Go/Break sequence, and stores the list internally (see TASK.List.tasks). If the task list or task characteristics change while the target is halted, a manual update of the task list may be necessary. This command forces an immediate re-evaluation of the task list.

See also

■ TASK

TASK.CONFIG

Configure OS Awareness

Format: TASK.CONFIG <os_awareness_file> <magic_address> <args> [/<option>]

<option>: ACCESS <class>

Configures the OS Awareness using a given configuration file. Please refer to the OS-specific manual. See OS Awareness Manuals.

Arguments:

<os_awareness_file> File name of the configuration file.

<magic_address> Address of the memory location holding the task magic number of the currently running task. See “What to know about the Task Parameters”, page 24.

<args> All other arguments are interpreted by the configuration file. Details of predefined files are described in the kernel-specific part of an OS Awareness Manual.
OPTIONS:

ACCESS

Defines the memory access class used by TASK-related windows. See TASK.ACCESS.

See also

- TASK
- EXTension LOAD
- MMU

▲ 'Configuration' in 'Bootloader Awareness Manual coreboot'
▲ 'Configuration' in 'Bootloader Awareness Manual GRUB'
▲ 'Release Information' in 'Release History'
▲ 'Configuration' in 'OS Awareness Manual FreeRTOS'
▲ 'Configuration' in 'OS Awareness Manual HIT7000'
▲ 'Configuration' in 'OS Awareness Manual NORTI'
▲ 'Configuration' in 'OS Awareness Manual OKL4'
▲ 'Configuration' in 'OS Awareness Manual PrKERNEL'
▲ 'Configuration' in 'OS Awareness Manual RTXC Quadros'
▲ 'Configuration' in 'OS Awareness Manual REALOS'
▲ 'Configuration' in 'OS Awareness Manual RTEMS'
▲ 'Configuration' in 'OS Awareness Manual RTX-ARM'
▲ 'Configuration' in 'OS Awareness Manual MicroC3/Compact'
▲ 'Configuration' in 'OS Awareness Manual MicroC3/Standard'
▲ 'Configuration' in 'OS Awareness Manual Zephyr'
▲ 'Configuration' in 'UEFI Awareness Manual BLDK'
▲ 'Configuration' in 'UEFI Awareness Manual TianoCore'

TASK.COPYDOWN

Copy file from host into target

Format:

    TASK.COPYDOWN <source_file_host> <destination_file_target>

Copies a file from the host into the target. Only supported for Linux and QNX run mode debugging.

See also

- TASK

▲ 'Commands for Run Mode Debugging' in 'Run Mode Debugging Manual Linux'
▲ 'Commands for Run Mode Debugging' in 'Run Mode Debugging Manual QNX'
**TASK.COPYUP**

Copy file from target into host

Format:

```
TASK.COPYUP <source_file_target> <destination_file_host>
```

Copies a file from the target into the host. Only supported for Linux and QNX run mode debugging.

**See also**

- TASK
- ‘Commands for Run Mode Debugging’ in ‘Run Mode Debugging Manual Linux’
- ‘Commands for Run Mode Debugging’ in ‘Run Mode Debugging Manual QNX’

---

**TASK.CreateExtraID**

Create a virtual task

Format:

```
TASK.CreateExtraID <task_name> <task_id> <space_id> <trace_id>
```

Creates a virtual task ID for trace analysis. Trace analysis will use the given task ID for task identification rather than the task magic number. Only for some dedicated applications.

**See also**

- TASK

---

**TASK.CreateID**

Create virtual task

Format:

```
TASK.CreateID <task_name> <task_id> <space_id> <trace_id>
```

Creates a virtual task name for trace analysis. Trace analysis will use the given task name for task identification, rather than the task magic. Only for some dedicated applications.

**See also**

- TASK
### TASK.Debug

**ICE only**

**Deprecated**

It was used in 68k ICE only for task selective debugging. TASK.Debug marks a task of the selective list as active for debugging.

See also
- TASK
- 'LynxOS Commands' in 'OS Awareness Manual LynxOS'

### TASK.DeleteID

**Delete virtual task**

Delete a virtual task created with TASK.CreateID or TASK.CreateExtraID.

See also
- TASK

### TASK.DETACH

**Debugger only**

**Detach from task**

Requests the debug agent to detach from the process <id>.

Only applicable if GDB (Linux) is used as debug agent.

**Example:**

```
TASK.DETACH 41.
```

See also
- TASK
- 'Commands for Run Mode Debugging' in 'Run Mode Debugging Manual QNX'
**TASK.INSTALL**

Deprecated

Format: TASK.INSTALL (deprecated)

See also
- TASK

**TASK.KILL**

End task

Debugger only

Format: TASK.KILL <id>

Request the debug agent to end the process <id>.

Only applicable if GDB (Linux) or TRK (Symbian) is used as debug agent.

**Example:**

```
TASK.KILL 41.
```

See also
- TASK
- 'Commands for Run Mode Debugging' in 'Run Mode Debugging Manual Linux'
- 'Commands for Run Mode Debugging' in 'Run Mode Debugging Manual QNX'
The windows of the **TASK.List** command group provide information about processes, space IDs, MMU spaces, machines, and tasks known to the debugger in an RTOS and hypervisor environment. The debugger needs a so-called “awareness” of the RTOS or hypervisor to be able to read out these items from the target.

See also

- **TASK.List.MACHINES**
- **TASK.List.SPACES**
- **TASK. List.tasks**
- **TASK.List.TREE**

### TASK.List.MACHINES

**List machines**

Lists information about all machines known to the debugger. Machines refer to virtual machines in a hypervisor environment. The hypervisor itself is listed as machine with ID 0.

Machines are only available if **SYStem.Option MACHINESPACES** is set to **ON**.

For several purposes, the debugger needs to know which machines are active in the system. The debugger uses the hypervisor specific awareness to read out all machine characteristics that it needs for its operation. **TASK.List.MACHINE** shows the machine characteristics that the debugger uses.

<table>
<thead>
<tr>
<th>magic</th>
<th>Machine magic number. Unique number for the machine. Usually the address of the control block structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the object, if available.</td>
</tr>
<tr>
<td>mid</td>
<td>Machine ID if a hypervisor system is set up.</td>
</tr>
</tbody>
</table>
TASK.List.SPACES

**Description of Columns in the TASK.List.SPACES Window:**

<table>
<thead>
<tr>
<th>access</th>
<th>Access class that an awareness uses for this machine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>vttb</td>
<td>“Virtual translation table base” address of this machine. The VTTB address points to the MMU table of the guest physical (= intermediate) address to host physical address translation.</td>
</tr>
<tr>
<td>extension(s)</td>
<td>Extensions loaded for this machine (EXTension.LOAD).</td>
</tr>
</tbody>
</table>

**See also**
- TASK.List
- TASK.List.tasks
- "Release Information" in "Release History"

**TASK.List.SPACES**

List MMU spaces

Lists all MMU spaces known to the debugger. MMU spaces usually refer to processes in an OS/RTOS environment. MMU spaces are only available if SYStem.Option MMUSPACES is set to ON.

For several purposes, the debugger needs to know which MMU spaces are active in the system. The debugger uses the kernel specific awareness to read out all space characteristics that it needs for its operation. TASK.List.SPACES shows the space characteristics that the debugger uses.

Each kernel specific awareness has a different display command to show the active processes with the characteristics that are essential to the specific kernel. Please see the appropriate OS Awareness Manual (rtos.<os>.pdf) for this command.

A The MMU space that is currently active on the selected core is marked.

**Description of Columns in the TASK.List.SPACES Window:**

<table>
<thead>
<tr>
<th>magic</th>
<th>Space magic number. Unique number for the space. Usually the address of the control block structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the object, if available.</td>
</tr>
<tr>
<td>id</td>
<td>ID of the object, if available.</td>
</tr>
</tbody>
</table>
### TASK.List.tasks

#### List all running tasks

**Format:**

<table>
<thead>
<tr>
<th>magic</th>
<th>Task magic number. Unique number for the task. Usually the address of the control block structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the object, if available.</td>
</tr>
<tr>
<td>id</td>
<td>ID of the object, if available.</td>
</tr>
<tr>
<td>space</td>
<td>Space name or ID if the OS uses MMU spaces.</td>
</tr>
<tr>
<td>traceid</td>
<td>ID that identifies an object in the trace list.</td>
</tr>
</tbody>
</table>

Lists all tasks known to the debugger. Additional information about machines and MMU spaces is only displayed if `SYStem.Option MMUSPACES` and `SYStem.Option MACHINESPACES` are set to `ON`.

For several purposes, the debugger needs to know which tasks are active in the system. The debugger uses the kernel specific awareness to read out all task characteristics that it needs for its operation. `TASK.List.tasks` shows the task characteristics that the debugger uses.

Each kernel specific awareness has a different display command to show the active tasks with the characteristics that are essential to the specific kernel. Please see the appropriate OS Awareness Manual (rtos_<os>.pdf) for this command.

A The task that is currently running on the selected core is marked.

**Description of Columns in the TASK.List.tasks Window:**
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td>Identifies in SMP systems at which core this task runs.</td>
</tr>
<tr>
<td>sel</td>
<td>Task selected for debugging (only in Run Mode Debugging).</td>
</tr>
<tr>
<td>stop</td>
<td>Task selected to stop on break (only in Run Mode Debugging).</td>
</tr>
<tr>
<td>machine</td>
<td>Machine name or machine ID if a hypervisor system is set up.</td>
</tr>
</tbody>
</table>

See also

- TASK.List
- TASK.List.MACHINES
- TASK.List.SPACES
- TASK.List.TREE
- ‘Commands for Run Mode Debugging’ in ‘Run Mode Debugging Manual Linux’
- ‘Commands for Run Mode Debugging’ in ‘Run Mode Debugging Manual QNX’
- ‘Commands for Run Mode Debugging’ in ‘Run Mode Debugging Manual Symbian’
- ‘Specific Commands’ in ‘Native Process Debugger’
Displays machines, MMU spaces, and tasks in the form of a tree structure.

A Level 1 of the tree: Machines.
B Level 2: MMU spaces.
C Level 3: Tasks.
D Yellow lines: The machine, the MMU space, and the task that are currently running on the selected core are marked.

**Description of Columns in the TASK.List.TREE Window**

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>magic</td>
<td>Magic number. Unique number for each object (machine/MMU space/task). Usually the address of the control block structure.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the object, if available.</td>
</tr>
</tbody>
</table>

**See also**
- TASK.List
- TASK.List.tasks
- 'Release Information' in 'Release History'

**TASK.ListID**

List virtual tasks

**Format:** TASK.ListID

Opens the TASK.ListID window, displaying virtual tasks created with TASK.CreateID or TASK.CreateExtraID.

**See also**
- TASK
This command is for internal use only and should not be used. TASK.MTXTRC shows a trace of the OS Awareness hyper processor activity.

See also
- TASK

Format: TASK.MTXTRC (deprecated)
Several windows of the OS Awareness show task-related information, e.g. TASK.STack or Trace.Chart.TASK. Internally, the OS Awareness always uses the task magic numbers to identify a task. When displaying the task-related information, the debugger can translate this task magic number into a more readable task name, using a task name translation table. If the debugger finds an entry with the appropriate task magic number, it shows the task name instead of the task magic number (or task ID).

The translation table can be populated manually or automatically. If the TASK configuration file supports it, the debugger automatically populates the table with the current available task magic numbers and their names. Additionally, or if no configuration file exists, or if the configuration doesn’t support task names, table entries may be added manually. If a manual entry and an automatic entry have the same task magic number, the manual entry overwrites the automatic one.

See also
- TASK.NAME.DELete
- TASK.NAME.RESet
- TASK.NAME.Set
- TASK.NAME.view
- TASK

**TASK.NAME.DELete**

Delete a task name table entry

Format: **TASK.NAME.DELete** `<task_magic>`

Deletes the entry, specified by `<task_magic>`, from the task name translation table. If the entry is an automatic entry, the next usage of task names may add the automatic entry again.

See also
- TASK.NAME
- TASK.NAME.view

**TASK.NAME.RESet**

Reset task name table

Format: **TASK.NAME.RESet**

Erases the whole task name translation table. If the TASK configuration file supports task name evaluation, the next usage of task names will populate the table again with automatic entries.

See also
- TASK.NAME
- TASK.NAME.view
**TASK.NAME.Set**

Set a task name table entry

Format: `TASK.NAME.Set <task_magic> <task_name>`

Adds a manual entry to the task name translation table.

- `<task_magic>`, `<task_name>`: The string specified by `<task_name>` is assigned to the task specified by `<task_magic>`. If the table contains already an automatic entry for the specified task magic number, it will be overwritten by the new entry!

**Example:**

```
TASK.NAME.Set 0x58D68 "My_Task 1"
```

**See also**

- TASK.NAME
- TASK.NAME.view

---

**TASK.NAME.view**

Show task name translation table

Format: `TASK.NAME.view`

Shows the contents of the task name translation table.

A Flag “a”: The entry was set automatically by the TASK configuration file.

B Flag “m”: The entry was set manually by the `TASK.NAME.Set` command.

**See also**

- TASK.NAME
- TASK.NAME.DELETE
- TASK.NAME.RESET
- TASK.NAME.Set
**TASK.NoBreak**  
**ICE only**  

It was used in 68k ICE only for task selective debugging. **TASK.NoBreak** marks a task of the selective list to be not breaked at the next entry.

See also  
- TASK

---

**TASK.NoDebug**  
**ICE only**  

It was used in 68k ICE only for task selective debugging. **TASK.NoDebug** marks a task of the selective list as inactive for debugging.

See also  
- TASK

---

**TASK.NoStop**  
**ICE only**  

It was used in 68k ICE only for task selective debugging. **TASK.NoStop** marks a task of the selective list to be stopped.

See also  
- TASK
**TASK.OFF**

**Deprecated**

ICE only

Format: **TASK.OFF** (deprecated)

It was used in 68k ICE only for task selective debugging. **TASK.OFF** switches off the task selective debugger.

**See also**

- TASK

**TASK.ON**

**Deprecated**

Format: **TASK.ON** (deprecated)

It was used in 68k ICE only for task selective debugging. **TASK.ON** switches on the task selective debugger.

**See also**

- TASK
### TASK.ORTI.CPU

**Set OSEK SMP CPU number**

| Format: | TASK.ORTI.CPU <cpu_id> |

If TRACE32 is set up in AMP mode (one PowerView instance for each core), it assigns a CPU ID to each individual core, starting with zero. An AUTOSAR/OSEK operating system in SMP mode may assign a different CPU ID to the cores, depending how the OS uses the chip.

This command instructs the debugger to use the given CPU ID when extracting core dependent information from the ORTI file.

**See also**
- TASK.ORTI
- TASK.ORTI.load
- TASK.ORTI.NOSTACK
- TASK.ORTI.SPLITSTACK

### TASK.ORTI.load

**Configure OS Awareness for OSEK/ORTI**

| Format: | TASK.ORTI.load <file> |

Configures the OS Awareness for AUTOSAR/OSEK operating systems using ORTI. For a detailed description, please refer to the chapter "OS Awareness Manual OSEK/ORTI" (rtos_orti.pdf).

**See also**
- TASK.ORTI

---

©1989-2020 Lauterbach GmbH
Excluding an ORTI task from stack evaluation

**Format:**

```
TASK.ORTI.NOSTACK <task_name>
```

When using the OS Awareness for ORTI (see `TASK.ORTI.load`), this command excludes a task from all stack evaluations, e.g. when performing a trace function analysis. Usually used for the idle routine if it isn't running as a separate task.

**See also**

- `TASK.ORTI`
Some AUTOSAR/OSEK OSs use the same magic (NO_TASK) for the idle ORTI tasks on all cores. However, for the function analysis, the idle tasks need to be split to the individual cores because the cores are executing the idle tasks concurrently.

The command `TASK.ORTI.SPLITSTACK` splits the stacks of the idle ORTI tasks to the individual cores.

```
<Task_name> Specify the name of the idle ORTI task.
```

**Example:**

```
TASK.ORTI.SPLITSTACK "idle"
```

**Output:** Function analysis in the `<trace>.STATistic.TREE` window

- **Before**
  - No stack overflow.
- **After**
  - Stack overflow after executing `TASK.ORTI.SPLITSTACK`.

- **B** The core numbers, here 0 and 1, are appended to the name of the idle task `idle:0` and `idle:1`.

- **C** Core coloring scheme, e.g. green for core 1. See also `CORE.SHOWACTIVE`.

**See also**

- `TASK.ORTI`
**TASK.RESet**

**Reset OS Awareness**

Resets the OS Awareness.

The configuration is cleared, all additional commands and features are removed.

**See also**

- TASK

---

**TASK.RUN**

**Load task**

**Format:**

```
 TASK.RUN <process>
```

Loads `<process>` and prepares it for debugging.

Only applicable if GDB (Linux) or TRK (Symbian) is used as debug agent.

**Example:**

```
 TASK.RUN /bin/hello
```

**See also**

- TASK
- 'Commands for Run Mode Debugging' in 'Run Mode Debugging Manual Linux'
- 'Commands for Run Mode Debugging' in 'Run Mode Debugging Manual QNX'
- 'Commands for Run Mode Debugging' in 'Run Mode Debugging Manual Symbian'
**TASK.select**

Display context of specified task

---

**Format:**  

`TASK.select <task_magic> | <task_id> | "<task_name>"`

---

**Stop mode debugging:** In the case of an SMP system the currently selected core is changed to the core running the specified task. As a result the debugger view is changed to this core and all TRACE32 commands without `/CORE <number>` option apply to it.

If the specified task is not running, TRACE32 reads the register set of the specified task from the OS data structures. This is needed to display the context of the specified task in the TRACE32 PowerView GUI.

The TRACE32 state line changes to a reddish look-and-feel (see screenshot below) to indicate that the context of a not-running task is displayed. TRACE32 display commands such as `List.auto`, `Register.view`, `Frame.view` or `Var.Local` apply to this task. Whereas all other commands switch back to the currently running task before they are executed.

If the task is running on different virtual machine, TRACE32 reads the context of the VCPU that is processing the task on this machine.

---

`<task_magic>, etc. | See also “What to know about the Task Parameters”
| (general_ref_t.pdf).`

---

**Run mode debugging:** Selects the specified task for debugging (e.g. GDB (Linux) or TRK (Symbian)).

```
TASK.select 41.
```

---

**See also**

- [TASK](#)
- [CORE.select](#)
- [MACHINE.select](#)
TASK.SETDIR  Set the awareness directory

OS awarenesses: Linux only

| Format: | TASK.SETDIR <path> |

The Linux awareness and menu call scripts from the awareness directory. This directory is set per default to `~/.demo/<arch>/kernel/linux/<linux_version>`. When loading the awareness outside this directory, TRACE32 prints a warning. With this command you can change the awareness directory. Scripts will be called then from the new directory.

See also

- TASK
The **TASK.STacK** command group allows to watch the stack usage in single tasking and multi-tasking systems. In single tasking systems, or in non supported operating systems, the user has to specify the stack area manually. The task magic number can be any number to identify a stack area.

In configured RTOS operation, the magic number must be the respective task magic number.

The debugger tries to get the current stack pointer. If the OS Awareness is configured, and the configuration file supports stack coverage, the current stack pointer is read out of the task control block of the application. When the application is stopped, the stack pointer is read from register and displayed at the current running task. Without any RTOS configuration the stack pointer will be displayed at the stack that fits to the pointer (pointer inside the stack). If no stack fits, or if the running task could not be found, the stack pointer of the register is displayed in an extra line. (See also **TASK.STacK.view**)

There are two methods to evaluate the maximum stack space. If flag memory is available (see **MAP.Flag**), the flag system can be used. Be sure that flag memory is mapped to the stack areas. The emulator searches from stack top to stack bottom for the first write flag. The found address is then used as maximum stack address. If no flag system is available, you can use a pattern search, if the stack is initialized with a known pattern. In this case, the debugger searches from stack top to stack bottom for the first byte, that is not equal to the specified pattern. (See also **TASK.STacK.PATtern**)

For more information on stack coverage in operating systems, refer to the **OS Awareness Manuals**.

---

**See also**

- **TASK.STacK.ADD**
- **TASK.STacK.DIRection**
- **TASK.STacK.Init**
- **TASK.STacK.RESet**
- **TASK.STacK.PATtern**
- **TASK.STacK.REMove**
- **TASK.STacK.view**

▲ ‘Features’ in ‘Hypervisor Awareness Manual Wind River Hypervisor’
▲ ‘Features’ in ‘OS Awareness Manual AMX’
▲ ‘Features’ in ‘OS Awareness Manual ARTK’
▲ ‘Features’ in ‘OS Awareness Manual Atomthreads’
▲ ‘Features’ in ‘OS Awareness Manual DSP/BIOS’
▲ ‘Features’ in ‘OS Awareness Manual ChibiOS/RT’
▲ ‘Features’ in ‘OS Awareness Manual Chorus Classic’
▲ ‘Features’ in ‘OS Awareness Manual CMX’
▲ ‘Features’ in ‘OS Awareness Manual CMX-TINY+’
▲ ‘Features’ in ‘OS Awareness Manual eCos’
▲ ‘Features’ in ‘OS Awareness Manual embOS’
▲ ‘Features’ in ‘OS Awareness Manual FAMOS’
▲ ‘Features’ in ‘OS Awareness Manual FreeRTOS’
▲ ‘Features’ in ‘OS Awareness Manual HI7000’
▲ ‘Features’ in ‘OS Awareness Manual Linux’
▲ ‘Features’ in ‘OS Awareness Manual LiteOS’
▲ ‘Features’ in ‘OS Awareness Manual MQX’
▲ ‘Features’ in ‘OS Awareness Manual MTOS-UX’
▲ ‘Features’ in ‘OS Awareness Manual NetBSD’
▲ ‘Features’ in ‘OS Awareness Manual NORTI’
▲ ‘Features’ in ‘OS Awareness Manual Nucleus PLUS’
▲ ‘Features’ in ‘OS Awareness Manual NuttX’
▲ ‘Features’ in ‘OS Awareness Manual OKL4’
▲ ‘Debug Features’ in ‘OS Awareness Manual OSEK/ORTI’
▲ ‘Features’ in ‘OS Awareness Manual OS21’
▲ ‘Features’ in ‘OS Awareness Manual OSEck’
▲ ‘Features’ in ‘OS Awareness Manual OSE Delta’

©1989-2020 Lauterbach GmbH
TASK.STacK.ADD

Add stack space coverage

Format: TASK.STacK.ADD [<task_magic> [stackrange]] [ Leoption >]

<option>: MACHINE <machine_magic> | <machine_id> | <machine_name>

With the 1st argument: Adds a stack area to the TASK.STacK.view window.

Without the 1st argument: Opens the TASK.STacK.ADD window. Double-click the entry of a stack area you want to add to the TASK.STacK.view window.

When no OS Awareness is loaded:

<task_magic>, <stackrange>

The task magic number is any number used to identify a stack area. In this case the stack range must be specified as a second parameter.

See also “What to know about the Task Parameters” (general_ref_t.pdf).
When an OS Awareness is loaded:

<table>
<thead>
<tr>
<th><code>&lt;task_magic&gt;</code></th>
<th>The magic number <strong>must</strong> be the task magic number. See also “What to know about the Task Parameters” (general_ref_t.pdf).</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;stackrange&gt;</code></td>
<td>If the extension definition file supplies automatic stack range detection (only possible in some OS’s), then the stack range parameter can be omitted. Otherwise specify the stack area manually. If available, you can omit the magic and select a task from a task list.</td>
</tr>
</tbody>
</table>

In hypervisor-based environments:

| MACHINE | Lets you add only stack areas that belong to the selected machine. See also “What to know about the Machine Parameters” (general_ref_t.pdf). |

**Examples**

**Example 1:** When no OS Awareness is loaded

```
TASK.STack.ADD 2 0x1000--0x1fff
```

**Example 2:** When an OS Awareness is loaded

```
TASK.STack.ADD 0x101433C0
```

**Example 3:** In a hypervisor-based environment

```
TASK.STack.ADD 0x101433C0 /MACHINE 3
```

**See also**

- TASK.STack
- TASK.STack.view
TASK.STacK.DIREction  Define stack growth direction

Format:  TASK.STacK.DIREction [UP | DOWN]

Defines whether the stack grows downwards or upwards.

DOWN  The stack starts with the high address and grows to a lower address.

UP  The stack starts with the low address and grows to a higher address.

See also
■ TASK.STacK  ■ TASK.STacK.view

TASK.STacK.Init  Initialize unused stack space

Format:  TASK.STacK.Init [<task_magic>]

If flag system is used on ICE or FIRE:
Resets the write flags on the stack areas. The evaluation for maximum stack space will then start from scratch. If a magic is specified, the write flags of the according stack space will be deleted, otherwise the write flags of all displayed stacks are reset.

If pattern check is used:
Overwrites the currently unused stack space with the pattern defined by TASK.STacK.PATtern. The memory starting from the stack pointer onto the stack boundary address (equals to the low address, if the stack grows downwards) will be initialized with the pattern.

CAUTION:  If the stack is used in an unusual way, e.g. some stack space is used even if the stack pointer does not point behind the used area, relevant target data may be overwritten, and your application may crash.

See also
■ TASK.STacK  ■ TASK.STacK.view

©1989-2020 Lauterbach GmbH
TASK.STacK.PATtern

Define stack check pattern

| Format: TASK.STacK.PATtern [ON | OFF | [%<format>] <pattern>] |

Defines, whether to use flag system or pattern check for stack coverage calculation.

**OFF**
Pattern check is switched off. Stack coverage calculation is done with the flag system (write flags). This option is only available, if a flag system is available.
*Only available on ICE or FIRE systems with flag memory.*  
*On ICD, the stack coverage calculation always uses pattern check.*

**ON**
Pattern check is switched on. Stack coverage calculation is done by comparing the stack data with the predefined pattern (see below). The default pattern is zero.
*Only available on ICE or FIRE systems with flag memory.*  
*On ICD, the stack coverage calculation always uses pattern check.*

**<pattern>**
Pattern check is switched on. Stack coverage calculation is done by comparing the stack data with defined pattern. The pattern must be the value, which represents unused stack space. This will only work, if the stack space is initialized with this value. Use **TASK.STacK.Init** to re-initialize currently unused stack space with the pattern.

*<pattern>* can also be a string enclosed in quotes.

**<format>**
Use a *<format>* to define formats other than bytes e.g. *%Long*.

See also

- TASK.STacK
- TASK.STacK.view
**TASK.STacK.PATternGAP**

Define check pattern gap

Format: \[\text{TASK.STacK.PATternGAP} \ [<\text{value}>]\]

If the stack check pattern defined with `TASK.STacK.PATtern` is not contiguous, this command defines the gap between two consecutive patterns.

\(<\text{value}>\) Number of bytes between two consecutive stack check patterns.

**Example:** If the stack is pre-filled with a 4-byte pattern `0xdeadbeef` on each 64-byte boundary, specify:

\[
\begin{align*}
\text{TASK.STacK.PATtern} & \ %\text{Long} \ 0x\text{DEADBEEF} \\
\text{TASK.STacK.PATternGAP} & \ 0x\text{40-4}
\end{align*}
\]

See also

- `TASK.STacK`
- `TASK.STacK.view`

**TASK.STacK.ReMove**

Remove stack space coverage

Format: \[\text{TASK.STacK.ReMove} \ [<\text{task_magic}>] \ [/<\text{option}>]\]

\(<\text{option}>\):

- `MACHINE <machine_magic> | <machine_id> | <machine_name>`

**With the 1st argument:** Removes a stack area from the `TASK.STacK.view` window.

**Without the 1st argument:** Opens the `TASK.STacK ReMove` window. Double-click the entry of a stack area you want to remove from the `TASK.STacK.view` window.

\(<\text{task_magic}>\) Specify the task magic number of the task whose stack area you want to remove.

See also “What to know about the Task Parameters” (general_ref_t.pdf).

**MACHINE** Lets you removes only stack areas that belong to the selected machine.

See also “What to know about the Machine Parameters” (general_ref_t.pdf).
**Example:**

```
TASK.STacK.ReMove 0x10147420
```

---

**TASK.STacK.RESet**

Reset stack coverage

**Format:**

```
TASK.STacK.RESet [<task_magic>]
```

Resets the stack coverage system and all manually defined stack areas.

If flag system is used on ICE or FIRE:
Resets the write flags on the stack areas. The evaluation for maximum stack space will then start from scratch. If a magic is specified, the write flags of the according stack space will be deleted, otherwise the write flags of all displayed stacks are reset.

If pattern check is used:
Resets the defined pattern to zero.

---

**See also**

- TASK.STacK
- TASK.STacK.view

---

**See also**

- TASK.STacK
- TASK.STacK.view
Opens a window with stack space coverage.

**Format:**

```
TASK.STacK.view [task_magic] [stackrange] [option]
```

**<option>:**

- **HumanReadable**
- **MACHINE**

Opens a window with stack space coverage.

```
<task_magic>  In single-tasking systems, or in non-supported multitasking systems, you have to specify the first stack manually. Use any task magic number as an ID, and specify the stack range to cover.

<stackrange>
   If the RTOS configuration file supports detection of the stack range, you can use the magic of a specific task and omit the stack range. The range will be automatically calculated from the information of the operating system. In the case of a fully supported operating system, you can start the window without any parameter. The debugger then automatically adds all current active tasks with its stacks to the window.

HumanReadable
   Shows the size of the stack and the spare stack memory in human readable form (byte, kilobytes, megabytes).

MACHINE
   Shows only the stacks of the selected machine.

See also “What to know about the Machine Parameters” (general_ref_t.pdf).
```

**Description of Columns in the TASK.STacK.view Window**

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name or ID for the stack space. In configured RTOS environment it specifies the name or ID of the task.</td>
</tr>
<tr>
<td>low and high</td>
<td>The lowest and highest address of the stack range.</td>
</tr>
<tr>
<td>sp (gray)</td>
<td>Gray: The stack pointer, calculated from a task control block (if available).</td>
</tr>
</tbody>
</table>
It was used in 68k ICE only for task selective debugging. **TASK.Stop** marks a task of the selective list to be not stopped.
The TCB (Trace Control Block) is the HW control interface to the MIPS hardware trace block. For details please refer to the MIPS Trace specifications.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the TCB.state window:

![TCB state configuration window]

In the following, TCB specific controlling and associated commands are described.

See also

- TCB.AllBranches
- TCB.CycleAccurate
- TCB.EX
- TCB.IM
- TCB.KE
- TCB.OFF
- TCB.PCTrace
- TCB.PortMode
- TCB.Register
- TCB.SourceSizeBits
- TCB.state
- TCB.SyncPeriod
- TCB.ThreadSizeBits
- TCB.UM
- TCB.CPU
- TCB.DataTrace
- TCB.FCR
- TCB.InstructionCompletionSizeBits
- TCB.LSM
- TCB.ON
- TCB.PortFilter
- TCB.PortWidth
- TCB.RESet
- TCB.STALL
- TCB.SV
- TCB.TC
- TCB.Type
- TCB.Version
### TCB.AllBranches

**Broadcast all branches**

Format:  
TCB.AllBranches [ON | OFF]

<table>
<thead>
<tr>
<th>OFF (default)</th>
<th>The TCB broadcasts only the address information when the processor branches to a location that cannot be directly inferred from the source code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>The TCB broadcasts the address information for all branches or jumps.</td>
</tr>
</tbody>
</table>

See also  
- TCB  
- TCB.state

### TCB.CPU

**Broadcast information for specified CPU only**

Format:  
TCB.CPU ALL | <cpu_x>

<cpu_x>:  
CPU0 | CPU1

The TCB broadcasts only information for the specified CPU.

<table>
<thead>
<tr>
<th>ALL (default)</th>
<th>The TCB broadcasts information for executed instructions of all active CPU’s.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;cpu_x&gt;</td>
<td>The TCB broadcasts only information for executed instructions of &lt;cpu_x&gt;.</td>
</tr>
</tbody>
</table>

See also  
- TCB  
- TCB.state
TCB.CycleAccurate

Cycle accurate tracing can be used to observe the exact number of cycles that a particular code sequence takes to execute. If cycle accurate tracing is used, trace information is generated for each clock cycle. In this case the \(<\text{core\_clock}>\) can be used to calculate the timestamps for the trace information.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ON</strong></td>
<td>The TCB broadcasts the information which instructions were executed, but additionally stall information. No timestamps are generated by TRACE32.</td>
</tr>
<tr>
<td><strong>OFF</strong> (default)</td>
<td>The TCB broadcasts only the information which instructions were executed. Timestamps are generated by TRACE32.</td>
</tr>
</tbody>
</table>

**Example:**

```
TCB.CycleAccurate ON
Trace.CLOCK 500.MHz ; specify the \(<\text{core\_clock}>\) as base for the trace timestamps
Trace.List ; display the trace information
```

**See also**

- TCB
- TCB.state
TCB.DataTrace

Broadcast specified address and data information

Format: TCB.DataTrace <def>

<def>: ON | OFF |
Address | ReadAddress | WriteAddress |
Data | ReadData | WriteData |
Read | Write

The TCB broadcasts only specified address and data information.

<table>
<thead>
<tr>
<th>ON</th>
<th>The TCB broadcasts all address and data information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF (default)</td>
<td>The TCB broadcasts no address and data but only PC information.</td>
</tr>
<tr>
<td>Address</td>
<td>The TCB broadcasts all address information.</td>
</tr>
<tr>
<td>ReadAddress</td>
<td>The TCB broadcasts only address information in case of a read.</td>
</tr>
<tr>
<td>WriteAddress</td>
<td>The TCB broadcasts only address information in case of a write.</td>
</tr>
<tr>
<td>Data</td>
<td>The TCB broadcasts all data information.</td>
</tr>
<tr>
<td>ReadData</td>
<td>The TCB broadcasts only data information in case of a read.</td>
</tr>
<tr>
<td>WriteData</td>
<td>The TCB broadcasts only data information in case of a write.</td>
</tr>
<tr>
<td>Read</td>
<td>The TCB broadcasts address and data information in case of a read.</td>
</tr>
<tr>
<td>Write</td>
<td>The TCB broadcasts address and data information in case of a write.</td>
</tr>
</tbody>
</table>

See also

- TCB
- TCB.state
### TCB.EX

Broadcast exception level information

**Format:**

```
TCB.EX [ON | OFF]
```

If enabled the TCB broadcasts information for instructions executed on exception level.

| ON (default) | The TCB broadcast information for executed instructions in exception operating mode. |
| OFF          | The TCB does not broadcast information for executed instructions in exception operating mode. |

**See also**

- TCB
- TCB.state

### TCB.FCR

Broadcast function call-return information

**Format:**

```
TCB.FCR [ON | OFF]
```

Enables broadcasting of function call-return information. This information is not treated within TRACE32 PowerView but has to be taken into account for trace decoding especially in case of a belated trace analysis.

**See also**

- TCB
- TCB.state

### TCB.IM

Broadcast instruction cache miss information

**Format:**

```
TCB.IM [ON | OFF]
```

Enables broadcasting of instruction cache miss information. This information is not treated within TRACE32 PowerView but has to be taken into account for trace decoding especially in case of a belated trace analysis.

**See also**

- TCB
- TCB.state
TCB.InstructionCompletionSizeBits  Specify size of completion message

| Format: | TCB.InstructionCompletionSizeBits <number> |

This command is only required if a TRACE32 Instruction Set Simulator is used for a belated analysis of SMP trace information.

This command allows to specify how many bits are used in the trace stream dot instruction completion message.

See also
- TCB
- TCB.state

TCB.KE  Broadcast kernel mode information

| Format: | TCB.KE [ON | OFF] |

If enabled the TCB broadcasts information for instructions executed in kernel mode.

<table>
<thead>
<tr>
<th>OFF</th>
<th>The TCB does not broadcast information for executed instructions in kernel operating mode.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON (default)</td>
<td>The TCB broadcast information for executed instructions in kernel operating mode.</td>
</tr>
</tbody>
</table>

See also
- TCB
- TCB.state
**TCB.LSM**

**Broadcast load store data cache information**

Format: `TCB.LSM [ON | OFF]`

Enables broadcasting of load store data cache miss information. This information is not treated within TRACE32 PowerView but has to be taken into account for trace decoding especially in case of a belated trace analysis.

See also
- TCB
- TCB.state

---

**TCB.OFF**

**Switch TCB off**

Format: `TCB.OFF`

Disables TCB functionality.

See also
- TCB
- TCB.state

---

**TCB.ON**

**Switch TCB on**

Format: `TCB.ON`

Enables TCB functionality.

See also
- TCB
- TCB.state
**TCB.PCTrace**

**Broadcast program counter trace**

| Format: TCB.PCTrace [ON | OFF] |

If enabled, the TCB broadcasts program counter trace information.

| OFF | The TCB does not broadcast program counter trace information. |
| ON (default) | The TCB broadcast program counter trace information. |

**See also**

- TCB
- TCB.state

---

**TCB.PortFilter**

**Disable port filter**

| Format: TCB.PortFilter [ON | OFF] |

Disable filtering of trace data within AutoFocus II Preprocessor. This command should only be used for diagnostic purpose!

**See also**

- TCB
- TCB.state
TCB.PortMode

Specify trace clock ratio

Format:  

TCB.PortMode <trace_clock>/<cpu_clock>

<trace_clock> 8/1 | 4/1 | 2/1 | 1/1 | 1/2 | 1/4 | 1/6 | 1/8

<cpu_clock>:

Specifies the ratio between trace- and CPU clock in case of off-chip trace.

Example:

TCB.PortMode 1/2 ; <trace_clock> is one half of <core_clock>.

See also  
■ TCB  ■ TCB.state

TCB.PortWidth

Specify trace port width

Format:  

TCB.PortWidth <width>

<width>: 4 | 8 | 16 | 64

Specify the trace port width in number of bits. This value is determined automatically by selecting trace method or reading trace configuration register from target. Therefore this command should only be used for diagnosis purpose or if necessary belated trace analysis.

See also  
■ TCB  ■ TCB.state
TCB.Register [<file>] [<option>]

<option>: SpotLight | DualPort | Track | CORE <core_number> | Deport

Default: OFF.

For a description of the options, see PER.view.

Deport

Updates the control registers while the program is running (only possible if SYStem.MemAccess CPU is selected).

Example:

```
TCB.Register permipstcb.per  ; display the TCB control registers
                        ; use the format description in permipstcb.per
TCB.Register, /SpotLight  ; display the TCB control registers
                        ; mark changes on the registers
```

See also
- TCB
- TCB.state
TCB.RESet

Reset TCB setup to default

Format: **TCB.RESet**

Resets the TCB settings to default.

See also
- TCB
- TCB.state

TCB.SourceSizeBits

Specify number of bit for core information in trace

Format: **TCB.SourceSizeBits** `<number>`

This command is only required if a TRACE32 Instruction Set Simulator is used for a belated analysis of SMP trace information.

This command allows to specify how many bits are used in the trace stream to identify the source core.

See also
- TCB
- TCB.state

TCB.STALL

Stall CPU for complete trace

Format: **TCB.STALL** [ON | OFF]

If enabled, TCB broadcasts slow but complete trace information.

<table>
<thead>
<tr>
<th>OFF</th>
<th>The TCB broadcasts trace information in real-time with the risk of broken trace flow.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>The TCB stall CPU if necessary and broadcast always complete information.</td>
</tr>
<tr>
<td>(default)</td>
<td></td>
</tr>
</tbody>
</table>

See also
- TCB
- TCB.state
TCB.state

Displays the TCB configuration window.

Format: TCB.state

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

See also
- TCB
- TCB.CPU
- TCB.DataTrace
- TCB.FCR
- TCB.InstructionCompletionSizeBits
- TCB.LSM
- TCB.ON
- TCB.PortFilter
- TCB.PortWidth
- TCB.RESet
- TCB.STALL
- TCB.SyncPeriod
- TCB.ThreadSizeBits
- TCB.UM
- TCB.AllBranches
- TCB.CycleAccurate
- TCB.EX
- TCB.IM
- TCB.KE
- TCB.OFF
- TCB.PCTrace
- TCB.PortMode
- TCB.Register
- TCB.SourceSizeBits
- TCB.SV
- TCB.TC
- TCB.Type
- TCB.Version
TCB.SV

Broadcast supervisor mode information

TCB.SV [ON | OFF]

If enabled the TCB broadcasts information for instructions executed in supervisor mode.

| ON (default) | The TCB broadcast information for executed instructions in supervisor operating mode. |
| OFF          | The TCB does not broadcast information for executed instructions in supervisor operating mode. |

See also
- TCB
- TCB.state

TCB.SyncPeriod

Specify TCB sync period

Format: TCB.SyncPeriod <period>

<period>: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Specify the period in cycles the TCB broadcasts a synchronization message.

<period> The TCB sync period in $2^{(<period> + 5)}$ cycles.

See also
- TCB
- TCB.state
TCB.TC

Broadcast information for specified HW thread

<table>
<thead>
<tr>
<th>Format:</th>
<th>TCB.TC ALL</th>
<th>&lt;tc_x&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;tc_x&gt;:</td>
<td>TC0</td>
<td>TC1</td>
</tr>
</tbody>
</table>

The TCB broadcasts only information for the specified HW thread.

| ALL (default) | The TCB broadcasts information for executed instructions of all active TCs. |
| <tc_x> | The TCB broadcasts only information for executed instructions of <tc_x>. |

See also

- TCB
- TCB.state

TCB.ThreadSizeBits

Specify number of bit for thread information in trace

| Format: | TCB.ThreadSizeBits <number> |

This command is only required if a TRACE32 Instruction Set Simulator is used for a belated analysis of SMP trace information.

This command allows to specify how many bits are used in the trace stream to identify the source thread context.

See also

- TCB
- TCB.state
TCB.Type

Specify TCB type

Format: TCB.Type | <tcb_type>

<tcb_type>: PD | PD74K | IFLOW | FALCON | ZEPHYR

This command is only required if a TRACE32 Instruction Set Simulator is used for a belated analysis of SMP trace information.

<table>
<thead>
<tr>
<th>TCB.Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD</td>
<td>MIPS standard program and data trace control block.</td>
</tr>
<tr>
<td>PD74K</td>
<td>Specific MIPS74K program data trace control block.</td>
</tr>
<tr>
<td>IFLOW</td>
<td>MIPS standard instruction flow trace control block.</td>
</tr>
<tr>
<td>FALCON</td>
<td>Lantiq specific instruction flow trace control block.</td>
</tr>
<tr>
<td>ZEPHYR</td>
<td>Broadcom specific program and data trace control block.</td>
</tr>
</tbody>
</table>

TCB.UM

Broadcast user mode information

Format: TCB.UM [ON | OFF]

If enabled the TCB broadcasts information for instructions executed in user mode.

<table>
<thead>
<tr>
<th>TCB.UM</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>The TCB broadcast information for executed instructions in user operating mode.</td>
</tr>
<tr>
<td>OFF</td>
<td>The TCB does not broadcast information for executed instructions in user operating mode.</td>
</tr>
</tbody>
</table>
TCB.Version

Specify trace cell version

Format: 

`TCB.Version <number>`

This command is only required if a TRACE32 Instruction Set Simulator is used for a belated trace analysis. This command allows to specify manually the version number of the TCB trace cell. The version number must fit to the TCB the trace data have been recorded with. It could be found in the header of the TCB window if TRACE32 is connected to the referring target.

See also

- TCB
- TCB.state
Overview TERM

Multitasking operating systems or monitor programs running on the target system often need a terminal interface for operation. This interface can be implemented either using peripherals (e.g. serial port) or as a memory based interface. The memory based interface can work in several operation modes. It can communicate either on character basis or with blocks of up to 255 bytes length. The memory access can either be made while the target is running (when the system supports such run-time memory accesses) or only when the target is stopped.

When the EPROM simulator (ESI) is used, the ESI can be used as communication port as well. Some processor architectures also provide a special communication interface which is accessible through the BDM/JTAG port (DCC modes).

The standard terminal window provides only the basic functions Backspace, Return and LineFeed. A VT100 emulation mode is also available. A character can only be entered when the cursor is positioned in an active window. The terminal window may also be used for “virtual hosting”. This allows to access some basic operation system functions and the file system of the host from the target program. This functionality is only available in the TERM.GATE command.
Interface Routines

In this section:

- EPROM Simulator
- Single Character Modes
- Buffered Modes
- Serial Line Debugger
- Special Hardware, JTAG

Interface Routines (EPROM Simulator)

This is an example in C to access the terminal window. The address of the ports depends on the width and location of the EPROMs. The example assumes 8-bit wide EPROMs. For 16-bit EPROMs the addresses must be doubled and the types changed from char to short.

```c
extern volatile unsigned char input_port at 0x1000;
extern volatile unsigned char status_port at 0x1400;
extern volatile unsigned char output_port[256] at 0x1800;

void char_out(c)
unsigned char c;
{
    unsigned char dummy;
    if ( c == 0 )    /* refuse to send 0 (break) character */
        return;
    while ( status_port&2);   /* wait until port is free */
    dummy = outport_port[c];   /* send character */
}

int char_in();
{
    unsigned char c;    /* wait until character is ready */
    while (! (status_port&1));  /* read character */
    c = input_port;    /* manual break executed ? */
    if ( c == 0 )
        break_emulation();
    return c;
}
```
Interface Routines (Single Character Modes)

This interface occupies two memory cells in which characters can be transferred. A zero means that no character is available and the interface is ready. When the target is not able to provide a dual-ported memory access it is possible to stop the target after it has placed a character in the communication area and the terminal command will restart the target automatically after it has processed the character.

This is an example in C to access the terminal window. By changing the char_in and char_out routines within the library, all more complex functions like printf() or scanf() are redirected to the terminal window.

```
extern volatile char input_port,
output_port

    void char_out(c)
    char c;
    {
        while (output_port != 0 ) ; /* wait until port is free */
        output_port = c; /* send character */
    }

int char_in();
    {
        char c;
        while ( input_port == 0 ) ; /* wait until character is ready */
        c = input_port; /* read character */
        input_port = 0; /* clear input port */
        return c;
    }
```

NOTE: Some emulation heads have special dual-port access modes, that require special cycles to be executed (e.g. IDLE mode on H8 probes).

Interface Routines (Buffered Modes)

An example for using the buffered mode can be found in ~/.demo/etc/terminal/t32term/t32term_memory.c. This example contains also examples for using the virtual hosting feature of the TERM.GATE command.

Interface Routines (Serial Line Debugger)

The serial line can be used as usual. Only the data values 0 have a special meaning. Receiving such a value means an emulation break. Sending such a value is not allowed for the user program.

Interface Routines (Special Hardware, JTAG)

Check the target appendix for your processor for details and availability.
Functions

TERM.LINE(<address>, <line>)

Returns the line of a displayed terminal window as a string. Negative line numbers retrieve count from the bottom of the window (line -1 is the last line).

Fast Data Write

The fast data write system allows to transfer data from the target to a file on the host. The data transfer rate can be up to 250 KBytes/s. The max. reaction time is 50 µs when the transfer is not interruptible or 150 µs when the transfer is interruptible. Data can be transferred either 8, 16 or 32 bit wide. The principle is similar to the terminal emulation. The interface occupies two memory cells, one byte to control the transfer and a second byte or word to hold the data. A zero in the control cell means that the emulator is ready to accept data. Writing a ‘01’ by the CPU causes the data to be transferred to the host. Writing ‘02’ saves the current data buffer to the host. The time required by this disk save depends on the host and communication speed. The data buffer is saved automatically after the buffer is full. The value ‘03’ can be used as a NOP command to wait for the start of the transfer. Writing ‘ff’ terminates the data transfer. The Fast Data Write system has been replaced by the FDX system for ICD and FIRE systems.

Interface Routines

This is an example in C to access the fast data transfer.

```c
extern volatile char control_port;
extern volatile short data_port;

void word_out()
short c;
{
    while (control_port != 0) ; /* wait until port is free */
data_port = c ; /* place 16 bit in buffer */
control_port = 1; /* send data to buffer/host */
}

int begin_transfer(c);
short c;
{
    while (control_port != 0) ; /* wait until transfer is ready */
}

int end_transfer(c);
short c;
{
    while (control_port != 0) ; /* wait until port is free */
control_port = 0xff; /* stop transfer program */
}
```
TERM.CLEAR

Clear terminal window

Format:
TERM.CLEAR [<channel>]
TERM.CLEAR [<address>] (deprecated)

<channel>: #<number>

Clears the terminal window and places the cursor to the home position.

See also
■ TERM
■ TERM.view

TERM.CLOSE

Close files

Format:
TERM.CLOSE [<channel>]
TERM.CLOSE [<address>] (deprecated)

<channel>: #<number>

Closes the output file created with TERM.WRITE.

See also
■ TERM
■ TERM.view

TERM.CMDLINE

Specify a command line

Format:
TERM.CMDLINE <cmdline>

The command can specify a command line for the SYS_GET_CMDLINE (0x15) system call if ARM compatible semihosting is used.

See also
■ TERM
■ TERM.view
Write data from the user program to a disk file. The interface occupies two memory cells, one byte to control the transfer and a second byte or word to hold the data. The address parameter defines the location of the data byte, the control byte is at 'address+4'. A zero in the control cell means that the emulator is ready to accept data. Writing a '01' by the CPU causes the data to be transferred to the host. Writing '02' saves the current data buffer to the host. The time required by this disk save depends on the host and communication speed. The data buffer is saved automatically after the buffer is full. The value '03' can be used as a NOP command to wait for the start of the transfer. Writing 'ff' terminates the data transfer. This command has been superseded by the FDX commands (which is available on all systems).

<table>
<thead>
<tr>
<th>Byte, Word, Long</th>
<th>Defines the data width of the transfer location.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFER</td>
<td>Defines the size of the buffer pool. The default size is 4096 bytes. As long as data fits into this pool, the response time to the host will be very short and independent of host and communication speed. If the buffer is full, it is transferred to the host. The time required for this transfer depends on the communication speed and host system.</td>
</tr>
<tr>
<td>NoBreak</td>
<td>Usually the command can be interrupted any time. This communication increases the maximum response time of the emulator. Interruptible transfers have a response time of 150 µs, non interruptible have 50 µs.</td>
</tr>
<tr>
<td>Single</td>
<td>Single shot mode. In this mode the transfer is terminated after the first buffer has been filled.</td>
</tr>
</tbody>
</table>

Example 1: Write 300000 bytes in one shot to the file 'data_adc'. The data is transferred in 32 bit words. The transfer function will stop after receiving all words from the user program.

```plaintext
E::d.s e:0x100 0x3
E::go
E::term.fastwrite e:0x100 data_adc /long /single /buffer 300000.
E::term.close
```
Example 2: Same as above, but data transfer now 16 bit and reducing the max. response time by making the function not interruptible.

```
E::d.s e:0x100 0x3
E::go
E::term.fastwrite e:0x100 data_adc /word /single /buffer 300000. /nobreak
E::term.close
```

See also
- TERM
- TERM.view

TERM.GATE
Terminal with virtual hosting

Format: `TERM.GATE [<channel>]`
TERM.GATE [<addresses>] (deprecated)

- `<channel>`: `#<number>`
- `<addresses>`: `[<address_out>] [<address_in>]`

Creates a terminal emulation window which allows virtual hosting.

See also
- TERM
- TERM.view
- 'Release Information' in 'Release History'

TERM.HARDCOPY
Print terminal window contents

Format: `TERM.HARDCOPY [<channel>]`

- `<channel>`: `#<number>`

Opens the Print dialog of the operating system. From the Print dialog, you can select a printer to make a hardcopy of the terminal window contents or print the terminal window contents to file.

See also
- TERM
- TERM.view
TERM.HEAPINFO

Define memory heap parameters

Format: TERM.HEAPINFO [<heap_base>] [<heap_limit>] [<stack_base>] [<stack_limit>]

Defines the memory heap and stack locations returned by the ARM compatible semihosting calls. Only relevant when ARM compatible semihosting is used.

Please note that the heap grows toward higher memory addresses (heap_base < heap_limit) and the stack grows towards lower memory addresses (stack_base > stack_limit). <heap_base> = 0 advises the application to locate the heap at the top of the memory region.

See also
- TERM
- TERM.view

TERM.LocalEcho

Enables/disables local echo for new terminal windows

Format: TERM.LocalEcho [<channel>] [ON | OFF]

<channel>: #<number>

Defines, if terminal windows, which are opened after the TERM.LocalEcho command with the TERM.view or TERM.GATE command, will have a local echo or not. Terminal windows with enabled local echo also show the transmitted characters in addition to the received characters.

See also
- TERM
- TERM.view
**TERM.METHOD**

Select terminal protocol

**Format:**

```
TERM.METHOD [ <channel> ] <method>
```

- `<channel>`: 
  
  ```
  #<number>
  ```

- `<method>`:
  
  ```
  SingleE [ <output> ] [ <input> ] [/ <option> ]
  BufferE [ <output> ] [ <input> ] [/ <option> ]
  SingleC [ <pc> ] [ <output> ] [ <input> ] [/ <option> ]
  BufferC [ <pc> ] [ <output> ] [ <input> ] [/ <option> ]
  SingleS [ <output> ] [ <input> ] [/ <option> ]
  BufferS [ <output> ] [ <input> ] [/ <option> ]
  COM [ <name> ] [ <baudrate> ] [ <bits> ] [ <parity> ] [ <stopbits> ] [ <handshake> ]
  TCP [ <host> ] [ <port> ]
  PIPE
  
  DCC [ / <option> ]
  DCC3 [ / <option> ]
  DCC4A [ / <option> ]
  DCC4B [ / <option> ]
  
  SIM
  VIRTIO
  
  CCIO
  BRK1_14 [ <address> ] [ / <option> ]
  ARMSWI [ <address> ] [ / <option> ]
  CHORUS
  
  ESI
  SERIAL
  
- `<input>`:
  
  ```
  <address>
  ```

- `<output>`:
  
  ```
  
  ```

- `<name>`:
  
  Windows:
  
  ```
  COM1 | COM2 | ... | COM9
  alternatively (if COMx fails) and for ports >9:
  \ \COM1 | \ \COM2 | ... | \ \COM10 | \ \COM11 | ...
  ```

  Linux: path to device, e.g.
  
  ```
  /dev/ttyS0 | /dev/ttyS1 | /dev/ttyUSB0 | ...
  ```

- `<bits>`:
  
  ```
  5 | 6 | 7 | 8
  ```
Defines how data is exchanged between the target application and the debugger. On some targets additional processor specific modes may be available.

<table>
<thead>
<tr>
<th>&lt;methods&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleE</td>
<td>Single characters using real time access (e.g. Dualport)</td>
</tr>
<tr>
<td>BufferE</td>
<td>Buffered transfer using real time access</td>
</tr>
<tr>
<td>SingleS</td>
<td>Single characters using regular access at spot points.</td>
</tr>
<tr>
<td>BufferS</td>
<td>Buffered transfer using regular access at spot points.</td>
</tr>
<tr>
<td>BRK1_14</td>
<td>This is a CPU specific option for XTENSA. For more information, see “CPU specific TERM.METHOD Command” (debugger_xtensa.pdf). Only available for Xtensa</td>
</tr>
<tr>
<td>SingleC</td>
<td>Single characters, accessed when CPU is stopped. The additional parameter the PC location of the breakpoint that stops the CPU for communication.</td>
</tr>
<tr>
<td>BufferC</td>
<td>Buffered transfer, accessed when CPU is stopped.</td>
</tr>
<tr>
<td>ESI</td>
<td>Use the ESI for communication. This protocol can also be used when a BDM/JTAG debugger is used together with an ESI (EPROM simulator).</td>
</tr>
<tr>
<td>SERIAL</td>
<td>Use the serial (or ethernet) interface of the debug monitor to exchange data.</td>
</tr>
<tr>
<td>DCC</td>
<td>Use the DCC port of the JTAG interface (only on some architectures)</td>
</tr>
<tr>
<td>DCC3</td>
<td>Same as DCC, but transfer up to 3 characters at once.</td>
</tr>
<tr>
<td>DCC4A</td>
<td>Same as DCC, but transfer up to 4 ascii characters at once.</td>
</tr>
<tr>
<td>DCC4B</td>
<td>Same as DCC, but transfer always 4 characters at once.</td>
</tr>
<tr>
<td>ARMSWI</td>
<td>ARM compatible SWI bases semihosting via SWI breakpoint.</td>
</tr>
<tr>
<td>SIM</td>
<td>Terminal via simulator API.</td>
</tr>
<tr>
<td>VIRTIO</td>
<td>Semihosting of VIRTIO simulation.</td>
</tr>
<tr>
<td>COM</td>
<td>Serial interface of the host.</td>
</tr>
</tbody>
</table>
### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;output&gt;</code></td>
<td>Addresses of the output (target-&gt;debugger) and input (debugger-&gt;target) buffers for memory based terminals.</td>
</tr>
<tr>
<td><code>&lt;input&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;host&gt;</code></td>
<td>Host name or IP address of TCP terminal (TELNET)</td>
</tr>
<tr>
<td><code>&lt;port&gt;</code></td>
<td>TCP terminal port number (default: 23)</td>
</tr>
</tbody>
</table>

### Examples:

```plaintext
TERM.METHOD BufferE Var.ADDRESS("messagebufferout") \ Var.ADDRESS("messagebufferin")
```

```plaintext
TERM.METHOD #1 BufferE Var.ADDRESS("messagebufferout") \ Var.ADDRESS("messagebufferin")
```

### See also

- TERM
- TERM.view

▲ 'Release Information' in 'Release History'
**TERM.Mode**

Define terminal type

| Format: | TERM.Mode [<channel>] [ASCII | STRING | RAW | HEX | VT100] |
|--------|------------------------------------------------|
| <channel>: | #<number> |
| <option>: | CORE <corenumber> |

Defines the terminal type used for new terminal windows.

| ASCII | Terminal behaves like a typewriter. CR and LF are evaluated. |
| STRING | Terminal interprets data as single line strings. Needed for some Printf libraries. CR is ignored. LF is evaluated. |
| RAW | Terminal shows the incoming data like an HEX/ASCII dump. E.g. Spaces, Tabs, CRs, LFs are displayed as special characters only. CR is ignored. LF is evaluated. |
| HEX | Terminal shows the incoming bytes as HEX values. CR and LF are ignored. |
| VT100 | Terminal interprets the VT100 protocol. Color Codes are evaluated e.g. Linux bash like console. CR and LF are evaluated. |

See also

- TERM
- TERM.view
- 'Release Information' in 'Release History'
TERM.Out

Send data to virtual terminal

Sends characters to a terminal. Can be used to control the terminal through a PRACTICE script (*.cmm) or to input non-printable characters from the command line.

Example:

```plaintext
; configure u-boot through serial terminal
TERM.METHOD #1 COM COM1 115200. 8 NONE 1STOP NONE
TERM.view   #1
TERM.Out    #1 10. ; send a single line feed
TERM.Out #1 "setenv bootcmd bootm 0xfe000000 0xfe800000 0xffe00000" 10.
TERM.Out #1 "setenv bootargs root=/dev/ram console=ttyS0,115200" 10.
TERM.Out #1 "saveenv" 10.
```

See also

- TERM
- TERM.view

TERM.PIPE

Connect terminal to named pipe

Connects the terminal to a bidirectional named pipe.

See also

- TERM
- TERM.view
TERM.PipeREAD

**Connect terminal input to named pipe**

**Format:**

```
TERM.PipeREAD [<channel>] <file>
TERM.PipeREAD [<address_in>] <file> (deprecated)
```

```
<channel>: #<number>
```

Connects the terminal to a pipe which sends data to the host.

**See also**

- TERM
- TERM.view

---

TERM.PipeWRITE

**Connect terminal output to named pipe**

**Format:**

```
TERM.PipeWRITE [<channel>] <file>
TERM.PipeWRITE [<output>] <file> (deprecated)
```

```
<channel>: #<number>
```

Connects the terminal to a pipe which receives data from the host.

**See also**

- TERM
- TERM.view

---

TERM.Protocol

**Select terminal protocol**

**Format:**

```
TERM.Protocol (deprecated)
Use TERM.METHOD instead.
```

**See also**

- TERM
- TERM.view
TERM.PULSE

Enable pulse generator for transfers

| Format: | TERM.PULSE [<channel>] [ON | OFF] |
|------------------|----------------------------------|
| <channel>:       | #<number>                        |

Issues a pulse on the exception pulse generator (ICE/FIRE) or PODBUS trigger after each transfer. This pulse may be used to trigger an interrupt on the target system to trigger interrupt based communication.

See also
- TERM
- TERM.view

TERM.Rate

Define polling rate

FIRE / ICE only

<table>
<thead>
<tr>
<th>Format:</th>
<th>TERM.Rate [&lt;channel&gt;] [&lt;samples_per_second&gt;] [&lt;buffer&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;channel&gt;:</td>
<td>#&lt;number&gt;</td>
</tr>
</tbody>
</table>

Defines the poll interval for the terminal (in dual-ported communication protocols on some hardware configurations). This allows to guarantee a minimum reaction time to outputs to the terminal.

Example:

E::TERM.Rate 1000. ; poll at least every millisecond

See also
- TERM
- TERM.view
### TERM.READ

**Get terminal input from file**

The contents of the file are send to the terminal, defined by the optional address. The terminal must already exist to use this command. The **TERM.CLOSE** command closes the input file after or during transfer.

**Example:**

```
TERM.READ #1 key_input.in
```

**See also**

- TERM
- TERM.view

### TERM.RESet

**Reset terminal parameters**

Closes the I/O redirection files and set all parameters to default values.

**See also**

- TERM
- TERM.view
TERM.SCROLL  
Enable automatic scrolling for terminal window

Format: **TERM.SCROLL** [\(<channel>\)] [\(\text{ON} \mid \text{OFF}\)]

\(<channel>\): \#<number>

Default: OFF.

Enables or disables automatic scrolling. With automatic scrolling enabled the visible window will follow the terminal cursor.

See also
- TERM
- TERM.view

TERM.SIZE  
Define size of terminal window

Format: **TERM.SIZE** [\(<channel>\)] [\(<columns>\)] [\(<lines>\)] [\(<backlog\_size>\)]

\(<channel>\): \#<number>

Defines the size of the virtual terminal in lines and columns.

\(<backlog\_size>\) 
This value defines the lines of the backlog buffer. The backlog is updated whenever a line scrolls out of the “real” part of the \(\text{TERM.view}\) window.

See also
- TERM
- TERM.view

▲ ‘Release Information’ in ‘Release History’
**TERM.TCP**

Route terminal input/output to TCP port.

Format: `TERM.TCP [<channel>] <port>`

<channel>: #<number>

Routes terminal input/output to TCP port.

See also
- TERM
- TERM.view

**TERM.TELNET**

Open TELNET terminal window

Format: `TERM.TELNET [<channel>]`

<channel>: #<number>

Opens the terminal emulation window for TELNET.

Example:

```
TERM.METHOD TCP 10.2.23.140 ;using default port 23
TERM.MODE VT100
TERM.TELNET
```

See also
- TERM
- TERM.view

▲ ‘Release Information’ in ‘Release History’
TERM.TRIGGER
Trigger on string in terminal window

Format:

TERM.TRIGGER [<channel>] <message_string>
TERM.TRIGGER [<address_out>] <string> (deprecated)

[channel]: #<number>

Sets a trigger for the occurrence of a specific string in the terminal window. The function TERM.TRIGGERED() returns if the trigger has occurred or not.

<table>
<thead>
<tr>
<th>&lt;channel&gt;</th>
<th>Handle to refer to a terminal. A new handle can be created with TERM.METHOD.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;address_out&gt;</td>
<td>Only required for memory-based data exchange (SingleE, BufferE, SingleS, BufferS).</td>
</tr>
<tr>
<td>&lt;message_string&gt;</td>
<td>Case sensitive. The message string or substring you want the TERM.TRIGGER() function to find in the TERM.view or TERM.GATE window.</td>
</tr>
</tbody>
</table>
**Example**: A typical use case might be to automatize the boot process. The following script stops the boot process after the string “Hit any key to stop autoboot” appears in the terminal window.

```plaintext
; <example_terminal_output>
; U-Boot <year>.<month>
; CPU: example CPU
; Board: example Board
; Boot: SD-Card
; DRAM: 2 GiB
; MMC: SDHC: 0
; In: serial
; Out: serial
; Err: serial
; Normal Boot
; Hit any key to stop autoboot: 3
; </example_terminal_output>

; create terminal configuration and assign it to the handle #1
TERM.METHOD #1 COM COM3 115200. 8 NONE 1STOP NONE

; create the terminal and open the TERM.view window
TERM.view #1

; STATE.RUN() -> STOPPED
Break

; wait for trigger with timeout, press ENTER
TERM.TRIGGER #1 "Hit any key"
; start CPU
Go
SCREEN.WAIT TERM.TRIGGERED(#1) 10.s
IF !TIMEOUT()
  (   TERM.OUT #1 0xA
  WAIT 0.1s
  TERM.OUT #1 "setenv bootargs ...."
  )
ELSE
  (
  ; error handler
  )

See also
■ TERM  ■ TERM.view  ■ TERM.TRIGGERED()  ■ TIMEOUT()
```

©1989-2020 Lauterbach GmbH
Term.Vector

ICE only

**Define interrupt vectors**

Defines interrupt vectors for the I/O port simulation. The exception generator will generate one pulse with the `<tx_vector>`, after the terminal has accepted one character and one pulse with `<rx_vector>`, if the terminal has placed one character in the dual-port memory. If one vector is zero, no exception will be generated (default).

**See also**

- TERM
- TERM.view

---

Format: `TERM.Vector [channel] [tx_vector] [rx_vector]`

`<channel>`:

```
#<number>
```

```
Defines interrupt vectors for the I/O port simulation. The exception generator will generate one pulse with the
<tx_vector>, after the terminal has accepted one character and one pulse with <rx_vector>, if the terminal
has placed one character in the dual-port memory. If one vector is zero, no exception will be generated
(default).

See also

- TERM
- TERM.view
```
TERM.view

Opens the terminal emulation window. The protocol of the terminal is defined through TERM.METHOD. For protocols based on memory based data exchange (SingleE, BufferE, SingleS, BufferS), the communication buffer addresses can either be specified with TERM.METHOD or directly with TERM.view.

**Example:**

```plaintext
; see terminal source code in
; ~/demo/etc/terminal/t32term/t32term_memory.c
TERM.METHOD #1 BufferE E:0x00000100 E:0x00000200
TERM.MODE #1 VT100
TERM.view #1

; Hint: the pre-commands WinExt and WinResist create a window that is
; (a) "external" to the TRACE32 PowerView main window and that is
; (b) "resistant" to the WinCLEAR command.
WinExt.WinResist.TERM.view #1
```

**See also**

- TERM
- TERM.CLEAR
- TERM.FastWRITE
- TERM.GATE
- TERM.LocalEcho
- TERM.METHOD
- TERM.Mode
- TERM.Pipe
- TERM.PipeREAD
- TERM.PipeWRITE
- TERM.PULSE
- TERM.RESet
- TERM.READ
- TERM.SCROLL
- TERM.SCROLL
- TERM.TRIGGER
- TERM.WRITE
- TERM.LINE()
- TERM.CMDLINE
- TERM.HEAPINFO
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
- TERM.HARDCOPY
TERM.WRITE
Write terminal output to file

The output sent from the target to the terminal emulation window is written to the specified file. The terminal emulation window must be opened before using this command. The TERM.CLOSE command closes the output file after or during transfer.

Example:

TERM.WRITE #1 term_out.lst

See also
- TERM
- TERM.view
- ‘Release Information’ in ‘Release History’
Overview TPIU

The **TPIU** command group enables you to configure and control the Trace Port Interface Unit (TPIU) of an ARM processor system or a non-ARM processor system using the ARM CoreSight trace. The TPIU is a trace sink which sends the trace data off-chip for capturing by a trace tool.

The TPIU typically outputs trace data via a parallel trace interface consisting of up to 32 trace data signals, a trace clock and optionally a trace control signal (indicating idle).

Some chip designs use these signals internally as an input to a High Speed Serial Trace Port (HSSTP) which converts the parallel data into a serial Xilinx-Aurora-based protocol for sending the serial bit stream off-chip on differential lanes.

A variant of the TPIU is the Serial Wire Output (SWO) which outputs trace data of the Serial Wire Viewer (SWV) via a single signal line. This output has a much lower bandwidth, is typically used for system trace, and is typically found on Cortex-M based designs. This variant does normally not use a dedicated trace connector. Instead it re-uses the TDO pin of a debug connector.

For TPIU setup, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **TPIU.state** window.
**TPIU.CLEAR**

Re-write the TPIU registers

**Format:**

```
TPIU.CLEAR
```

Re-writes the TPIU registers on the target with the settings displayed on the **TPIU.state** window.

**See also**

- TPIU
- TPIU.state

---

**TPIU.IGNOREZEROS**

Workaround for a special chip

**Format:**

```
TPIU.IGNOREZEROS [ON | OFF]
```

**See also**

- TPIU
- TPIU.state

---

**TPIU.NOFLUSH**

Workaround for a chip bug affecting TPIU flush

**Format:**

```
TPIU.NOFLUSH [ON | OFF]
```

Default: OFF.

Activates a workaround for a chip bug which caused serious issues when the trace tool caused a TPIU flush at the end of the trace recording.

**See also**

- TPIU
- TPIU.state
Inform debugger about HSSTP trace frequency

**Format:**  
```
TPIU.PortClock <frequency>
```

Default: 1500Mbps

Informs the debugger about the HSSTP trace frequency to improve the accuracy of the timestamp calculation.

**Example:**
```
TPIU.PortClock 3125Mbps
TPIU.PortClock 3125M ; M is the short form of Mbps
```
TPIU.PortMode

Select the operation mode of the TPIU.

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TPIU.PortMode</strong> <em>&lt;mode&gt;</em></td>
</tr>
</tbody>
</table>

**<mode>:** Bypass | Wrapped | Continuous | NRZ

Selects the operation mode of the TPIU.

---

### Modes for Parallel Trace and HSSTP

The TPIU can optionally output a trace control signal (TRACECTL) which indicates idle cycles of the trace port not worth to record. The TPIU formatter can be used to add the idle information to the trace packets. The formatter needs to be used in case of multiple trace sources to add the ID of the trace source.

| Bypass | TRACECTL pin is available, formatter is not used. |
|Wrapped | TRACECTL pin is available, formatter is used. |
|Continuous | TRACECTL pin is not available, formatter is used. |

---

### Modes for Serial Wire Output

TRACE32 supports the UART/NRZ (NRZ = Non-Return-to-Zero) coding of the Serial Wire Output but not yet the Manchester coding. The bitrate of this asynchronous interface is derived by dividing the CPU frequency.

<table>
<thead>
<tr>
<th>NRZ</th>
<th>NRZ coding at CPU clock divided by <em>&lt;divisor&gt;</em> set up by: <strong>TPIU.SWVPrescaler</strong> <em>&lt;divisor&gt;</em> (default: 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRZ/2 (deprecated) See example below.</td>
<td>NRZ coding at half of the CPU clock speed.</td>
</tr>
<tr>
<td>NRZ/3 (deprecated) See example below.</td>
<td>NRZ coding at a third of the CPU clock speed.</td>
</tr>
<tr>
<td>NRZ/4 (deprecated) See example below.</td>
<td>NRZ coding at a quarter of the CPU clock speed.</td>
</tr>
</tbody>
</table>

---

**Example:**

```plaintext
;(deprecated)
TPIU.PortMode NRZ/4

;please use these two commands instead of NRZ/<divisor>
TPIU.PortMode NRZ
TPIU.SWVPrescaler 4.
```

---

**See also**

- TPIU
- TPIU.state

©1989-2020 Lauterbach GmbH

General Commands Reference Guide T 101
TPIU.PortSize
Select interface type and port size of the TPIU

<table>
<thead>
<tr>
<th>Format:</th>
<th>TPIU.PortSize &lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;size&gt;:</td>
<td>1</td>
</tr>
</tbody>
</table>

Specifies the interface type and port size of the TPIU.

Size in case of Parallel Trace:

| 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 16, 18, 20, 24, 32 | Number of trace data signals. TRACE32 supports the listed sizes. A TPIU can support all sizes from 1 to 32 or only a few out of 1 to 32. |
| 8A, 12A, 16A, 16E | Variants of “8”, “12”, “16” in case of SoC from Texas Instruments. The selected size is the same, but additionally the Debug Resource Manager (DRM) gets configured which maps trace signals to output pins:
  - 16A: TRACEDATA[0:15] -> EMU[4:19]

Size in case of HSSTP:

| 1Lane, 2Lane, 3Lane, 4Lane, 5Lane, 6Lane | Number of used differential lanes. |

Size in case of Serial Wire Viewer (SWV) / Serial Wire Output (SWO):

| SWV | Selects SWV/SWO which uses only one signal. |

See also
- TPIU
- TPIU.state
- <trace>.PortSize

©1989-2020 Lauterbach GmbH
**TPIU.RefClock**

Set up reference clock for HSSTP

<table>
<thead>
<tr>
<th>Format:</th>
<th>TPIU.RefClock [/&lt;option&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;:</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Defines the reference clock frequency the serial preprocessor outputs to the target. Defaults depending on architecture:

- PowerPC: bit clock frequency
- TriCore and RH850: 100MHz
- ARM: bit clock frequency

<table>
<thead>
<tr>
<th>OFF</th>
<th>TRACE32 does not send any reference clock to the target.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSC</td>
<td>An asynchronous oscillator will be enabled. Its frequency is architecture dependent.</td>
</tr>
<tr>
<td>1/&lt;x&gt;</td>
<td>A synchronous clock source will be enabled. Its dividers generate a reference clock as a fraction of the bit clock (lane speed), e.g. 100MHz at 5Gbps with divider 1/50. Once a divider is selected, the reference clock will automatically change with the lane speed.</td>
</tr>
</tbody>
</table>

**See also**
- TPIU
- TPIU.state

---

**TPIU.Register**

Display TPIU registers

<table>
<thead>
<tr>
<th>Format:</th>
<th>TPIU.Register [/&lt;option&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;:</td>
<td>SpotLight</td>
</tr>
</tbody>
</table>

Opens the TPIU.Register window, displaying the TPIU registers and the registers of other trace related modules.

| <option>       | For a description of the options, see PER.view. |

**See also**
- TPIU
- TPIU.state

©1989-2020 Lauterbach GmbH
TPIU.RESet

Resets the settings in the TPIU.state window to their default values and re-configures the TPIU registers on the target.

See also
- TPIU
- TPIU.state

TPIU.state

Display TPIU configuration window

Displays the TPIU.state configuration window.

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

Exceptions:
- The setting TPIU.ON and TPIU.OFF is read-only. The setting depends on the selected trace mode (Analyzer, Onchip, ...).
- The Trace button opens the main trace control window (Trace.state).
- The List button the main trace list window (Trace.List).

See also
- TPIU
- TPIU.PortClock
- TPIU.Register
- TPIU.SyncPeriod
- TPIU.CLEAR
- TPIU.PortMode
- TPIU.RESet
- TPIU.IGNOREZEROS
- TPIU.PortSize
- TPIU.SWVPrescaler
- TPIU.NOFLUSH
- TPIU.RefClock
- TPIU.SWVZEROS
TPIU.SWVPrescaler

Set up SWV prescaler

Format: TPIU.SWVPrescaler <divisor>

Default: 1.

In case of TPIU.PortMode NRZ, the bit rate of the Serial Wire Viewer / Serial Wire Output is derived by dividing the CPU frequency. The command TPIU.SWVPrescaler sets up the divisor, which can range from 0x1 to 0x1000 (1. to 4096.).

Examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPIU.PortMode NRZ TPIU.SWVPrescaler 7.</td>
<td>NRZ coding at a 7th of the CPU clock</td>
</tr>
<tr>
<td>TPIU.PortMode NRZ TPIU.SWVPrescaler 10.</td>
<td>NRZ coding at a 10th of the CPU clock</td>
</tr>
</tbody>
</table>

See also
- TPIU
- TPIU.state

TPIU.SWVZEROS

Workaround for a chip bug

Format: TPIU.SWVZEROS [ON | OFF]

Default: OFF.

Activates a workaround for a chip bug affecting SWV/SWO data of a certain device.

See also
- TPIU
- TPIU.state

©1989-2020 Lauterbach GmbH
Sets the number of regular TPIU packets which will be output to the trace stream between two synchronization packets.

**What are synchronization packets?** Synchronization packets are periodic starting points in the trace stream, which allow the recorded flow trace data to be decoded. The result can then be visualized in the `<trace>*` windows of TRACE32, e.g. the Trace.List or the Trace.PROfileChart.sYmbol window. A visualization of the flow trace data is usually *not possible without* synchronization packets in the trace stream.

<table>
<thead>
<tr>
<th>&lt;packets&gt;</th>
<th>If omitted, then the default number of regular packets between synchronization packets is chosen by the debugger or the chip.</th>
</tr>
</thead>
</table>

**In this example**, the number of regular packets is 1024.

```
RP ... RP SP RP ... RP SP RP ... RP SP RP ...
1024       1024       1024
```

*RP = regular packet
*SP = synchronization packet

**See also**
- TPIU
- TPIU.state
## TPU

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TPU.BASE</strong></td>
<td>Base address</td>
<td>See command <strong>TPU.BASE</strong> in 'TPU Debugger' (tpu.pdf, page 4).</td>
</tr>
<tr>
<td><strong>TPU.Break</strong></td>
<td>Break TPU</td>
<td>See command <strong>TPU.Break</strong> in 'TPU Debugger' (tpu.pdf, page 10).</td>
</tr>
<tr>
<td><strong>TPU.Dump</strong></td>
<td>Memory display</td>
<td>See command <strong>TPU.Dump</strong> in 'TPU Debugger' (tpu.pdf, page 8).</td>
</tr>
<tr>
<td><strong>TPU.Go</strong></td>
<td>Start TPU</td>
<td>See command <strong>TPU.Go</strong> in 'TPU Debugger' (tpu.pdf, page 10).</td>
</tr>
<tr>
<td><strong>TPU.Register.ALL</strong></td>
<td>Register operation mode</td>
<td>See command <strong>TPU.Register.ALL</strong> in 'TPU Debugger' (tpu.pdf, page 5).</td>
</tr>
<tr>
<td><strong>TPU.Register.NEWSTEP</strong></td>
<td>New debugging mode</td>
<td>See command <strong>TPU.Register.NEWSTEP</strong> in 'TPU Debugger' (tpu.pdf, page 6).</td>
</tr>
<tr>
<td><strong>TPU.Register.Set</strong></td>
<td>Register modification</td>
<td>See command <strong>TPU.Register.Set</strong> in 'TPU Debugger' (tpu.pdf, page 8).</td>
</tr>
</tbody>
</table>

©1989-2020 Lauterbach GmbH
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>See command in 'TPU Debugger'</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU.Register.view</td>
<td>Register display</td>
<td>TPU.Register.view (tpu.pdf, page 7).</td>
</tr>
<tr>
<td>TPU.RESet</td>
<td>Disable TPU debugger</td>
<td>TPU.RESet (tpu.pdf, page 11).</td>
</tr>
<tr>
<td>TPU.SESelect</td>
<td>Select TPU for debugging</td>
<td>TPU.SESelect (tpu.pdf, page 11).</td>
</tr>
<tr>
<td>TPU.view</td>
<td>View TPU channels</td>
<td>TPU.view (tpu.pdf, page 5).</td>
</tr>
</tbody>
</table>
## Trace

**Trace configuration and display**

<table>
<thead>
<tr>
<th>Format:</th>
<th>Trace</th>
<th>&lt;trace&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;trace&gt;:</td>
<td>&lt;trace_method&gt;</td>
<td>&lt;trace_source&gt;&lt;trace_method&gt;</td>
</tr>
</tbody>
</table>

### Trace

- For information, see section **Overview Trace** in this command group description.

### <trace>

- For information, see subsection **About the Command Placeholder <trace>** in this command group description.

### <trace_method>

- For information, see subsection **Replacing <trace> with a Trace Method - Examples** in this command group description.

### <trace_source><trace_method>

- For information, see subsection **Replacing <trace> with Trace Source and Trace Method - Examples** in this command group description.

### NOTE:

There is **NO** period between **<trace_source><trace_method>**.

This syntax convention is reserved for:

- Processing trace data from only one particular trace source, e.g. ITM.
- Processing trace data from more than one trace source, e.g. ITM and HTM.
- Processing trace data from very special trace sources.

### See also

- <trace>.ACCESS
- <trace>.Arm
- <trace>.AutoFocus
- <trace>.AutoStart
- <trace>.BookMark
- <trace>.ComPare
- <trace>.CustomTraceLoad
- <trace>.DisConfig
- <trace>.EXPORT
- <trace>.Find
- <trace>.FindChange
- <trace>.FLOWSTART
- <trace>.GOTO
- <trace>.List
- <trace>.ListVar
- <trace>.Mode
- <trace>.PC
- <trace>.PortType
- <trace>.ADDRESS
- <trace>.AutoArm
- <trace>.AutoInit
- <trace>.AutoTEST
- <trace>.Chart
- <trace>.CustomTrace
- <trace>.DISable
- <trace>.DRAW
- <trace>.FILE
- <trace>.FindAll
- <trace>.FLOWPROCESS
- <trace>.Get
- <trace>.Init
- <trace>.ListNesting
- <trace>.LOAD
- <trace>.OFF
- <trace>.PipePROTO
- <trace>.PROfile

©1989-2020 Lauterbach GmbH
Overview Trace

The command Trace is a general command for trace configuration and trace display. It is available for all kind of trace methods provided by TRACE32. The currently used trace method is displayed under METHOD in the Trace.state window.

For descriptions of the trace methods, see Trace.METHOD.

In this section:

- About the Command Placeholder <trace>
- What to know about the TRACE32 default settings for Trace.METHOD
- Types of Replacements for <trace>
- Replacing <trace> with a Trace Method - Examples
- Replacing <trace> with a Trace Evaluation - Example
- Replacing <trace> with RTS for Real-time Profiling - Example
- Replacing <trace> with Trace Source and Trace Method - Examples
- How to access the trace sources in TRACE32
- List of <trace> Command Groups consisting of <trace_source><trace_method>
- Related Trace Command Groups
About the Command Placeholder `<trace>`

In the TRACE32 manuals, `<trace>` is used as a placeholder for all types of trace commands. As the name `placeholder` implies, it cannot be used directly in the TRACE32 command line. As soon as you type `<trace>.List` at the command line, you receive the error message "unknown command". Consequently, you need to replace `<trace>` with the correct trace command before the command line accepts your input.

What to know about the TRACE32 default settings for Trace.METHOD

The easiest way to replace `<trace>` with a correct command is to type `Trace` at the command line. The meaning of `Trace`, e.g. in `Trace.List`, is then controlled by a sequence of TRACE32 default settings.

1. The TRACE32 `hardware module` connected to your target board determines the `trace method`. And this trace method will be used for recording the trace data. In the header of the `Trace.state` window, you can view the selected trace method.

   TRACE32 determines the default trace method as follows:
   - If the hardware module connected to your target board is a PowerTrace, then the `Analyzer` trace method becomes the default setting for the 1st TRACE32 PowerView GUI. For the other GUIs of an AMP configuration, the default setting is `Trace.METHOD NONE`.
   - If a hardware module other than a PowerTrace is connected to your target board, TRACE32 adjusts the trace method accordingly. For the other GUIs of an AMP configuration, the default setting is `Trace.METHOD NONE`.
   - If the chip has an onchip trace sink, then the `Onchip` trace method becomes the default setting for the 1st TRACE32 PowerView GUI. However, if the onchip trace recording is not yet operational, then the trace method is set to `NONE`. For the other GUIs of an AMP configuration, the default setting is `Trace.METHOD NONE`.
   - If the chip does not have an onchip trace sink, then the `ART` trace method becomes the default setting.
   - If TRACE32 runs in software-only mode as an instruction set simulator, then it is again the `Analyzer` trace method that becomes the default setting.

2. The `Analyzer` trace method is designed to look for a specific `trace source` that generates the program flow trace on the chip. For ARM chips, this trace source is called Embedded Trace Macrocell (ETM). For other chips, the trace source can be NEXUS or a proprietary trace block.

3. All `Trace` commands refer to the selected trace method.

In the following first figure, the arrows illustrate the default settings used by the 1st TRACE32 PowerView GUI.

The second figure shows the effects of the default setting `Trace.METHOD NONE` on all other TRACE32 PowerView GUIs of an AMP configuration.
1st TRACE32 PowerView GUI:

PowerTrace hardware module

Chip

ITM

HTM

ETM*

(...)

*ETM, ITM, and HTM are the names of <trace_sources> on a chip.

All other TRACE32 PowerView GUs: How does a TRACE32 PowerView GUI indicate that the Trace.METHOD is set to NONE?

A In the Trace.state window, the radio options for the trace method selection remain visible. But no trace method is selected. Select the trace method you want to use.

B All other GUI controls in the Trace.state window are temporarily hidden. Their underlying Trace.* commands cannot be successfully executed at the TRACE32 command line either. The only command exceptions are Trace.METHOD and Trace.state.

C The state line displays a white X against a red background.
Types of Replacements for `<trace>`

You can rely on the trace method that TRACE32 selects by default, but you can also select a trace method other than the default. As soon as you have selected the trace method you want in the `Trace.state` window, you can replace the placeholder `<trace>` with:

- **Trace** as explained in the previous section (Click here)
- The name of the trace method you have selected in the `Trace.state` window (Click here)
- Trace evaluation commands (Click here)
- **RTS**, the command for real-time profiling (Click here)
- Names of trace sources immediately followed by the name of the trace methods (Click here)

Replacing `<trace>` with a Trace Method - Examples

You can replace `<trace>` with the name of the selected trace method. The trace method commands are displayed in the `Trace.state` window:

- Analyzer, CAnalyzer, Onchip, ART, LOGGER, SNOOPer, FDX, LA, Integrator, Probe, IProbe, HAnalyzer

**Example 1 for the trace method SNOOPer:**

```
Trace.state ;select the trace method SNOOPer for recording
Trace.METHOD SNOOPer ;trace data.
;<configuration>

;trace data is recorded using the commands Go, WAIT, Break

Trace.List ;display the trace data recorded with SNOOPer ;as a trace listing.
SNOOPer.List ;this is the equivalent and explicit command.
```

**Example 2 for the trace method LOGGER:**

```
Trace.state ;select the trace method LOGGER for recording
Trace.METHOD LOGGER ;trace data.
;<configuration>

;trace data is recorded using the commands Go, WAIT, Break

Trace.List ;display the trace data recorded with LOGGER ;as a trace listing.
LOGGER.List ;this is the equivalent and explicit command.
```
For trace evaluations, you can replace `<trace>` with a trace evaluation command; the name of the trace method is omitted.

The trace evaluation commands are accessible via the TRACE32 softkey bar:

- COVerage, ISTATistic, MIPS, CTS, ETA, BMC

**Example:**

```plaintext
Trace.state ; select the trace method Analyzer for recording
Trace.METHOD Analyzer ; trace data.
; <configuration>

; trace data is recorded using the commands Go, WAIT, Break

COVerage.List ; <trace> is just replaced with the trace evaluation command, since the trace method ; Analyzer is defined above anyway.
```
For real-time profiling, you can replace the placeholder `<trace>` with RTS.

The RTS command is accessible via the TRACE32 softkey bar:

Example:

```
Trace.state ;select the trace method Analyzer for
Trace.METHOD Analyzer ;recording trace data.
;<configuration>

RTS.state
RTS.ON
;<configuration>

Go ;processes the trace data being recorded from
;the target while the target is running.

ISTATistic.ListModule ;ISTATistic windows display real-time
;trace data as long as RTS is switched ON
;(RTS.ON)
```
As stated in the blue Format table, the placeholder <trace> can be replaced with trace commands consisting of <trace_source> and <trace_method>.

These <trace> command groups are accessible via the TRACE32 softkey bar and include for example:

- CoreSightTrace, ETMTrace, ETMAnalyzer, STMAalyzer, CoreSightCAnalyzer, ...
- For an overview, see List of <trace> Command Groups consisting of <trace_source><trace_method>.

Using these <trace> command groups, you can display trace data recorded from one or more trace sources.

**Example for displaying trace data from one trace source**: This script assumes that the CoreSight components of the chip output their trace data to the same trace sink.

```plaintext
Trace.state ;select the trace method Analyzer for recording
Trace.METHOD Analyzer ;trace data.
;<configuration>

ETM.ON ;switch on the trace source from which you want
;<configuration> ;to record trace data, here the ETM.

;trace data is recorded using the commands Go, WAIT, Break

Trace.List ;display the ETM trace data recorded with the
;trace method Analyzer as a trace listing.
Analyzer.List ;this is the equivalent and explicit command.
```
Example for displaying trace data from two trace sources: This script assumes that the CoreSight components of the chip output their trace data to the same trace sink.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace.state</td>
<td>;select the trace method Analyzer for recording trace data.</td>
</tr>
<tr>
<td>Trace.METHOD Analyzer</td>
<td>;trace data.</td>
</tr>
<tr>
<td>ETM.ON</td>
<td>;switch the 1st trace source ETM on.</td>
</tr>
<tr>
<td>HTM.ON</td>
<td>;switch the 2nd trace source HTM on.</td>
</tr>
<tr>
<td>Trace.List</td>
<td>;display the ETM trace data.</td>
</tr>
<tr>
<td>HTMTrace.List</td>
<td>;display the HTM trace data.</td>
</tr>
</tbody>
</table>

;trace data is recorded using the commands Go, WAIT, Break
How to access the trace sources in TRACE32

As you have seen in the previous sections, the **Trace.state** window is the starting point for configuring a trace recording and recording the trace data: It provides an overview of the trace methods [A], and it dynamically adjusts to the trace method you have selected [B].

In addition, the **Trace.state** window displays buttons for each trace source found on the chip [C]. Clicking a button lets you access a `<trace_source>.state` window, where you can configure the selected trace source directly in TRACE32.

**Example:** TRACE32 has found has three trace sources on a QorIQ chip, including a NEXUS trace source [C]. Click the **NEXUS** button to open the **NEXUS.state** window [D]. You can now configure the NEXUS trace source.
### List of `<trace>` Command Groups consisting of `<trace_source>`<trace_method>:

<table>
<thead>
<tr>
<th>Command Group</th>
<th>Analyzer/Trace</th>
<th>Analyzer/Trace</th>
<th>Analyzer/Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>AETAnalyzer</td>
<td>ETMXCAnalyzer</td>
<td>NPKReorderHAnalyzer</td>
<td></td>
</tr>
<tr>
<td>BMONTrace</td>
<td>ETMXHAnalyzer</td>
<td>NPKReorderLA</td>
<td></td>
</tr>
<tr>
<td>CIProbe</td>
<td>ETMXLA</td>
<td>NPKReorderTrace</td>
<td></td>
</tr>
<tr>
<td>CoreSightAnalyzer</td>
<td>ETMXOnchip</td>
<td>OCEANAnalyzer</td>
<td></td>
</tr>
<tr>
<td>CoreSightCAnalyzer</td>
<td>ETMXTrace</td>
<td>OCEANOchip</td>
<td></td>
</tr>
<tr>
<td>CoreSightHAnalyzer</td>
<td>FunnelAnalyzer</td>
<td>OCeaNTrace</td>
<td></td>
</tr>
<tr>
<td>CoreSightLA</td>
<td>FunnelOnchip</td>
<td>PCPOchip</td>
<td></td>
</tr>
<tr>
<td>CoreSightOnchip</td>
<td>HAnalyzer</td>
<td>RTPAnalyzer</td>
<td></td>
</tr>
<tr>
<td>CoreSightOnchip2</td>
<td>HTMAnalyzer</td>
<td>SFTAnalyzer</td>
<td></td>
</tr>
<tr>
<td>CoreSightTrace</td>
<td>HTMCAnalyzer</td>
<td>SFTOnchip</td>
<td></td>
</tr>
<tr>
<td>DDRAnalyzer</td>
<td>HTMHAnalyzer</td>
<td>SFTTrace</td>
<td></td>
</tr>
<tr>
<td>DDROnchip</td>
<td>HTMLA</td>
<td>STMAnalyzer</td>
<td></td>
</tr>
<tr>
<td>DDRTrace</td>
<td>HTMOchip</td>
<td>STMCAnalyzer</td>
<td></td>
</tr>
<tr>
<td>DQMAalyzer</td>
<td>HTMTrace</td>
<td>STMHAnalyzer</td>
<td></td>
</tr>
<tr>
<td>DQMOncip</td>
<td>ITHTrace</td>
<td>STMLA</td>
<td></td>
</tr>
<tr>
<td>DQMTTrace</td>
<td>ITMAnalyzer</td>
<td>STMONchip</td>
<td></td>
</tr>
<tr>
<td>DTMAnalyzer</td>
<td>ITMCAnalyzer</td>
<td>STMONchip2</td>
<td></td>
</tr>
<tr>
<td>DTMCAnalyzer</td>
<td>ITMHAnalyzer</td>
<td>STMTrace</td>
<td></td>
</tr>
<tr>
<td>DTMHAnalyzer</td>
<td>ITMLA</td>
<td>TSIAnalyzer</td>
<td></td>
</tr>
<tr>
<td>ETMAnalyzer</td>
<td>ITMOncip</td>
<td>TSICAnalyzer</td>
<td></td>
</tr>
<tr>
<td>ETMCAnalyzer</td>
<td>ITMTrace</td>
<td>TSIHAnalyzer</td>
<td></td>
</tr>
<tr>
<td>ETMDAnalyzer</td>
<td>MCDSBaseAnalyzer</td>
<td>TSILA</td>
<td></td>
</tr>
<tr>
<td>ETMDCAnalyzer</td>
<td>MCDSBaseCAnalyzer</td>
<td>TSIOnchip</td>
<td></td>
</tr>
<tr>
<td>ETMDHAalyzer</td>
<td>MCDSBaseOnchip</td>
<td>TSITrace</td>
<td></td>
</tr>
<tr>
<td>ETMDLA</td>
<td>MCDSDCAnalyzer</td>
<td>UltraSOCHAnalyzer</td>
<td></td>
</tr>
<tr>
<td>ETMDOncip</td>
<td>MCDSDCACAnalyzer</td>
<td>UltraSOCLA</td>
<td></td>
</tr>
<tr>
<td>ETMDTrace</td>
<td>MCDSDCAOncip</td>
<td>UltraSOCTrace</td>
<td></td>
</tr>
<tr>
<td>ETMHAnalyzer</td>
<td>MCDSDDTUAnalyzer</td>
<td>XGateOnchip</td>
<td></td>
</tr>
<tr>
<td>ETMLA</td>
<td>MCDSDDTUCAalyzer</td>
<td>XTICAnalyzer</td>
<td></td>
</tr>
</tbody>
</table>
## Related Trace Command Groups

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMITrace</td>
<td>Clock management instrumentation trace by Texas Instruments on OMAP4.</td>
</tr>
<tr>
<td>CPTracerTrace</td>
<td>Analyzes and displays CPT trace data.</td>
</tr>
<tr>
<td>HTMRTS</td>
<td>Real-time processing for HTM trace.</td>
</tr>
<tr>
<td>OCPTrace</td>
<td>OpenCoreProtocol WatchPoint trace by Texas Instruments on OMAP4 and OMAP5.</td>
</tr>
<tr>
<td>PMITrace</td>
<td>Power management instrumentation trace by Texas Instruments on OMAP4.</td>
</tr>
<tr>
<td>PrintfTrace</td>
<td>Displays and analyzes software messages.</td>
</tr>
<tr>
<td>SLTrace</td>
<td>Allows to trace and analyze SYStem.LOG events.</td>
</tr>
<tr>
<td>StatColTrace</td>
<td>Statistics collector trace by Texas Instruments on OMAP4 and OMAP5.</td>
</tr>
<tr>
<td>SystemTrace</td>
<td>Displays and analyzes trace information generated by various trace sources.</td>
</tr>
</tbody>
</table>
### Sampling configuration for probes ABCDEF

**PowerIntegrator only**

<table>
<thead>
<tr>
<th>Format:</th>
<th>Integrator.ABCDEF &lt;option&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;:</td>
<td>500MHZ Fixed500MHZ 250MHZ State StatePLL StatePLLBoth</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>500MHZ</td>
<td>Timing Mode 500 MHz for probes ABCDEF only, probes JKLMNO get lost.</td>
</tr>
<tr>
<td>Fixed500MHZ</td>
<td>Timing Mode fixed sampling with 500 MHz for probes ABCDEF only, probes JKLMNO get lost.</td>
</tr>
<tr>
<td>250MHZ</td>
<td>Timing Mode 250 MHz.</td>
</tr>
<tr>
<td>State</td>
<td>State Mode clocked by CLKA or CLKB.</td>
</tr>
<tr>
<td>StatePLL</td>
<td>State PLL Mode, clocked by CLKA or CLKB.</td>
</tr>
<tr>
<td>StatePLLBoth</td>
<td>State PLL Mode, clocked by CLKA or CLKB, for all probes.</td>
</tr>
<tr>
<td>CLKA</td>
<td>Clock A select for State-Mode or State-PLL-Mode.</td>
</tr>
<tr>
<td>CLKB</td>
<td>Clock B select for State-Mode or State-PLL-Mode.</td>
</tr>
<tr>
<td>Falling</td>
<td>sampling on falling edge of selected clock CLKA or CLKB.</td>
</tr>
<tr>
<td>Rising</td>
<td>sampling on rising edge of selected clock CLKA or CLKB.</td>
</tr>
<tr>
<td>DDR</td>
<td>sampling on rising and falling edge of selected clock CLKA or CLKB.</td>
</tr>
<tr>
<td>SAMPLE</td>
<td>sampling delay of selected clock CLKA or CLKB (-3 …+6 ns in steps of 250 ps), State-PLL-Mode only.</td>
</tr>
</tbody>
</table>
The core trace generation logic on the processor/chip generates trace packets to indicate the instruction execution sequence (program flow). TRACE32 merges the following sources of information in order to provide an intuitive display of the instruction execution sequence (flow trace).

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

**Format:**

```<trace>.ACCESS <path>`
```

- **Recorded trace packets:** Uploaded from the source of trace information
- **Program code from target memory:** Read via JTAG interface
- **Symbol and debug information loaded to TRACE32**

---

©1989-2020 Lauterbach GmbH
1. **Trace information should be analyzed while the program execution is running and the debugger has no run-time access to the target memory to read the program code.**

NOACCESS in a trace display window indicates that the debugger can not read the target memory.

![Trace display window](image)

You can overcome this problem by loading the program code to the **TRACE32 virtual memory**.

```
; load the program code additional to the TRACE32 virtual memory
; whenever you load it to the target memory
Data.LOAD.Elf diabc.x /PlusVM
```

2. **Reading the target via JTAG is very slow therefore all trace display and analysis windows are slow.**

You can overcome this problem by loading the program code to the **TRACE32 virtual memory** and by specifying `Trace.ACCESS AutoVM`.

```
; load the program code additional to the TRACE32 virtual memory
; whenever you load it to the target memory
Data.LOAD.Elf diabc.x /PlusVM

; advise TRACE32 to read the target code from the virtual memory

; if no code is loaded to the virtual memory for a program address
; TRACE32 will read the code by using the best practice procedure
Trace.ACCESS AutoVM
```

3. **Trace information should be inspected, but there is no program code available.**

You can overcome this problem by specifying `Trace.ACCESS Denied` to advise TRACE32 not to merge program code information. The **Trace.List** window will list the available program addresses and mark all cycles as unknown.

![Trace list window](image)
Recommended access paths:

**auto**

TRACE32 uses its own best practice procedure to read the program code. *(Note: For the ARM architecture this mode is usually *not* using the DualPort access.)*

**AutoVM**

If the program code for a program address is available via the TRACE32 virtual memory it is read from there. Otherwise the best practice procedure is used.

**VM**

The program code is always read from the TRACE32 virtual memory.

**Denied**

No program code information is read.

Rarely used access paths:

**OVS**

Code overlays are handled by the best practice procedure. If the best practice procedure does not deliver correct results, you can advise TRACE32 to read the program code by using the overlay table.

**CPU**

Advise TRACE32 to read the code via the CPU/core.

**DualPort**

Advise TRACE32 to read the code via the run-time access to the target memory.

See also

- `<trace>.state`
- `Trace`

---

**<trace>.ADDRESS**

Software trace address

Format:  

`<trace>.ADDRESS [<address>]`

**LOGGER.ADDRESS** defines the address of the logger trace buffer control block in target memory.

See also

- `<trace>.state`
- `Trace`
<trace>.Arm

Format: <trace>.Arm

The trace memory and if available the trigger unit are prepared for recording and triggering. It is not possible to read the trace contents while the trace is in Arm state.

For most trace methods it is possible to AutoArm (<trace>.AutoArm) the trace. That means:

- Recording and triggering are prepared whenever the program execution is started.
- Recording and triggering are stopped whenever the program execution is stopped.

This is the default setting.

It is also possible to manually switch off the trace (<trace>.OFF) to read the trace contents and arm it again afterwards.

See also

- <trace>.AutoArm
- <trace>.AutoStart
- <trace>.Init
- Analyzer.STATE()
- CIProbe
- SNOOPer.STATE()
- Trace
- CIProbe
- Analyzer.STATE()
- SNOOPer.STATE()

▲ 'Emulator Functions' in 'FIRE User's Guide'
▲ 'Release Information' in 'Release History'
**<trace>.AutoArm**

Arm automatically

Default: <trace>.AutoArm ON.

- Recording and if available triggering is prepared whenever the program execution is started.
- Recording and if available triggering is stopped whenever the program execution is stopped.

**See also**

- <trace>.Arm
- <trace>.state
- Analyzer.STATE()
- SNOOPer.STATE()
- Trace
- CIProbe
- ‘Emulator Functions’ in ‘FIRE User’s Guide’

**<trace>.AutoFocus**

Calibrate AUTOFOCUS preprocessor

The command **Trace.AutoFocus** configures an AutoFocus preprocessor for an error-free sampling on a high-speed trace port.

For preprocessors without AUTOFOCUS technology, but adjustable reference voltage, this command will modify the reference voltage (see **Trace.THreshold**) and try to find a value were the trace capture is free of errors. This might take anywhere from a few up to 30 s.

If available the test pattern generator of the trace port is used to generate the trace data for the auto-configuration. Otherwise a test program is loaded and started by TRACE32.

If a test program is used, TRACE32 attempts to load the test program to the memory addressed by the PC or the stack pointer. It is also possible to define an `<address_range>` for the test program.

```
Trace.AutoFocus ; start the auto-configuration
Trace.AutoFocus 0x24000000++0xfff ; start auto-configuration, load the test program to address 0x24000000
```

©1989-2020 Lauterbach GmbH
If TRACE32 is unable to load the test program the following error message is displayed:
“Don’t know where to execute the test code”.

By default the original RAM contents is restored after the auto-configuration and the trace contents is deleted.

**Accumulate**

If the application program varies the CPU clock frequency, this affects also the trace port and the auto-configuration. In such a case it is recommended to overlay the auto-configurations for all relevant CPU clock frequencies by using the option `/Accumulate`.

**KEEP**

When the auto-configuration is completed, the test pattern generator/test program is started once again to test the correctness of the trace recording. After this test the trace is cleared and an eventually loaded test program is removed from the target RAM.

With the option `/KEEP` the test trace is not cleared and can be viewed with the Trace.List command. If a test program was loaded by TRACE32 it also remains in the target RAM.

**ALTERNATE**

If the trace port provides a test pattern generator, it is always used for the auto-configuration. The option `/ALTERNATE` forces TRACE32 to use its own test program.

This is recommended e.g. if a CoreSight test pattern generator is not stimulating the TRACECLT signal.

**NoThreshold**

Do not calibrate the Trace.THreshold reference voltage.

The option `/Accumulate` allows to overlay several auto-configurations. It is recommended to proceed as follows:

1. Execute the command Trace.AutoFocus at the highest CPU clock frequency.

2. Reduce the CPU clock frequency and execute the command Trace.AutoFocus /Accumulate.

   If a preprocessor with AUTOFOCUS technology is used, the clock and data delays are adjusted, while the termination voltage, the clock reference voltage and the data reference voltage remain unchanged.

3. Repeat step 2 for all relevant frequencies.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace.AutoFocus</td>
<td>; Execute the command for the highest CPU clock</td>
</tr>
<tr>
<td>Trace.AutoFocus /Accumulate</td>
<td>; Re-execute the command for the next lower CPU clock</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Trace.AutoFocus /Accumulate</td>
<td>; Re-execute the command for the lowest relevant CPU clock</td>
</tr>
</tbody>
</table>
A failure in the **Trace.AutoFocus** command results in a stop of a PRACTICE script. The following workaround can be used to avoid this behavior:

```plaintext
; go to the label **error_autofocus**: if an error occurred in the script
ON ERROR GOTO error_autofocus
Trace.AutoFocus
; go to the label **end**: if an error occurred in the script
ON ERROR GOTO end
...
end:
ENDDO
error_autofocus:
PRINT %ERROR "Trace.AutoFocus failed. Script is aborted"
ENDDO

**NOTE:** The NEXUS AutoFocus adapter does not support this feature.
The **Trace.AutoFocus** command causes the preprocessor with AUTOFOCUS technology to configure itself. The auto-configuration searches for the best set of reference voltages and assures optimal sampling of the information broadcast by the trace port. The higher the trace port data rate, the more effort is put in the hardware configuration. For trace port data rates higher 200 Mbit/s the command may need up to 7 s for completion.

In contrast to **Trace.TestFocus**, the command **Trace.AutoFocus** does both the hardware configuration as well as a trace port test.

For preprocessors with AUTOFOCUS technology the hardware auto-configuration includes:

- Automatic setup of proper termination voltage to assure signal integrity.
- Automatic setup of clock reference voltage resulting in a stable clock with 50/50 duty cycle.
- Automatic setup of data reference voltage resulting in broad data eyes.
- Automatic setup of clock and data delays resulting in optimal sampling for each data channel.

The complete auto-configuration executes the following steps:

1. If available the trace port’s test pattern generator is started. Otherwise a test program (maximum size 4 kB) is loaded by TRACE32 to the target RAM and started.

2. A hardware auto-configuration as described above is executed. When the optimal hardware configuration is found the test pattern generator/test program is stopped and the trace data is discarded. After executing the hardware auto-configuration the data eyes and optimal sampling points are known to the TRACE32 software and can be viewed by the user with the **Trace.ShowFocus** command.

3. The test pattern generator/test program is started once again and the program and data flow is recorded to the trace buffer to allow TRACE32 to verify the correctness of the trace recording.

If the self calibration was successful, the following message is displayed in the message line (f=<trace_port_frequency>):

```
B:::
Analyser data capture o.k. (f=156.0MHz)
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R:00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** The trace port frequency does not necessarily equal the CPU clock frequency. E. g. for the ARM-ETM:

- An ETMv1 or ETMv2 operating at HalfRate results in an ETM clock frequency that is half the CPU clock frequency
- An ETMv3 operating with PortMode 1/2 results in an ETM frequency that is a quarter of the CPU clock frequency.

The result of the **Trace.AutoFocus** command can be displayed with the **Trace.ShowFocus** command. If the user wants to verify that the current hardware configuration is complying with the current requirements (e.g. after a frequency change) without wanting to change this configuration, the **Trace.TestFocus** command can be used.
If the auto-configuration fails and you need technical support, please use the **AutoFocus Diagnosis** to prepare all relevant information for the support person.

**See also**

- `<trace>.ShowFocus`
- `<trace>.ShowFocusClockEye`
- `<trace>.ShowFocusEye`
- `<trace>.TestFocus`
- `<trace>.TestFocusEye`
- `Trace`
- `<trace>.state`
- `<trace>.TestFocusClockEye`
- `AUTOFOCUS.OK()
- `<trace>.ShowFocus
- AUTOFOCUS.FREQUENCY()

▲ 'Introduction' in 'AutoFocus User's Guide'
▲ 'Introduction' in 'PowerTrace Serial User's Guide'
▲ 'Installation' in 'ARM-ETM Trace'
▲ 'Release Information' in 'Release History'
The `<trace>.Init` command will be executed automatically, when the user program is started (or stepped through). This causes that

- Trace memory contents is erased and previous records are no longer visible.
- The trigger unit is set to its initial state.
- All used counters are initialized and all used flags are set to OFF.

In combination with the command `<trace>.SelfArm` the trace is able to generate continuous recording and display like a trace snapshot.

**See also**

- CIProbe
- `<trace>.Init`
- `<trace>.state`
- Trace
- Analyzer.STATE()
- SNOOPer.STATE()
- 'Emulator Functions' in 'FIRE User’s Guide'
- 'Release Information' in 'Release History'

### `<trace>.AutoStart`

**Automatic start**

The `<trace>.AutoStart` command will execute the `<trace>.Init` automatically, when a specified break event (as defined in the emulator trigger unit) is encountered and a user program is re-started with the command Go or Step.

**See also**

- `<trace>.Arm`
- `<trace>.state`
- Trace
- Analyzer.STATE()
- 'Emulator Functions’ in ‘FIRE User’s Guide’
If the CPU is running and the trace is in OFF state, the command `<trace>.Arm` will be executed. If the `<trace>.AutoInit` function is activated, the command `<trace>.Init` will be executed prior to the `ARM` command, otherwise the trigger unit will be reset. After the trace goes to the `OFF` or `break` state again (by trace trigger break or full trace in stack mode) all windows are updated and the measurement is continued. The measurement will be stopped, if a display window of the trace is entered.

```
'Analyzer.Program'

(  ADDR AlphaB flags    ; access to first element of the array
      Sample.enable if AlphaB ; sample access to address flags
      BREAK if AlphaB     ; break analyzer for display
 )
Analyzer.Mode Fifo    ; standard mode
Analyzer.AutoTEST ON  ; select automatic test
Analyzer.AutoInit OFF  ; endless recording
Analyzer.List         ; display the accesses
Go
Port.OFF             ; switch mode to timing analyzer
Port.Timing          ; create a display window
Port.TSelect ALways  ; make 200.us long sampling
Port.TMode High      ; start continuous measurement
Port.TDelay 200.us    ; start continuous measurement
Port.AutoTEST ON
```

©1989-2020 Lauterbach GmbH
The memory accesses to flags are displayed dynamically in the list window.

; Mark all function entries with AlphaB and function exits with BetaB
Break.SetFunc

; Program the analyzer
Analyzer.ReProgram ~/demo/analyzer/perf

; Switch the trace buffer to stack mode
Analyzer.Mode Stack

; Limit the trace size
Analyzer.SIZE 1000.

; Select automatic test
Analyzer.AutoTEST ON

; Single measurements
Analyzer.AutoInit ON

; Display a statistic window
Analyzer.STATistic.Func

Go

A detailed dynamic performance analysis is displayed by using the trace to record the function entries and exits.

See also
- <trace>.state
- Trace
- 'Emulator Functions' in 'FIRE User's Guide'
Sets a trace bookmark in the trace listing. A small yellow rectangle next to the record number indicates a trace bookmark.

The **BookMark.List** window provides an overview of all trace bookmarks. Clicking a yellow trace bookmark takes you to the location of that trace bookmark. Additionally, you can use the **Goto** button in a `<trace>.List` window to jump to a bookmarked trace record.

**Example 1:**

```plaintext
; create a trace bookmark named "BM2" for the trace record -120000.
Trace.BookMark "BM2" -120000.

Trace.List DEFault \Track   ; list the trace contents
BookMark.List               ; display all bookmarks in a list
```
Example 2 shows how to create a bookmark 0.300ms after the zero-time reference point. The optional steps are included in this example to let you view on screen what happens behind the scenes.

```
;optional step: In the trace listing, the TIME.ZERO column is displayed
;as the first column, followed by the DEFault columns
Trace.List TIME.ZERO DEFault /Track

;optional step: go to the first trace record, i.e. the record with the
;lowest record number
Trace.GOTO Trace.FIRST()

;set the zero-time reference point to the first trace record
ZERO.offset Trace.RECORD.TIME(Trace.FIRST())

Trace.BookMark "BM3" 0.300ms ;create a bookmark 0.300ms after the
;zero-time reference point

Trace.GOTO "BM3" ;optional step: got to the new bookmark

BookMark.List ;optional step: display all bookmarks
```

See also

- `<trace> List`
- `<trace> state`
- `Trace`
- `<trace> BookMarkToggle`
- `<trace> GOTO`
- `BookMark`
- `BookMark.Create`
- `BookMark.EditRemark`

▲ 'BookMark' in 'General Commands Reference Guide B'
Toggles a single trace bookmark

Switches a single trace bookmark on or off. TRACE32 executes the same command when you right-click in a <trace>.List window, and then choose Toggle Bookmark. The resulting bookmark names are auto-incremented 1, 2, 3, etc. User-defined bookmark names can be created via the command line. A small yellow rectangle next to the record number indicates a trace bookmark.

Example:

```
Trace.List Time.Zero Default /Track ;list the trace contents

;let's toggle two trace bookmarks with user-defined names
Trace.BookMarkToggle "TStart" -Trace.Records() ;bookmark at first record
Trace.BookMarkToggle "TEnd"   -1.              ;bookmark at last record

BookMark.List        ;display all bookmarks in a list
```

See also

- ▲ 'BookMark' in 'General Commands Reference Guide B'
- ©1989-2020 Lauterbach GmbH
Format: `<trace>.Break`

The trace is stopped and the trace storage is ready for read-out.
The `<trace>.Chart` command group allows to display the analyzed trace information graphically. Examples are:

- Function run-time (`Trace.Chart.Func`)
- Time chart (`Trace.Chart.sYmbol`)
- Task run-time (`Trace.Chart.TASK`)
- Variable contents (`Trace.Chart.VarState`)

### Parameters

This section describes the optional `<trace_area>` parameters of the `<trace>.Chart` command group.

| `<record_range>` | Defines which part of the trace buffer is displayed. See example. |
| `<record>`      | Defines which trace record is centered on the x-axis when the window is opened. Records at the beginning or end of the x-axis are not centered. See example. |
| `<time>`        | Defines which timestamp is centered on the x-axis when the window is opened. Timestamps at the beginning or end of the x-axis are not centered. **NOTE**: Only zero-time timestamps can be used as `<time>` parameters. You can display the zero-time timestamps in a `Trace` window by adding the `TimeZero` option to `Trace.Chart.*` or by adding the `Time.Zero` column to `Trace.List`. See examples. |
| `<timerange>`   | Defines which timestamp is displayed on left of the x-axis when the window is opened. **NOTE**: Only zero-time timestamps can be used as `<timerange>` parameters. You can display the zero-time timestamps in a `Trace` window by adding the `TimeZero` option to `Trace.Chart.*` or by adding the `Time.Zero` column to `Trace.List`. See example. |
### General Options

This section describes the options of the `<trace>.Chart` command group.

#### <timescale>

Rule of thumb: The smaller the `<timescale>` value, the higher the resolution and the wider the chart in the data area of a `<trace>.Chart.*` window.

The `<timescale>` parameter defines the display scaling as time per character. It is useful for printing operations and allows to print out any timing chart in a fixed scale on multiple pages.

- See example.
- For the units of measurement, see “Parameter Types” (ide_user.pdf).

#### <trace_bookmark>

Defines which bookmark position is centered on the x-axis when the window is opened. Bookmark positions at the beginning or end of the x-axis are not centered.

**NOTE**: You can only use the names of trace bookmarks, which are created with the `<trace>.BookMark` command.

See example.

---

### General Options

This section describes the options of the `<trace>.Chart` command group.

#### Track

The cursor in the `<trace>.Chart` window follows the cursor movement in other trace windows. Default is a time tracking. If no time information is available tracking to record number is performed. The zoom factor of the `<trace>.Chart` window is retained, even if the trace content changes.

#### ZoomTrack

Same as option Track. If the tracking in performed with another `<trace>.Chart` window the same zoom factor is used.

#### Sort

Specify sorting criterion for analyzed items. For almost all commands the analyzed items are displayed in the order they are recorded by default.

Details on the sorting criterion can be found at the description of the command `Trace.STATistic.Sort`.

#### INCremental

Intermediate results are displayed while TRACE32 PowerView is processing the trace analysis (default).

#### FULL

TRACE32 PowerView displays the result when the processing is done.
**FILE**

Use the trace contents loaded with the command `<trace>.FILE`.

**TASK `<task_magic>`, etc.**

Operating system task in OS-aware debugging and tracing.

See also *“What to know about the Task Parameters”* (general_ref_t.pdf).

Option for **SMP** multicore tracing

<table>
<thead>
<tr>
<th><strong>CORE <code>&lt;n&gt;</code></strong></th>
<th>Time chart is only displayed for the specified core.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SplitCORE</strong></td>
<td>Trace information is analyzed independently for each core. The time chart displays these individual results.</td>
</tr>
<tr>
<td><strong>MergeCORE</strong></td>
<td>Trace information is analyzed independently for each core. The time chart summarizes these results to a single result.</td>
</tr>
<tr>
<td><strong>JoinCORE</strong></td>
<td>Core information is ignored for the time chart.</td>
</tr>
</tbody>
</table>

| **RecScale**     | Display trace in fixed record raster. This is the default. |
| **TimeScale**    | Display trace as true time display, time relative to the trigger point (respectively the last record in the trace). |
| **TimeZero**     | Display trace as true time display, time relative to zero point. For more information about the zero point refer to `ZERO`. |
| **TimeREF**      | Display trace as true time display, time relative to the reference point. For more information about the reference point refer to `<trace>.REF`. |

| **FlowTrace**    | Trace works as a program flow Trace. This option is usually not required. |
| **BusTrace**     | Trace works as a bus trace. This option is usually not required. |
| **CTS**          | Use Context Tracking System to fill trace gaps and then perform `<trace>.Chart`. |
| **NoInline**     | Inline functions are treated as separate functions. The option NoInline can be used to discard inline functions. |
A **Trace.Chart** window may contain a Drag & Drop area which is marked by a straight line.

![Drag & Drop area](image)

Items of interest can be dragged to the appropriate position in the Drag & Drop area with the left mouse button.

The sort order of all items outside of the Drag & Drop area remains unchanged.

![Items can be removed](image)

Items can be removed from the Drag & Drop area by dropping them to the item description area.
Example for `<trace_bookmark>`

Trace.BookMark "begin" 10.005s
Trace.BookMark "end" 10.010s

Trace.Chart.sYmbol "begin" /Track /TimeZERO

Trace.GOTO "begin" ;highlight the bookmark in the chart

BookMark.List ;optional: ;display all bookmarks in a list

A To display the zero-time timestamps on the x-axis, the **TimeZero** option is used.

Example for `<record>`

;print distribution of data values written to flags[3], with the record ;-1950. centered on the x-axis of the window
Trace.Chart.DistriB -1950. Data.L /Filter Address Var.RANGE(flags[3]) \ /RecScale

A To display the record numbers on the x-axis, the **RecScale** option is used.

**NOTE:** The backslash \ can be used as a line continuation character in PRACTICE script files (*.cmm). No white space permitted after the backslash.
Example for `<record_range>`

;print distribution of data values written to flags[3] for the
;record range (-2000.)--(-1000.)
Trace.Chart.Distrib (-2000.)--(-1000.) Data.L /Filter Address \Var.RANGE(flags[3]) /RecScale

A To display the record numbers on the x-axis, the **RecScale** option is used.

Examples for `<time>`

Example 1:

;open the chart window with the zero-time timestamp 10.009s and set the
;<timescale> resolution to 10us (optional)
Trace.Chart.TREE **10.009s** 10us /Track /TimeZero
Trace.GOTO **10.009s** ;highlight the timestamp in the chart

A To display the zero-time timestamps on the x-axis, the **TimeZero** option is used.
Example 2: This PRACTICE script shows how to open the `Trace.Chart.sYmbol` window with a `<time>` parameter that is located 50 microseconds after the 4th occurrence of the HLL symbol `sieve`.

```plaintext
; find the first occurrence of the HLL symbol 'sieve'
Trace.Find , sYmbol sieve

RePeaT 3. ; find the next three occurrences of 'sieve'
Trace.Find

IF FOUND()==TRUE() ; if the 4th occurrences of 'sieve' has been found
{
; get the timestamp of the 4th occurrence and add an offset of 50.us
&time=TRACK.TIME()+50.us

; open the chart window with the calculated timestamp and set the
; <timescale> resolution to 9.5us
Trace.Chart.sYmbol &time 9.5us /Address encode||subst||sieve \ /Track /TimeZero

Trace.GOTO &time ; highlight the timestamp in the chart
}
```

A Location of the calculated timestamp

Example for `<timerange>`

```plaintext
Trace.Chart.sYmbol (10.005s) -- (10.010s) 10.us /Track /TimeZero
```

A Location of the calculated timestamp at 10.005s -- 10.010s
Examples for `<timescale>`

Example 1: Using **WinPrint**, you can print the window content without actually opening the window.

```plaintext
PRinTer.select WIN ;select the printer to which you want to print

;print distribution of data values written to flags[3] for the
;record range (-2000.)--(-1000.), use resolution 10.us per pixel
Address Var.RANGE(flags[3])
```

Example 2: Using the **WinPOS** command, you can assign a name to a window. Then you open the window and print it with **WinPRT** `<name>`. This example illustrates three different `<timescale>` resolutions.

```plaintext
;the following resolutions are used:
;[A] 5.us per pixel, [B] 1.us per pixel, [C] 0.5us per pixel

PRinTer.select WIN ;select the printer to which you want to print

WinPOS , , , , , W0
Trace.Chart.DistrIB (-2000.)--(-1000.) 5.us Data.L /Filter Address \ Var.RANGE(flags[3])

WinPOS , , , , , W1
Trace.Chart.DistrIB (-2000.)--(-1000.) 1.us Data.L /Filter Address \ Var.RANGE(flags[3])

WinPOS , , , , , W2
Trace.Chart.DistrIB (-2000.)--(-1000.) 0.5us Data.L /Filter Address \ Var.RANGE(flags[3])

WinPRT W0 ;print the window named W0
```
## <trace>.Chart.Address

**Time between program events as a chart**

Format:  

```
<trace>.Chart.Address <address1> [<address1> ...]
```

Displays the time interval between up to 8 program events as a chart. The `<trace>.Chart.Address` command is the counterpart of the `<trace>.STATistic.Address` command.

### See also
- `<trace>.Chart`
Example:

```
Trace.Chart.ChildTREE main
```

See also
- `<trace>.Chart`
The command **Trace.Chart.DatasYmbol** analyzes the contents of a pointer graphically.

If a full program and data trace is analyzed, the following command is recommended:

```plaintext
; analyze the contents of the pointer vpchar graphically
Trace.Chart.DatasYmbol /Filter Address vpchar
```
A more effective usage of the trace memory is possible, if only write accesses to the pointer are recorded to the trace.

```plaintext
; set a filter to record only write cycles to the pointer vpchar to the trace
Var.Break.Set vpchar /Write /TraceEnable

... 

; analyze the contents of the pointer
Trace.Chart.DatasYmbol

; analyze the contents of the pointer, sort the result by symbol names
Trace.Chart.DatasYmbol /Sort sYmbol
```

See also
- `<trace>.Chart`
- 'Release Information' in 'Release History'
Distribution display graphically

Format:  \texttt{<trace>.Chart.DistriB [<trace_area>] [/<option>]}

\texttt{<trace_area>}:  \texttt{<trace_bookmark> | <record> | <record_range> | <time> | <time_range> [/<time_scale>]}

\texttt{<option>}:  \texttt{FILE}
\texttt{FlowTrace | BusTrace | CTS}
\texttt{Track}
\texttt{RecScale | TimeScale | TimeZero | TimeREF}
\texttt{Filter <item>}
\texttt{<general_options>}

The distribution of any trace data is displayed if \texttt{<item>} is specified. Without argument the distribution of the addresses is displayed symbolically.

\texttt{<trace_area>}

For parameter descriptions and examples, see Parameters.

\texttt{<option>}

For the general options for all \texttt{<trace>.Chart} commands, refer to \texttt{<trace>.Chart}.

\texttt{Filter <item>}

If no selective trace on the requested \texttt{<item>} is performed, a filter can be used.

If no selective tracing is done, use the option \texttt{/Filter} to filter out the \texttt{<item>} of interest.

\begin{verbatim}
; Display distribution of data value for flags[3]
Trace.Chart.DistriB Data.L /Filter Address V.RANGE(flags[3])

; Print distribution of data value written for flags[3] for the
; record range (-2000.)--(-1000.)
WinPrint.Trace.Chart.DistriB (-2000.)--(-1000.) Data.L /Filter Address V.RANGE(flags[3])
\end{verbatim}
Example for TRACE32-ICE or TRACE32-FIRE:

Perform a selective trace on the data of interest:

```plaintext
Analyzer.ReProgram
  (ADDR AlphaBreak V.RANGE(GlobalStateVar)
    Sample.enable IF AlphaBreak)
Go
... Break
Trace.Chart.Distrib Data.W
```

Example for all trace methods:

```plaintext
Trace.STATistic.Sort Address ; sort chart by addresses
Trace.Chart.Distrib            ; display the time spent at different
                               ; addresses
```

See also
- `<trace>.Chart`
- 'Filter and Trigger - Single-Core and AMP' in 'AURIX Trace Training'
- 'Filter and Trigger - SMP Systems' in 'AURIX Trace Training'
The time spent in different functions is displayed graphically. The measurement is the same as for the command `<trace>.STATistic.Func`.

**Format:**

```
<trace>.Chart.Func [ <trace_area> ] [ / <option> ]
```

**<trace_area>:**

```
<trace_bookmark> | <record> | <record_range> | <time> | <time_range>
[ <time_scale> ]
```

**<option>:**

```
FILE
FlowTrace | BusTrace | CTS
Track
RecScale | TimeScale | TimeZero | TimeREF
```

For parameter descriptions and examples, see Parameters.

For the general options for all `<trace>.Chart` commands, refer to `<trace>.Chart`.

See also

- `<trace>.Chart`
- CTS.CACHE.Chart
<trace>.Chart.GROUP

Group activity chart

Format:  
<trace>.Chart.GROUP [<trace_area>] [/<option>]

<trace_area>:  
<trace_bookmark> | <record> | <record_range> | <time> | <time_range>  
[<time_scale>]

<option>:  
FILE
FlowTrace | BusTrace | CTS
Track
RecScale | TimeScale | TimeZero | TimeREF
Filter <item>

Displays a GROUP time chart (flat statistic).

For parameter descriptions and examples, see Parameters.

For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

Example:

GROUP.Create "INPUT" \jquant2 \jquant1 \jidctred \jdinput /AQUA

GROUP.Create "JPEG" \jdapimin \jdcolor \jddctmgr \jdcoefct /NAVY

Go

Break

Trace.Chart.GROUP

See also
- <trace>.Chart
- GROUP.Create
- 'Release Information' in 'Release History'

©1989-2020 Lauterbach GmbH
### <trace>.Chart.INTERRUPTTREE

Display interrupt nesting

<table>
<thead>
<tr>
<th>Format:</th>
<th>&lt;trace&gt;.Chart.INTERRUPTTREE [trace_area] [option]</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace_area:</td>
<td>trace_bookmark</td>
</tr>
<tr>
<td></td>
<td>[time_scale]</td>
</tr>
</tbody>
</table>

<trace_area> For parameter descriptions and examples, see Parameters.

<option> For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

---

See also
- <trace>.Chart

©1989-2020 Lauterbach GmbH
The time spent in different HLL lines is analyzed graphically.

<trace_area> For parameter descriptions and examples, see Parameters.
<option> For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

Example for TRACE32-ICD and PowerTrace:

If no selective trace is possible use the option /Filter to filter out the module or function of interest.
Example for TRACE32-ICE and TRACE32-FIRE:

If only a specific module or function should be analyzed, perform a selective trace on the module or function:

```
Analyzer.ReProgram
    ( 
        ADDR AlphaBreak V.RANGE(module1) 
        Sample.Enable IF AlphaBreak 
    )
Go
Break
Trace.Chart.Line
```

; sample only instruction from module1
; collect data
; graphical display of HLL analysis

See also
- `<trace>.Chart`

▲ 'Release Information' in 'Release History'

**<trace>.Chart.MODULE**  
**Code execution broken down by module as chart**

Format:  
```
<trace>.Chart.MODULE [ <trace_area> ] [ / <option> ]
```

- `<trace_area>`:  
  - `<trace_bookmark>`
  - `<record>`
  - `<record_range>`
  - `<time>`
  - `<time_range>`

- `<option>`:  
  - `FILE`
  - `FlowTrace` | `BusTrace`
  - `TASK` | `SplitTASK` | `MergeTASK`
  - `Track` | `ZoomTrack`
  - `RecScale` | `TimeScale` | `TimeZero` | `TimeREF`
  - `INCremental` | `FULL`
  - `FILTER` `<item>`
  - `Sort` `<item>`

Displays the code execution broken down by module as chart. The measurement is done similar to the command `<trace>.STATistic.MODULE`.

- `<trace_area>`: For parameter descriptions and examples, see Parameters.
- `<option>`: For the general options for all `<trace>.Chart` commands, refer to `<trace>.Chart`. 
See also
■ <trace>.Chart

<trace>.Chart.Nesting  Show function nesting at cursor position


<trace_area>:  <trace_bookmark> | <record> | <record_range> | <time> | <time_range> | [<time_scale>]

<option>:  FILE | FlowTrace | BusTrace
RecScale | TimeScale | TimeZero | TimeREF
INCremental | FULL
Sort <item>
Track
ZoomTrack
TASK

Shows the function call stack as a time chart.

<trace_area>  For parameter descriptions and examples, see Parameters.

<option>  For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

See also
■ <trace>.Chart
<trace>.Chart.PAddress /Filter Address <address>

The command provides a graphical chart of the instructions that accessed the data address specified by the Filter option similar to the <trace>.STATistic.PAddress command.

Example:

Trace.Chart.PAddress /Filter Address mstatic1

See also

- <trace>.Chart
**<trace>.Chart.PROGRAM**

Code execution broken down by program as chart.

Displays the code execution broken down by program as chart. The measurement is done similar to the command `<trace>.STATistic.PROGRAM`.

- `<trace_area>`: For parameter descriptions and examples, see Parameters.
- `<option>`: For the general options for all `<trace>.Chart` commands, refer to `<trace>.Chart`.

See also
- `<trace>.Chart`

---

**<trace>.Chart.PsYmbol**

Shows which functions accessed data address

The command provides a graphical chart of the functions that accessed the data address specified by the `/Filter` option similar to the `<trace>.STATistic.PsYmbol`.

- Trace.Chart.PsYmbol /Filter sYmbol mstatic1
- Trace.Chart.PsYmbol /Filter sYmbol mstatic1 CYcle Write

See also
- `<trace>.Chart`
The distribution of program execution time at different symbols is displayed as a time chart. This can be used to get a quick overview about the functions sampled in the trace buffer.

**Format:**

```
<trace>.Chart.Symbol [trace_area] [/option]
```

**<trace_area>:**

```
trace_bookmark | record | record_range | time | time_range
[time_scale]
```

**<option>:**

- FILE
- FlowTrace | BusTrace | CTS
- NoInline
- Track
- RecScale | TimeScale | TimeZero | TimeREF
- Address <function1>||<function2> ...
- Address <function1>--><function2>
- Filter Address <function1>||<function2> ...
- Filter Address <function1>--><function2>

For parameter descriptions and examples, see [Parameters](#).

For the general options for all `<trace>.Chart` commands, refer to `<trace>.Chart`.

![Symbol analysis example](image.png)
Example for TRACE32-ICD and TRACE32-PowerTrace:

Go
Break
Trace.STATistic.Sort sYmbol ; sort the result alphabetically

; draw time chart for specified functions, assign time for all other functions to (other)
Trace.Chart.sYmbol /Address func2||func10||sfpDoubleNormalize

; draw time chart for specified functions (address range), assign time for all other functions to (other)
Trace.Chart.sYmbol /Address func2--func10

The **GROUP** command provides more features to structure your time chart.
; filter specified functions out of the address stream
; and draw time chart for filtered trace information
Trace.Chart.Symbol /Filter Address main||func2||func10||func26

init

main

func2

func2

func2b

main

func10

func10

func11

func12

Analysis result

Recording (filtered functions are displayed in black)
Example for TRACE32-ICE and TRACE32-FIRE:

If only a specific module or function should be analyzed, perform a selective trace on the module or function:

```c
Analyzer.ReProgram
(
    ADDR AlphaBreak Y.SECRANGE(\diab555\.text)
    Sample.Enable IF AlphaBreak
)
Go
Break
Trace.STATistic.Sort Ratio
Trace.Chart.sYmbol
```

See also

- `<trace>.Chart`
- `<trace>.STATistic.sYmbol`
- `<trace>.Chart`
- `CTS.CACHE.Chart`
- `CTS.state`
- 'Function Run-Times Analysis’ in ‘ARM-ETM Training’
- 'Flat Function-Runtime Analysis - Single-Core and AMP’ in ‘AURIX Trace Training’
- 'Function Run-Times Analysis - Single’ in ‘Nexus Training’
- 'Release Information’ in ‘Release History’
### Task activity chart

Displays the time spent in different tasks. The measurement is done similar to the command `<trace>.STATistic.TASK`.

**<trace_area>**

For parameter descriptions and examples, see [Parameters](#).

**<option>**

For the general options for all `<trace>.Chart` commands, refer to [Trace Chart](#).

#### E:\Trace.Chart.TASK

<table>
<thead>
<tr>
<th>range</th>
<th>-190.000ms</th>
<th>-180.000ms</th>
<th>-170.000ms</th>
<th>-160.000ms</th>
<th>-150.000ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>(kernel)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(root)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRCE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See also

- `<trace>.Chart`
- 'Release Information' in 'Release History'
- 'OS-Aware Tracing - Single-Core and AMP' in 'AURIX Trace Training'
- 'OS-Aware Tracing - SMP Systems' in 'AURIX Trace Training'
- 'OS-Aware Tracing' in 'Intel® Processor Trace Training'
- 'OS-Aware Tracing - Single Core' in 'Nexus Training'
Task related function run-time analysis (legacy)

For details, refer to `<trace>.Chart.Func`.

See also
- `<trace>.Chart`

Display ISR2 time chart (ORTI)

For parameter descriptions and examples, see Parameters.

For the general options for all `<trace>.Chart` commands, refer to `<trace>.Chart`.

See also
- `<trace>.Chart`
  ▲ 'OS-Aware Tracing - Single-Core and AMP' in 'AURIX Trace Training'
  ▲ 'OS-Aware Tracing - SMP Systems' in 'AURIX Trace Training'
Format:  \texttt{<trace>.Chart.TASKKernel [<trace_area>] [/<option>]}

\texttt{<trace_area>}:  \texttt{<trace_bookmark> | <record> | <record_range> | <time> | <time_range> [<time_scale>]}

Time chart for results of \textit{Trace.STATistic.TASKKernel}.

\texttt{<trace_area>}: For parameter descriptions and examples, see \texttt{Parameters}.

\texttt{<option>}: For the general options for all \texttt{<trace>.Chart} commands, refer to \texttt{<trace>.Chart}.

\textbf{See also}
\begin{itemize}
  \item \texttt{<trace>.Chart}
\end{itemize}
<trace>.Chart.TASKSRV

Service routine run-time analysis

Format:  
<trace>.Chart.TASKSRV [<trace_area>] [/<option>]

<trace_area>:  
<trace_bookmark> | <record> | <record_range> | <time> | <time_range>  
[<time_scale>]

<option>:  
FILE
FlowTrace | BusTrace | CTS
Track
RecScale | TimeScale | TimeZero | TimeREF

The time spent in OS service routines and different tasks is displayed. Service routines that are used by multiple tasks are displayed for each task. The measurement is similar to the command <trace>.STATistic.TASKSRV.

<trace_area>  
For parameter descriptions and examples, see Parameters.

<option>  
For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

See also
- <trace>.Chart
- 'OS-Aware Tracing - Single-Core and AMP’ in 'AURIX Trace Training'
- 'OS-Aware Tracing - SMP Systems’ in 'AURIX Trace Training'
The time different task spent in specific states is displayed. The measurement is similar to the command `<trace>.STATistic.TASKState`.

**<trace_area>**  For parameter descriptions and examples, see Parameters.

**<option>**  For the general options for all `<trace>.Chart` commands, refer to `<trace>.Chart`.

### Graphics

<table>
<thead>
<tr>
<th>State</th>
<th>Graphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>running</td>
<td>solid black bar</td>
</tr>
<tr>
<td>ready</td>
<td>medium blue bar</td>
</tr>
<tr>
<td>waiting</td>
<td>two thin red lines</td>
</tr>
<tr>
<td>suspended</td>
<td>thin grey line</td>
</tr>
<tr>
<td>undefined/unknown</td>
<td>no line</td>
</tr>
</tbody>
</table>

**See also**

- `<trace>.Chart`
Format: `<trace>.Chart.TASKVSINTERRUPT` [`<trace_area>`] [`/<option>`]

`<trace_area>`:  `<trace_bookmark>` | `<record>` | `<record_range>` | `<time>` | `<time_range>`
[`<time_scale>`]

`<option>`:
- FILE
- FlowTrace | BusTrace
- Track
- ZoomTrack
- RecScale | TimeScale | TimeZero | TimeREF
- INCremental | FULL
- Filter `<item>`
- Sort `<item>`

Shows a graphical representation of tasks that were interrupted by interrupt service routines.

For parameter descriptions and examples, see `Parameters`.

For the general options for all `<trace>.Chart` commands, refer to `<trace>.Chart`.

See also
- `<trace>.Chart`
Displays a time-chart for task-related interrupt service routines.

For parameter descriptions and examples, see Parameters.

For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

See also
- <trace>.Chart
- 'OS-Aware Tracing - Single-Core and AMP’ in 'AURIX Trace Training'
- ‘OS-Aware Tracing - SMP Systems’ in 'AURIX Trace Training'
The result of this command shows a graphical chart tree of the function nesting.

For parameter descriptions and examples, see Parameters.

For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

See also
  ▪ <trace>.Chart

<trace>.Chart.Var

Variable chart

The command provides a graphical chart of variable accesses.

For parameter descriptions and examples, see Parameters.

For the general options for all <trace>.Chart commands, refer to <trace>.Chart.
Example:

; Display a graphical chart of all variable accesses:
Trace.Chart.Var /Filter sYmbol mstatic1 /Filter CYcle Write

; Display a graphical chart of write accesses to the mstatic1 variable
Trace.Chart.Var /Filter sYmbol mstatic1 /Filter CYcle Write

See also
■ <trace>.Chart
▲ 'Release Information' in 'Release History'

<trace>.Chart.VarState

Variable activity chart

Format:  
<trace>.Chart.VarState [<trace_area>] [/<option>]

<trace_area>:  
<trace_bookmark> | <record> | <record_range> | <time> | <time_range> 
[<time_scale>]

<option>:  
FILE
FlowTrace | BusTrace | CTS
Track
RecScale | TimeScale | TimeZero | TimeREF
Fill | FillFirst
DECODE <value> ...
Filter <item>

Displays the contents of variables over the time. Each variable access must be sampled with one single CPU cycle. If an address is not a variable it is displayed in form of a single marker. This can be used to track program execution addresses.

For parameter descriptions and examples, see Parameters.

For the general options for all <trace>.Chart commands, refer to <trace>.Chart.
Example 1 for TRACE32-ICE and TRACE32-FIRE:

Perform a selective trace on the data section.

Analyzer.ReProgram
(
    ADDR AlphaBreak Y.SECRANGE(.data)
    Sample.Enable IF AlphaBreak
)
Go
Break

Example 2 for TRACE32-ICE and TRACE32-FIRE:

If no selective trace is possible use the option /Filter to filter out the variables.

Go
Break
Trace.Chart.VarState /Filter Y.SECRANGE(.bss)
**Fill**

Repeat the value instead of displaying the value only directly after the transition.

<table>
<thead>
<tr>
<th>F::Trace.Chart.VarState /Fill</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>range</strong></td>
</tr>
<tr>
<td>ast.word</td>
</tr>
<tr>
<td>vtripplearray[0][0][0]</td>
</tr>
<tr>
<td>vtripplearray[0][0][1]</td>
</tr>
<tr>
<td>flags[7]</td>
</tr>
<tr>
<td>flags[8]</td>
</tr>
<tr>
<td>flags[9]</td>
</tr>
<tr>
<td>flags[10]</td>
</tr>
<tr>
<td>flags[11]</td>
</tr>
</tbody>
</table>

**FillFirst**

Repeat the value without any space instead of displaying the value only directly after the transition.

**DECODE <value>**

Define a decoding for enumeration variables.

`Trace.Chart.VarState /DECODE 2 4 7`

---

**See also**

- `<trace>.Chart`
- 'Filter and Trigger - Single-Core and AMP’ in 'AURIX Trace Training'
- 'Filter and Trigger - SMP Systems’ in 'AURIX Trace Training'
- 'Release Information’ in 'Release History’
**<trace>.CLEAR**  
Clear FDX communication buffers

Format:  
<trace>.CLEAR [<address>]

<trace>:  
FDX

**FDX.CLEAR** clears the communication buffers of a FDX channel. All buffer contents are lost. Without arguments all FDX channels will be cleared.

**<trace>.CLOCK**  
Clock to calculate time out of cycle count information

Format:  
<trace>.CLOCK <frequency>  
<trace>.CLOCK <frequency0> <frequency1> … (SMP tracing only)

Some trace protocols can generate cycle count information. TRACE32 can calculate time information out of the cycle count information if the appropriate clock frequency is specified with the **Trace.CLOCK** command.

For most trace protocols cycle count indicates the number of core clock cycles. That's why **<frequency>** has to be the core clock frequency. Please be aware the specifying the core clock frequency only makes sense if the frequency was constant while recording.

**Example for the ARM-ETM:**

```
ETM.TImeMode CycleAccurate
Trace.CLOCK 800.MHz
```

If the cores of an SMP run at different speeds, the frequency can be specified per core.

```
ETM.TImeMode CycleAccurate
Trace.CLOCK 800.MHz 600.MHz 1.GHz
```

See also

- ETM.CycleAccurate
**<trace>.CLOSE**

Format: `<trace>.CLOSE [<address>]`

Closes the file related to the given FDX channel. Without arguments all files used by FDX are closed.

**<trace>.ComPare**

Compare trace contents

Format: `<trace>.ComPare [<record_range>] [<record_number>] [{<items>}] [{<options>}]`

<options>:
- **Tolerance <count>**
- **FILE**
- **Back**

Compares the trace contents. If the command `<trace>.ComPare` is used without arguments the previous compare is repeated.

<table>
<thead>
<tr>
<th>&lt;item&gt;</th>
<th>Only the given &lt;item&gt; … are compared.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;record_range&gt;, &lt;record_number&gt;</td>
<td>If &lt;record_range&gt; and &lt;record_number&gt; are not used, a comparison of the complete trace is performed.</td>
</tr>
<tr>
<td>FILE</td>
<td>Compare the trace contents with the loaded file. See also Trace.FILE.</td>
</tr>
<tr>
<td>Back</td>
<td>Compare backwards.</td>
</tr>
<tr>
<td>Tolerance &lt;count&gt;</td>
<td>When external asynchronous data are traced, a jitter in the signal will result in different sampling data. In this case the precision of the compare function may be controlled by the option Tolerance.</td>
</tr>
</tbody>
</table>

The compare function will set the pointers for the tracking option. All analyzer windows, which are in track mode, will follow these pointers.

For valid channel names refer to the:

- Processor Architecture Manuals
- Target Guides FIRE
- Target Guides ICE
Examples:

; compare the current trace contents from record (500.--1000.) with the current trace contents starting at record number 5000. with regards to the address
Trace.Compare (500.--1000.) 5000. Address

; load saved trace contents
Trace.FILE old_trace

; compare the current trace contents from record (500.--1000.) with the loaded trace contents starting at record number 300. with regards to the data on byte 0
Trace.ComPare (500.--1000.) 300. Data.B0 /FILE

; load saved trace contents
Trace.FILE old_trace

; compare the complete current trace contents with the complete loaded trace contents with regards to the data on byte 0
Trace.ComPare Data.B0 /FILE

; Repeat the previous compare
Trace.ComPare

; load saved trace contents
Trace.FILE old_trace

; compare the complete current trace contents with the complete loaded trace contents with regards to the data and address
Trace.ComPare Data Address /FILE
; compare against file TEST1 on line RXD
Port.FILE TEST1 ; load reference file
Port.ComPare RXD /Tolerance 3. /FILE ; compare line RXD
IF FOUND()
   PRINT "Difference found"
... ; print result if difference
Port.ComPare ; will be found
; search for next difference

See also
■ <trace>.state
■ Trace

<trace>.CSELect Select signal for counter

Format: <trace>.CSELect <channel>

See also
■ Trace
PRACTICE script examples of custom trace demos can be found in the following *_demo.cmm files:

- ~/demo/customtrace/pipe_dll/dll_stp_demo.cmm
- ~/demo/customtrace/pipe_dll/dll_csstm_demo.cmm
- ~/demo/customtrace/pipe_dll/dll_itm_demo.cmm

For details about these files, refer to the readme.txt in the demo folder.

See also
- <trace>.CustomTrace.<label>.COMMAND
- <trace>.CustomTrace.<label>.UNLOAD
- <trace>.CustomTraceLoad
- <trace>.state
- Trace

▲ 'Software Trace with the ITM' in 'CombiProbe for Cortex-M User's Guide'
▲ 'Software Trace with the ITM' in 'uTrace for Cortex-M User's Guide'

<trace>.CustomTrace.<label>.COMMAND Send command to specific DLL

Format 1:  
<trace>.CustomTrace.<label>.COMMAND  <command_line_args>

Sends a command to a specific DLL that has been assigned a user-defined <label>.

See also
- <trace>.CustomTrace

<trace>.CustomTrace.<label>.UNLOAD Unload a single DLL

Format 1:  
<trace>.CustomTrace.<label>.UNLOAD

Unloads a single DLL identified by <label>.

See also
- <trace>.CustomTrace
**<trace>.CustomTraceLoad**  
Load a DLL for trace analysis/Unload all DLLs

| Format 1: | `<trace>.CustomTraceLoad "<name>" <file>` |
| Format 2: | `<trace>.CustomTraceLoad ""` |

**Format 1:** TRACE32 supports a mechanism for passing trace data to a shared library or DLL allowing for custom trace handling. This command loads the shared object.

**Format 2:** When executed with an empty string, the command unloads all DLLs.

| NOTE: | Use the command `<trace>.CustomTrace.<label>.UNLOAD` to unload a single DLL. |

**<name>**  
A user-defined name for the DLL or shared object. TRACE32 supports up to 8 loaded shared objects at any one time. The `<name>` is used to differentiate them.

**<file>**  
A shared library or DLL which is appropriate for your host Operating System. This DLL will receive trace data from TRACE32 and perform custom analysis on it.

**See also**
- `<trace>.CustomTrace`
- `<trace>.state`
- `Trace`

©1989-2020 Lauterbach GmbH
### `<trace>.DISable`

**Disable the trace**

Disables the trace.

**See also**
- CIProbe
- `<trace>.state`
- Trace

### `<trace>.DISableChannel`

**Disable FDX communication**

Disables an FDX communication channel. Without parameters all channels are disabled. Disabling keeps the buffer contents of FDX. Communication can be re-enabled with `FDX.ENableChannel`.

**See also**
- Trace

**Format:**

```
<trace>.DISable
```

**Format:**

```
<trace>.DISableChannel [ <address> ]
```
For background information and examples about how to use the `<trace>.DisConfig` command group, see:

- **“PowerIntegrator Trace DisConfig Application Note”** (powerintegrator_app_dc.pdf)

**See also**

- `<trace>.DisConfig.CYcle`
- `<trace>.DisConfig.FlowMode`
- `<trace>.DisConfig.RESet`
- `<trace>.DisConfig.view`
- `Trace`
- ‘General Function’ in ‘PowerIntegrator Trace DisConfig Application Note’

---

### `<trace>.DisConfig.CYcle`  Trace disassemble setting

**Debugger only**

**Format:**

```
<trace>.DisConfig.CYcle "<name> [, <ext>]" <cycle>
```

**<cycle>:**

- `Read` `<definition>`
- `Write` `<definition>`
- `Fetch` `<definition>`
- `FLOW` `<definition>`
- `Fetch1` `<definition>`
- `ReadOrFetch` `<definition>`
- `ReadSpecial` `<definition>`
- `WriteSpecial` `<definition>`
- `MERGE ["<name>" <offset>...]`

**<definition>:**

- `TransientStrobe [<time>] [<channels>]`

  - `Strobe[2|3] [[<channel> [Low | High | Falling | Rising]]`

  - `Strobe[2|3]Sample [Last | Next | AT number] [<channel> [Low | High | Falling | Rising]]`

  - `Address[2]Sample [Last | Next | AT number] [<channel> [Low | High | Falling | Rising]]`

  - `Address[2] [<channels>]

  - `Address[2] SHift <value>`
The command `<trace>.DisConfig.CYcle` informs the trace software where to find program-fetch, data-read and data-write cycles in a not qualified trace recording which was taken by the PowerProbe or PowerIntegrator. With this information a standard bus trace listing can be generated.

- **AddressBase** `<address>`
- **Data[2]Sample** [Last | Next | AT number] [ `<channel>` [Low | High | Falling | Rising]]
- **Data[2]** `<channels>`
- **Data[2]SHift** `<value>`
- **DataUnknown**
- **DataWidthUnknown**
- **SpaceID | SpaceIDSample**
- **Word | Group | Integrator.<x> | eXt.<x>**

The command `<trace>.DisConfig.CYcle` informs the trace software where to find program-fetch, data-read and data-write cycles in a not qualified trace recording which was taken by the PowerProbe or PowerIntegrator. With this information a standard bus trace listing can be generated.

- **<name>, <ext>** “name” is displayed in the cycle-type row of the Trace.List window. Its length is limited to 7. The "ext" is not displayed but used to differ between cycle types. Example: “rd_byte,0” --> rd_byte.
  - This way it is possible to define different cycle types (rd_byte,0; rd_byte,1 …) which are displayed in the same way (rd_byte)

- **<cycle>** Is used by the trace disassembler
  - **Read**: data read cycle
  - **Write**: data write cycle
  - **Fetch**: program fetch cycle
  - **Fetch1**: first program fetch code of an instruction
  - **ReadOrFetch**: data-read or program-fetch cycle. The disassembler will do the final decision out of the program flow knowledge
  - **ReadSpecial**: special cycle (e.g. dma)
  - **WriteSpecial**: tbd.
  - **MERGE**: merge the data of multiple cycles

- **<definition>** Defines where to find a `<cycle>` in the trace, where to find the appropriate address and data, and how to display them.

### See also
- `<trace>.DisConfig`
- `<trace>.DisConfig.view`

©1989-2020 Lauterbach GmbH

General Commands Reference Guide T 183
<trace>.DisConfig.FlowMode

Enable FlowTrace analysis

Debugger only

Format:  

<trace>.DisConfig.FlowMode [ETMB | ETMK | OFF]

Enables the analysis of certain FlowTrace protocols like ARM-ETM.

OFF  FlowTrace analysis disabled
ETMB ARM-ETM FlowTrace analysis enabled, Mictor probe AB in use.
ETMK ARM-ETM FlowTrace analysis enabled, Mictor probe JK in use.

See also
■ <trace>.DisConfig  ■ <trace>.DisConfig.view

<trace>.DisConfig.RESet

Reset trace disassemble setting

Debugger only

Format:  

<trace>.DisConfig.RESet

Resets the trace disassemble setting.

See also
■ <trace>.DisConfig  ■ <trace>.DisConfig.view

<trace>.DisConfig.view

Trace disassemble setting

Format:  

<trace>.DisConfig.view

See also
■ <trace>.DisConfig.CYcle  ■ <trace>.DisConfig.RESet

©1989-2020 Lauterbach GmbH
The `<trace>.DRAW` command group can be used to plot the values of recorded trace data against time.

### See also

- `<trace>.state`
- `<trace>.DRAW.Var`
- `Data.DRAWXY`
- `<trace>.DRAW.channel`
- `<trace>.DRAW.Data`
- `<trace>. DRAW.channel`
- `Trace`
- `ClProbe`
- `Data.IMAGE`
- `<trace>.DRAW.channel`
- `<trace>.DRAW.Data`
- `Data.DRAW`
- `Var.DRAW`
- `<trace>.DRAW.Data`
- `Data.DRAWFFT`
- `Data.DRAWXY`
- `Data.IMAGE`
- `Var.DRAW`
- `<trace>.DRAW.Data`
- `Data.DRAWFFT`

▲ ‘Introduction’ in ‘Application Note for the Trace.DRAW Command’
▲ ‘How to use the PROTOanalyzer’ in ‘DigRF Protocol Analyzer’
▲ ‘Release Information’ in ‘Release History’

---

### `<trace>.DRAW.channel`

**Plot no-data values against time**

**Format:**

```
<trace>.DRAW.channel [<record_range>] [%<format>] [<items …>] [/<options>]
```

<trace>.Chart.Draw (deprecated)

- `<record_range>`:
  - `<start_record>--><end_record>`

- `<format>`:
  - `Decimal. [<width>]`
  - `DecimalU. [<width>]`
  - `Hex. [<width>]`
  - `Float. [leee | leeeDbl | leeeeXt | leeeMFFP | …]`

- `<width>`:
  - `DEFAULT` | `Byte` | `Word` | `Long` | `Quad` | `TByte` | `HByte`

- `<items>`:
  - `ENERGY.Abs` | `POWER` ...

- `<options>`:
  - `<draw_options>` | `FILE` | `BusTrace` | `RecScale` | `TimeScale` | `TimeZero` | `TimeREF` | `MinMax` | `Color` | `LOG` | `FIRST` | `Filter` `<filter_items>` | `Track` | `ZoomTrack`

- `<draw_options>`:
  - `Points` | `Vector` | `MarkedVector` | `Steps` | `Impulses`

- `<filter_items>`:
  - `<range>` | `<address>` | `<bitmask>`

Plot specified `<item>` against time. This command is mainly used to plot no-data items.
The example below shows a temperature measurement recorded by a logic analyzer. The `Trace.DRAW.channel` command is used to show the temperature profile.

See also

- `<trace>.DRAW`
- 'Release Information’ in 'Release History'
<trace>.DRAW.Data

Plot data values against time

| Format 1: | <trace>.DRAW.Data %<format> {<data_address> | <data_range>} [/<options>] |
| Format 2: | <trace>.DRAW.Data [<start> | <range>] [hscale] <vscale> <v_offset> [%<format>] {<data_address>} [/<options>] |

| <format>: | [<radix>.]<width> |
| <start>: | <bookmark> | <record> | <time> |
| <range>: | <record_range> | <time_range> |
| <vscale>: | <float> |
| <v_offset>: | |
| <hscale>: | <time> |
| <radix>: | Decimal | DecimalU | Hex | HexS | OCTal |
| <width>: | DEFAULT | Byte | Word | Long | Quad | … |
| <option>: | [<source_option>] [<draw_option>] [<scale_option>] [<zoom_option>] |
| <source_option>: | FILE | FlowTrace | BusTrace |
| <draw_option>: | Points | Vector | MarkedVector | Steps | Impulses | MinMax | LOG |
| <scale_option>: | RecScale | TimeScale | TimeZero | TimeREF |
| <zoom_option>: | Track | ZoomTrack |

Plots one or more data values. An introduction to the usage of the Trace.DRAW.Data command is provided in “Application Note for the Trace.DRAW Command” (app_trace_draw.pdf).
Example for a core trace, trace filter applied, <time_range> specified:

```plaintext
PRIVATE &address
&address=0x40004068

Break.Set &address /Write /TraceEnable
Trace.List TIme.Zero DEFault

Trace.Find Address &address Data 0x95
IF FOUND()
  (  
    ZERO Trace.RECORD.TIME(TRACK.RECORD())  
  )
; <time_range>       %<format>           <data_address>
Trace.DRAW.Data 0us--20.ms %Decimal.Byte &address /TimeZero
```

Example that scales the plot vertically into a window of the specified size:

```
WinPOS 20. 20. 120. 13. 16. 2. W001  
; <vscale> <v_offset> %<format>       <data_address>
Trace.Draw.Data 1.9 0.0 %DecimalU.Byte 0x40004068
```
Description of the `<trace>.DRAW.Data Parameters`

**<record_range>**

To specify the record range, open the **Trace.List** window. Then choose the record range you want to plot.

**Example:**

```
; Opens the Trace.List window.
Trace.List /Track

; Plot graph for the specified record range.
Trace.DRAW.Data (-131072.)--(-121070.) %Hex.Word 0x1234 /Track
```

**Supported Radices**

- **Decimal, DecimalU** (Decimal Unsigned), **Hex**, and **Float** format the display of the y-axis.
- **Float.** The following floating-point formats are available:
  - Ieee | IeeeDbl | IeeeEx | IeeeQuad | IeeeXt10 | IeeeRev | IeeeS | IeeeDblS | IeeeDblT |
  - MFFP | Pdp11 | Pdp11Dbl | RTOSUH | RTOSUHD |
  - Dsp16 | Dsp16C | Dsp16Fix | Dsp32Fix |
  - M56 | M560 | M561 | LACCUM |
  - Fract8 | Fract16 | Fract24 | Fract32 | Fract48 | Fract64 |
  - UFract8 | UFract16 | UFract24 | UFract32 | UFract48 | UFract64 | Fract40G |
  - MICRO | MICRO64 | MILLI | MILLI64 | NANO64 | PICO64

**Basic Options**

**FILE:** Visualizes the trace contents loaded with the command `<trace>.FILE`.

**BusTrace:** This option is usually not required. It switches off the FlowTrace decoder. In the bus trace mode, all valid bus cycles are sampled.
**RecScale**: The resolution of the x-axis is based on records, e.g. if timestamps are not available. The record numbers are displayed on the x-axis.

[Image: RecScale.png]

**TimeScale**: The resolution of the x-axis is based on timestamps. The timestamps are displayed on the x-axis.

[Image: TimeScale.png]

**TimeZero**: Displays the trace as a real-time display, time relative to the zero point. For more information about the zero point refer to **ZERO**.

**TimeREF**: Displays the trace as a real-time display, time relative to the reference point. For more information about the reference point refer to **<trace>.REF**.

**FIRST** `<address>`: Defines which address contains the first part of the data value if the data value cannot be sampled within one bus cycle (e.g. a 16 bit data value on a 8 bit data bus).

See also
- `<trace>.DRAW`
- 'Release Information' in 'Release History'
### Trace.DRAW.Var

Plot variable values against time

| Format 1: | `$<trace>.DRAW.Var\ %[<format>] \{<var>\} [l<options>]$` |
| Format 2: | `$<trace>.DRAW.Var\ [<start> | <range>] [ <hscale>] <vscale> <v_offset>\ [%<format>] \{<var>\} [l<options>]$` |
| $<start>$: | `$<bookmark> | <record> | <time>` |
| $<range>$: | `$<record_range> | <time_range>` |
| $<vscale>$: | `$<float>` |
| $<v_offset>$: |  |
| $<hscale>$: | `$<time>` |
| $<format>$: | `DEFault | STandDard | Decimal | Hex` |
| $<option>$: | `[$<source_option>] [ <draw_option>] [ <scale_option>] [ <zoom_option>]` |
| $<source_option>$: | `FILE | FlowTrace | BusTrace` |
| $<draw_option>$: | `Points | Vector | MarkedVector | Steps | Impulses | MinMax | LOG` |
| $<scale_option>$: | `RecScale | TimeScale | TimeZero | TimeREF` |
| $<zoom_option>$: | `Track | ZoomTrack` |

Plots the value changes of one or more variables against time, based on the recorded trace information. An introduction to the usage of the `Trace.DRAW.Var` command is provided in “Application Note for the Trace.DRAW Command” (app_trace_draw.pdf).
Example for a core trace with data trace enabled, no trace filter applied:

```plaintext
; plot value of a single variable
;      %<format>    <var>
Trace.DRAW.Var %DEFault mstatic1

; plot values of two variables
; colors are assigned by TRACE32
Trace.DRAW.Var %DEFault mstatic1 fstatic fstatic2

; plot values of three variables
; colors are assigned by TRACE32
; <display_option> Steps
Trace.DRAW.Var %DEFault mstatic1 fstatic fstatic2 /Steps
```
Example for a core trace, trace filter applied, $<$record_range$>$ specified:

```plaintext
; advice trace generation logic to only generate trace messages for
; write accesses to variable vchar
Var.Break.Set vchar /Write /TraceEnable

; plot values of variable vchar for
; specified $<$record_range$>$
;                     %$<$record_range$>$  %$<$format$>$  <var>
Trace.DRAW.Var (-30000.)--(-29000.) %DEFault vchar
```

Example for a core trace, trace filter applied, $<$time_range$>$ specified:

```plaintext
Var.Break.Set vchar /Write /TraceEnable
Trace.List Time.Zero DEFault

Trace.Find Address Var.RANGE("vchar") Data 0x95
IF FOUND()
(
    ZERO Trace.RECORD.TIME(TRACK.RECORD())
)
;                     %$<$time_range$>$  %$<$format$>$  <var>  /$<$scale_option$>$
Trace.DRAW.Var 0us--20.ms %DEFault vchar /TimeZero
```

Example that scales the plot vertically into a window of the specified size:

```plaintext
WinPOS 20. 20. 120. 13. 16. 2. W001
;                      <vscale> <<v_offset> <var>
Trace.Draw.Var 1.9 0.0 vchar
```

Example that
- specifies an starting point for the plot
- specifies an horizontal scale
- specifies a vertical scale
- specifies a vertical offset:

```plaintext
WinPOS 20. 20. 120. 13. 16. 2. W001
Trace.Find Address Var.RANGE("vchar")
IF FOUND()
;                     <record>      <hscale> <vscale> <v_offset> <var>
Trace.DRAW.Var TRACK.RECORD() 100us 1.9 0.0 vchar
```
Perform a selective trace on the data of interest.

Analyzer.ReProgram
{
  ADDR AlphaBreak V.RANGE(P1)  ; sample all accesses to P1
  Sample.Enable IF AlphaBreak
}
Go  ; collect data
Break
Trace.DRAW %DEFault P1  ; Graphical display of P1

Multi-channel trace of three variables:

Analyzer.ReProgram
{
  ADDR AlphaBreak x_axis
  ADDR BetaBreak y_axis
  ADDR CharlyBreak z_axis

  Sample.Enable IF (AlphaBreak||BetaBreak||CharlBreak)&&Write
}
Go  ; collect data
Break
Trace.DRAW %DEFault x_axis y_axis z_axis  ; plot

See also
- <trace>.DRAW
- 'Release Information' in 'Release History'
- 'Filter and Trigger - Single-Core and AMP' in 'AURIX Trace Training'
### Operation mode

**<trace>.Enable**

| Format: | `<trace>.Enable [ALways | Running]` |

If the port analyzer is in timing mode (not slave), it will be run either totally free or controlled by the emulator operation.

- **ALways**: The analyzer can be used independently of the state of the emulator.
- **Running**: The port analyzer is enabled only, while the emulator system is in real-time emulation. When running to a breakpoint or stopped by an asynchronous event, the timing analyzer will be stopped, too. The trigger system of the port analyzer is not activated until the emulation is started.

**<trace>.EnableChannel**

Enable FDX communication

| Format: | `<trace>.EnableChannel [address]` |

Enables the data transfer over a FDX channel. Without parameters all existing FDX channels are enabled.

©1989-2020 Lauterbach GmbH
Using the `<trace>.EXPORT` command group, you can export trace data for processing in other applications. Various export file formats are available, including ASCII, binary, PGT, VERILOG, etc.

**NOTE:** The various export formats are primarily designed for import into other applications. Trace data exported with the `<trace>.EXPORT.*` commands can only be imported back into TRACE32 if you inform the debugger about all the trace-relevant circumstances.

We recommend the following approach if you want to view and analyze recorded trace data in a subsequent TRACE32 session:
1. Save the trace data to file using `<trace>.SAVE`.
2. To load this file back into TRACE32, use `<trace>.LOAD`.

**See also**
- `<trace>.SAVE`
- `<trace>.state`
- `<trace>.EXPORT.Ascii`
- `<trace>.state`
- `<trace>.EXPORT.Bin`
- `<trace>.EXPORT.VERILOG`
- `<trace>.EXPORT.CSVFunc`
- `<trace>.EXPORT.BRANCHFLOW`
- `<trace>.EXPORT.flow`
- `<trace>.EXPORT.TASKEVENTS`
- `<trace>.EXPORT.MTV`
- `<trace>.EXPORT.TracePort`
- `<trace>.EXPORT.VHDL`
- `<trace>.IMPORT`

▲ 'Release Information' in 'Release History'
▲ 'Further NEXUS Trace Analysis' in 'ARM Application Note for MXC Chips'
▲ 'Further NEXUS Trace Analysis' in 'StarCore Application Note for MXC Chips'
Exports the trace contents to an ASCII file. White spaces are used as delimiters.

<table>
<thead>
<tr>
<th>&lt;option&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILTER</td>
<td>Exports only records matching the filter. For an example, see below.</td>
</tr>
<tr>
<td>ShowRecord</td>
<td>Includes the trace record numbers in the export file.</td>
</tr>
</tbody>
</table>

Example:

```
Trace.EXPORT.Ascii ~~\myfile.ad (-120000.)--(-1.) /ShowRecord \
    /FILTER ADDRESS Var.RANGE(sieve)
```

The backslash \ is used as a line continuation character. No white space permitted after the backslash.

**See also**

- `<trace>.EXPORT`  
- `<trace>.EXPORT.flow`
Exports the trace contents to a file in binary format. This command is used to export logic analyzer (PowerProbe, Integrator, IProbe) recordings. The data is stored in little endian format.

The file starts with a text header describing item names and byte size of each item. Each record begins with an 8-byte timestamp (1 ns per tick), followed by the selected items in the order as given in the command. Each item has a minimum width of 1 byte (max. 8 byte). The following options are available:

- **FILE**: Exports the trace contents loaded with `<trace>.FILE.`
- **NoDummy**: Exclude records which do not hold flow information (do not use when exporting logic analyzer data).
- **NoHeader**: The resulting file does not contain a header.
- **NoTimeStamps**: The records do not contain the 8 byte timestamp.
- **NoFetch**: Exclude control cycles from export.

**Example**: This script export data from a parallel port recorded with the IProbe.

```plaintext
;define the data word of the port, connected to signals ip.00...ip.07
NAME.WORD W.PARPORT ip.00 ip.01 ip.02 ip.03 ip.04 ip.05 ip.06 ip.07

;export analyzer data
IProbe.EXPORT.Bin pardat.ad W.PARPORT /NoHeader

;show resulting file: one record has 9 byte (W.PARPORT has 1 bytes)
DUMP pardat.ad /WIDTH 9
```

![Screen capture of exported binary file]

**See also**

- `<trace>.EXPORT`
- `<trace>.EXPORT.flow`

©1989-2020 Lauterbach GmbH

General Commands Reference Guide T 198
### <trace>.EXPORT.BRANCHFLOW

Export branch events from trace data.

<table>
<thead>
<tr>
<th>Format:</th>
<th><code>&lt;trace&gt;.EXPORT.BRANCHFLOW &lt;file&gt; [&lt;record_range&gt;] [/&lt;options&gt;]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;option&gt;</code>:</td>
<td><code>TRaceRecord</code></td>
</tr>
</tbody>
</table>

Exports the branch events from the trace data.

- **TRaceRecord**: Branch events are exported with trace record numbers.
- **NOINNER**: Only branch events that jump to the current symbol are exported. The internal branch is not exported.
- **NOSYMBOL**: Branch events are exported with addresses instead of symbols.
- **CALLer**: Branch events are exported with caller events.

#### See also
- `<trace>.EXPORT`  
- `<trace>.EXPORT.flow`
Export the function nesting to a CSV file

Exports the function nesting of the recorded trace data to a CSV file for processing by an external tool.

Format: `<trace>.EXPORT.CSVFunc <file> [<trace_area>]`

- `<trace_area>`: `<string>`
- `<range>`
- `<value>`
- `<timerange>`

The default extension of the file name is `.csv`.

Example:

```
;export the entire function nesting
Analyzer.EXPORT.CSVFunc ~~\csvfunc_all.csv

EDIT ~~\csvfunc_all.csv
```

See also
- `<trace>.EXPORT`
- `<trace>.EXPORT.flow`
- 'Release Information' in 'Release History'
Exports the trace contents for postprocessing by an external analysis tool.

The trace contents can only be exported when the trace is in **OFF** or **break** state. Please refer to the **Trace.state** command for more information.

The default export format is binary. A description of the binary format is given at the end of this command description.

<table>
<thead>
<tr>
<th><code>&lt;file&gt;</code></th>
<th>The default extension of the file name is <code>.ad</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Exports the trace contents loaded with <code>&lt;trace&gt;.FILE</code>.</td>
</tr>
<tr>
<td>ZIP</td>
<td>File is compressed with the gzip archive format.</td>
</tr>
</tbody>
</table>

In the case of an SMP system, the following options are provided:

<table>
<thead>
<tr>
<th>MergeCORE</th>
<th>The trace information for all cores is exported.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE <code>&lt;number&gt;</code></td>
<td>Only the trace information for the specified core is exported.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SplitCORE</th>
<th>Same as MergeCORE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>JoinCORE</td>
<td>Same as MergeCORE.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;option&gt;</code>:</th>
<th>CORE <code>&lt;number&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>MergeCORE</td>
</tr>
</tbody>
</table>

Format: `<trace>.EXPORT.flow` `<file>` [ `<record_range>` ] [ / `<options>` ]
Binary File Format Header and Data Structure

When an exported file contains a file header (not the case e.g. for /ByteStream, /CoreByteStream, ...) it has the following format:

<table>
<thead>
<tr>
<th>Byte Nr.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..31</td>
<td>Export file header string (“trace32 analyzer export data” 0x1a 0x00)</td>
</tr>
<tr>
<td>32</td>
<td>Reserved (set to zero for IMPORT)</td>
</tr>
<tr>
<td>33</td>
<td>CPU code</td>
</tr>
<tr>
<td>34</td>
<td>Timestamp available flag</td>
</tr>
<tr>
<td>35</td>
<td>Prestore mode flag</td>
</tr>
<tr>
<td>36</td>
<td>Trigger unit available flag</td>
</tr>
<tr>
<td>37</td>
<td>Port analyzer available/mode flag</td>
</tr>
<tr>
<td>38</td>
<td>Analyzer type</td>
</tr>
<tr>
<td>39</td>
<td>Reserved</td>
</tr>
<tr>
<td>40</td>
<td>Length of one record in bytes (0x20)</td>
</tr>
<tr>
<td>41..43</td>
<td>Reserved</td>
</tr>
<tr>
<td>44..47</td>
<td>Number of records in file (if record number can exceed 32 bits, e.g. Trace.Mode.STREAM, calculate number of records based on file size)</td>
</tr>
<tr>
<td>48..51</td>
<td>Record number of last recorded record</td>
</tr>
<tr>
<td>52..55</td>
<td>Reference record number</td>
</tr>
<tr>
<td>56..63</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte Nr.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..3</td>
<td>Cycle information flags:</td>
</tr>
<tr>
<td></td>
<td>Bit 0: data cycle</td>
</tr>
<tr>
<td></td>
<td>Bit 1: program cycle</td>
</tr>
<tr>
<td></td>
<td>Bit 6: write cycle</td>
</tr>
<tr>
<td></td>
<td>Bit 8: Power Architecture MPC5XXX: read/write cycle of peripheral NEXUS bus master</td>
</tr>
<tr>
<td></td>
<td>Bit 21: FLOW ERROR</td>
</tr>
<tr>
<td></td>
<td>Bit 25: FIFO OVERFLOW</td>
</tr>
<tr>
<td></td>
<td>Bit 31: OWNERSHIP Cycle</td>
</tr>
<tr>
<td>Byte Nr.</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| 4       | Data byte enable mask  
|         | Bit 0: Byte 0 valid  
|         | Bit 1: Byte 1 valid  
|         | ... |
| 5       | CPU specific information  
|         | SH2A I-bus marker (bit meaning is device specific):  
|         | Bit 0: iadma bus  
|         | Bit 1: idma bus  
|         | Bit 2: icpu1 bus  
|         | Bit 3: icpu2 bus  
|         | ARM Bustrace:  
|         | Bit 0: EXEC signal (relevant only when SYStem.Option EXEC is set to ON)  
|         | ARM Flowtrace (ETM/PTM):  
|         | Bit 1: Thumb Mode  
|         | Bit 2: ARM Mode  
|         | Bit 5: not executed  
|         | Bit 6: executed |
| 6       | Reserved |
| 7       | Core number (only on SMP targets) |
| 8..11   | Address (bus/data) |
| 12..15  | Address (upper part or program flow address) |
| 16..23  | Data bytes (64 bits) |
| 24..31  | Timestamp (time relative to ZERO in ns) |

**See also**

- <trace>.EXPORT
- <trace>.EXPORT_BIN
- <trace>.EXPORT_CSVFunc
- <trace>.EXPORT_MTV
- <trace>.EXPORT_TracePort
- <trace>.EXPORT_VERILOG
- <trace>.EXPORT_Ascii
- <trace>.EXPORT_BRANCHFLOW
- <trace>.EXPORT_Func
- <trace>.EXPORT_TASKEVENTS
- <trace>.EXPORT_VCD
- <trace>.EXPORT_VHDL
**<trace>.EXPORT.Func**

Export function nesting

**Format:**

```
<trace>.EXPORT.Func <file> [<record_range>] [/<options>]
```

**<option>:**

- FILE
- BusTrace
- ZIP

Exports the function nesting from the trace contents to a binary file.

Exported function nestings contain the function entries and exits as well as task switches with task entries and exits. Function nestings are displayed in the `<trace>.ListNesting` window.

<table>
<thead>
<tr>
<th>&lt;file&gt;</th>
<th>The default extension of the file name is <em>ad</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;</td>
<td>For a description of the options, see <code>&lt;trace&gt;.EXPORT.flow</code>.</td>
</tr>
</tbody>
</table>

**Example:**

```
Analyzer.EXPORT.Func ~~~\trace.ad (-131072.)--(-100000.)
```

**See also**

- `<trace>.EXPORT`
- `<trace>.EXPORT.flow`
Exports a trace recording in the MCDS Trace Viewer format.

Format: `<trace>.EXPORT.MTV <file> [<record_range>] [/<options>]`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FILE</code></td>
<td>Exports the trace contents loaded with <code>&lt;trace&gt;.FILE</code>.</td>
</tr>
<tr>
<td><code>BusTrace</code></td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td><code>NoDummy</code></td>
<td>Exclude records which do not hold flow information (do not use when exporting logic analyzer data).</td>
</tr>
</tbody>
</table>

The default extension of the file name is `*.mcds`.

See also
- `<trace>.EXPORT`
- `<trace>.EXPORT.flow`
Generates a CSV file that contains task event information and time information.


See also

- `<trace>.EXPORT`  
- `<trace>.EXPORT.flow`

▲ 'Release Information' in 'Release History'
## <trace>.EXPORT.TracePort

Export trace packets as recorded at trace port

### Format:

```
<trace>.EXPORT.TracePort <file> [<record_range>] [/<options>]
```

### <option>:

- `FILE`
- `ZIP`

- `FullByteStream` | `ByteStream` | `CoreByteStream` | `TimedByteStream` | `TPStream` | `TimedCoreByteStream` (ETMv3 only) | `NibbleStream`

Exports the recorded trace data in a low-level binary format. Available options depend on the used processor architecture and trace port.

### <file>

The default extension of the file name is `*.bin`.

| **ByteStream** | Exports the byte stream broadcast by the ETM (same as `TP` column if command `Trace.List TP DEFault` is used). |
| **CoreByteStream** | Similar to the option `ByteStream`, but strips away synchronisation patterns (continuous mode) and trace source identifiers (e.g. in case of multicore systems). The exported data is that shown in the `TPC` column in the command `Trace.List TPC DEFault`. By default, the data corresponding to the currently active core is exported (selected by the `CORE` command), but this can be overridden by the `/CORE <number>` option. |
| **TimedByteStream** | Exports the byte stream broadcast by the ETM together with the `Time.Zero` timestamp information. For a description of the file format, see below. |
| **TPStream** | Power Architecture only. Exports NEXUS packets received through Aurora interface. |
| **TimedCoreByteStream** (ETMv3 only) | Exports the unwrapped byte stream broadcast by the ETM together with the `Time.Zero` timestamp information. This format also supports multiple cores in SMP configuration. |
| **FullByteStream** | Exports the trace data in the format that allows to re-import it using the command `LA.IMPORT <file>`. |
| **NibbleStream** | Exports just pure STP data, excluding non-STP headers (STP = System Trace Protocol). |
The **TimedByteStream** format consists of two-byte records; possible formats are:

- **0y0xxxxxx** `<tracedata_byte>`
  - `xxxxxx`: Time relative to previous records (in nanoseconds).

- **0y10xxxxxx 0yxxxxxxxx**
  - `xxxxxx`: Time relative to previous record (bits 7 to 12).
  - `xxxxxxxx`: Upper bits (bits 13 to 20).

- **0y11000xxx 0yxxxxxxxx**
  - `xxx`: Selects which part of the absolute time is transferred.
  - `xxxxxxxx`: Byte of absolute timestamp.

- **0y11001000 0yxxxxxxxx**
  - `xxxxxxxx`: Selects to which core the following data belongs (only in **CoreByteStream** with SMP).

---

**See also**
- [trace].EXPORT
- [trace].EXPORT.flow

---

**<trace>.EXPORT.VCD**

*Export trace data in VCD format*

Format: 

```
<trace>.EXPORT.VCD <file> [<record_range>] [items …] [/<option>]
```

Exports the trace contents collected by the TRACE32 logic analyzers PowerProbe and PowerIntegrator to a file in VCD format.

<table>
<thead>
<tr>
<th><code>&lt;file&gt;</code></th>
<th>The default extension of the file name is <code>.vcd</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;option&gt;</code></td>
<td>For a description of the options, see <code>&lt;trace&gt;.EXPORT.flow</code>.</td>
</tr>
</tbody>
</table>

---

**See also**
- [trace].EXPORT
- [trace].EXPORT.flow

---
### <trace>.EXPORT.VERILOG

Export trace data in VERILOG format

| Format: | `<trace>.EXPORT.VERILOG <file> [<record_range>] [<items> …] [/<option>]` |
|-----------------------------------------------|

Exports the trace contents collected by the TRACE32 logic analyzers PowerProbe and PowerIntegrator to a file in VERILOG format.

| `<file>` | The default extension of the file name is *.v. |
| `<option>` | For a description of the options, see `<trace>.EXPORT.flow`. |

See also
- `<trace>.EXPORT`
- `<trace>.EXPORT.flow`

### <trace>.EXPORT.VHDL

Export trace data in VHDL format

| Format: | `<trace>.EXPORT.VHDL <file> [<record_range>] [<items> …] [/<option>]` |
|-----------------------------------------------|

Exports the trace contents collected by the TRACE32 logic analyzers PowerProbe and PowerIntegrator to a file in VHDL format.

| `<file>` | The default extension of the file name is *.vhd. |
| `<option>` | For a description of the options, see `<trace>.EXPORT.flow`. |

See also
- `<trace>.EXPORT`
- `<trace>.EXPORT.flow`
Load a file into the file trace buffer

Loads trace data from a file into a dedicated file trace buffer on the host. Typically this feature is used to analyze data in a simulator or to compare different recordings.

Format:  \(<trace>.FILE <file> [/Config]\)

<table>
<thead>
<tr>
<th>&lt;file&gt;</th>
<th>The default extension of the file name is *.ad.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>Restore analyzer and NAME settings contained in &lt;file&gt;. Only applicable for Trace.METHOD Probe and Trace.METHOD Integrator.</td>
</tr>
</tbody>
</table>

Example: To use the file trace buffer as source for trace-related commands, the commands need to be invoked with the additional parameter /FILE

Trace.FILE myfile.ad
Trace.List /FILE

Windows working on trace contents loaded with the Trace.FILE command are marked with a red label FILE in the bottom-left corner:

Trace-related commands without the parameter /FILE keep operating on the trace data stored in the "normal" trace buffer which is filled when recording data using the analyzer hardware (e.g. PowerTrace, PowerProbe, PowerIntegrator).
Using the `file trace buffer` and the “normal” trace buffer concurrently allows to compare trace data stored in a file from a previous recording with recently recorded data as shown in the following example:

| Trace.FILE test4          ; load trace contents from test4.ad |
|---------------------------|-------------------------------------------------------------|
| Trace.List /FILE          ; display loaded trace contents    |
| Trace.Chart.Symbol /FILE   ; works on loaded trace data       |

; compare the recently recorded trace with the trace contents loaded ; from test4.ad regarding to the addresses Trace.ComPare , Address /FILE

**NOTE:** In addition to `Trace.FILE` there is a command `Trace.LOAD` for loading trace data from a file into the “normal” trace buffer. Therefore data loaded with `Trace.LOAD` is treated as if it was recently recorded by the analyzer hardware. As a consequence all standard trace commands automatically work on the loaded via `Trace.LOAD` (without specifying additional parameters).

See also

- `<trace>.LOAD`
- `<trace>.state`
- `Trace`
- 'Release Information' in 'Release History'
Searches for matching items in the given range of trace records. The default search range is the complete trace. When the command is invoked without parameters, the previous search is repeated.

If the search finds a matching trace record, the PRACTICE function `FOUND()` will return `TRUE()`. If a matching trace record was found, `TRACK.RECORD()` returns the record number of the matching record.

Details about the `<trace>.Find` command can be found in “Application Note for the Trace.Find Command” (app_trace_find.pdf).
### Examples:

```plaintext
; find matching address and data,  
; start search at the beginning of the trace recording
Trace.Find Address 0x100--0x200 Data.B 0x55
; find next match
Trace.Find

; find matching address and data,  
; start search at the beginning of the trace recording  
; print all matching records
Trace.Find Address 0x100--0x200 Data.B 0x55
WHILE FOUND()
  (   
    PRINT "Matching record: " TRACK.RECORD()
    Trace.Find
  )

; find any instruction of the function sieve, 
; start search at the end of the trace recording
Trace.Find Address Var.RANGE(sieve) /Back
```
; find specified data value in record range (-1000.)--(-700.)
; start search at the beginning of the trace recording
Trace.Find (-1000.)--(-700.) Data 0x100

; find specified data,
; start search at the beginning of the trace recording
Trace.Find Data 0x00--0xAA

; find write accesses to variable flags with specified data values
; start search at specified record number down to the end of the
; trace recording
Trace.Find -3224833. Address V.RANGE("flags") Data 0x00--0xaa CYcle Write
; find next match
Trace.Find

; find read access to variable flags[3]
; start search at the specified record number up to the beginning of the
; trace recording
Trace.Find -3224832. Address Var.RANGE("flags[3]") CYcle Read /Back

; find any trace information assigned to the group sieve
; start search at the beginning of the trace recording
Trace.Find GROUP "sieve"

; find all trace entries with a TIme.Back time between 500.us--700.us
Trace.Find TIMe.Back 0.500us--0.700us

; find ptrace cycle which contains address 0x40000B7C
Trace.Find FAddress 0x40000B7C

; find one of the specified data values
Trace.Find Data 0x5--0x44 OR Data 0x55 OR Data 0x0

; search for rising edge of
; the NMI signal
Trace.Find NMI ON AT -1. NMI OFF
; search for trace entry generated by core 0 in an SMP system
Trace.Find, CORE 0

; searching for pin RXD high
Port.Find Port.RXD High

; searching for pin RXD high and pin CTS low
Port.Find Port.RXD High Port.CTS Low

; searching for pin RXD high or pin CTS low
Port.Find Port.RXD High OR Port.CTS Low

; searching for pin RXD changing from low to high
Port.Find Port.RXD Low AT 1. Port.RXD High

; searching for pin RXD changing from low to high, but staying high for
; minimum 2 clock cycles
Port.Find Port.RXD Low AT 1. Port.RXD High AT 2. Port.RXD High

See also

■ <trace>.FindAll  ■ <trace>.FindChange  ■ <trace>.state  ■ Trace
■ FOUND()  ■ FOUND.COUNT()
▲ 'The Trace Find Dialog' in 'Application Note for the Trace.Find Command'
▲ 'Release Information' in 'Release History'
▲ 'Displaying the Trace' in 'Training FIRE Analyzer'
Find all specified entries in trace

```
Format:  <trace>.FindAll [<record_number> | <record_range>] <items> ... [/<options>]

<option>
  Back
  FILE
  FlowTrace | BusTrace
  <other_options>
```

Searches for and displays all entries matching the item specification. Without range, the complete trace memory is searched for matching entries.

- **Back**
  The option Back reverses the direction of the search command.

- **BusTrace**
  The trace works as a bus trace. This option is usually not required.

- **FILE**
  Takes trace memory contents loaded by Trace.FILE.

- **FlowTrace**
  The trace works as a program flow trace. This option is usually not required.

- **List**
  Change the default display of the result.

- **<other_options>**
  Details about the <trace>.FindAll command can be found in “Application Note for the Trace.Find Command” (app_trace_find.pdf).

**Example:**

```
Trace.FindAll, sYmbol sieve /List TIme.Zero DEFault
```

**See also**

- <trace>.Find
- <trace>.FindChange
- <trace>.state
- Trace
- FOUND()
- FOUND.COUNT()

▲ 'Release Information’ in 'Release History'
<trace>.FindChange

Search for changes in trace flow

Format:

\[
<\text{trace}>.\text{FindChange} \begin{cases} [<\text{record_number}> | <\text{record_range}>] & \{[<\text{items}>]\} \end{cases} \begin{cases} [\langle\text{options}\rangle]\end{cases}
\]

**<items>:**

OR
<channels>
AT <offset>

**<option>:**

Back
FILE
FlowTrace | BusTrace
<other_options>

Searches for entries in the given range where the specified items have new values. Without range the entry is searched within the complete trace memory. Without items the command searches for changes in program flow. This is useful to search for the end of a complex program loop, or in general to search for “something happens” in a traced program flow.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACK</td>
<td>Reverses the direction of the search command.</td>
</tr>
<tr>
<td>FILE</td>
<td>Takes trace memory contents loaded by Trace.FILE.</td>
</tr>
<tr>
<td>FlowTrace</td>
<td>The trace works as a program flow trace. This option is usually not required.</td>
</tr>
<tr>
<td>BusTrace</td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
</tbody>
</table>

<other_options> Details about the <trace>.FindChange command can be found in “Application Note for the Trace.Find Command” (app_trace_find.pdf).

See also

- <trace>.Find
- <trace>.FindAll
- <trace>.state
- Trace

©1989-2020 Lauterbach GmbH
Processes all trace data in the analyzer and calculates the instruction flow for all of it. This is in contrast to \texttt{<trace>.FLOWSTART} which discards the processing results and thus indirectly causes a reprocessing of the limited set of trace data required to draw the currently open windows (reprocessing on demand).

The command is used mostly for diagnostic purposes.

\textbf{See also}\n
\begin{itemize}
\item <trace>.state
\item Trace
\end{itemize}

\texttt{<trace>.FLOWSTART} \\
\textbf{Restart flowtrace processing} \\
\texttt{Debugger only}

Format: \texttt{<trace>.FLOWSTART [<address>]}\n
Discards all results from previous decoding of instruction flow. This indirectly causes a reprocessing of the limited set of trace data required to draw the currently open windows (reprocessing on demand). Effectively the decoding of flow information is done again “from the start”.

The command is typically used when the memory contents at the time of decoding was wrong and the decoding is therefore incorrect (contains flow errors). The command is executed after providing a correct memory image (e.g. by activating chip selects) to re-initialize the flow processing.

The optional address parameter can be used to indicate the address of the first instruction executed by the processor. In this way the debugger can correctly decode code sequences even before the first sync message appears in the trace stream.

\textbf{See also}\n
\begin{itemize}
\item <trace>.state
\item Trace
\item FOUND()
\end{itemize}
### <trace>.Get

**Display input level**

<table>
<thead>
<tr>
<th>Format:</th>
<th><code>&lt;trace&gt;.Get [%&lt;format&gt;] [&lt;item&gt; ...]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;items&gt;</code>:</td>
<td><code>%&lt;format&gt;</code></td>
</tr>
<tr>
<td>Default</td>
<td>ALL</td>
</tr>
<tr>
<td>Run</td>
<td></td>
</tr>
</tbody>
</table>
| CYcle | Data[.<subitem>] | BDATA | List[.<subitem>]
| Address | BAddress | FAddress |
| sYmbol | sYmbolN | PAddress | PsYmbol | Var |
| Time[.<subitem>] |
| FUNC | FUNCr | FUNCVar | IGNORE |
| LeVel | MARK[.<marker>] | FLAG[.<flag_index>]
| Trigger | Trigger.A | Trigger.B |
| SPARE |
| `<special_lines>` |

<table>
<thead>
<tr>
<th><code>&lt;format&gt;</code>:</th>
<th>Ascii</th>
<th>BINary</th>
<th>Decimal</th>
<th>Hex</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>HighLow</td>
<td>Timing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimeAuto</td>
<td>TimeFixed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEN <code>&lt;size&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;option&gt;</code>:</th>
<th>FILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track</td>
<td></td>
</tr>
<tr>
<td>FlowTrace</td>
<td>BusTrace</td>
</tr>
<tr>
<td>Mark <code>&lt;item&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

Displays the current state of all input lines. The format of the channel definition is similar to the `<trace>.View` command. This command can be executed, while the port analyzer is running.

For valid channel names refer to the:

- Processor Architecture Manuals
- Target Guides FIRE
- Target Guides ICE
Displays the state of all port lines in hex and HIGH/LOW format.

```
E::Port.Get
record  p.2  p.30  p.31  p.32  %d  p.5  %Hex  p.4  %Ascii  p.x

1.00  00  LOW  LOW  LOW  LOW  LOW  33  HIGH  HIGH  LOW
05  50  p.51  p.52  p.53  p.54  p.55  p.56  p.57  p.x  p.x0  p.x1  p.x2  p.x3  p.x4  p.x5
LOW  LOW  FF  HIGH  HIGH  HIGH  HIGH  FF  HIGH  HIGH  HIGH  p.13  p.14  p.15  p.16  p.17  p.0  p.00  p.01  p.02  p.03  p.04  p.05  p.06  p.07
HIGH  HIGH  HIGH  HIGH  07  HIGH  HIGH  HIGH  HIGH  HIGH  FF  HIGH  HIGH  HIGH
```

Displays the state of port lines P2 in binary format, lines P3.0, P3.1 and P3.2 in timing waveform, port lines P5 in decimal format, port lines P4 in hex format and port PX in ASCII format.

```
E::Port.Get  ALL
record  p.ipl0  p.ipl1  p.ipl2  p.bclr  p.cs0  p.cs1  p.cs2  p.cs3  p.pb8
p.iack1  p.tin1  p.tout1  p.tin2  p.tout2  p.wdog  p.rxd1  p.txd1  p.rclk1
p.tclk1  p.cts1  p.rts1  p.cd1  p.brg1  p.rxd2  p.txd2  p.rclk2  p.tclk2
p.rts3  p.cd3  p.brg3  p.x0  p.x1  p.x2  p.x3  p.x4  p.x5  p.x6  p.x7  p.bnk0
```

Displays the state of all port lines in timing waveform.

See also
- `<trace>.state`
- `Trace`
<trace>.GOTO

Move cursor to specified trace record

Format:

<trace>.GOTO "<bookmark>" | <record_number> | <time> [/<options>]

<option>:

FILE
FlowTrace | BusTrace | CORE <number>

Goes to the specified trace record in a Trace.* window by moving the cursor to that trace record. Alternatively, click the Goto button in a Trace.* window, and enter a record number, a time index, or the name of a trace bookmark.

A  To go to a trace <bookmark>, enclose the bookmark name in quotation marks.
B  To go to a trace <record_number>, append a period (.). Mind the + or - sign of the record number.
C  To go to a <time>, prepend a plus or minus sign and append the unit of measurement. To view the <time>, include the TIme.ZERO column in the Trace.List command, as shown in the example below.

<table>
<thead>
<tr>
<th>BusTrace</th>
<th>The trace works as a bus trace. This option is usually not required.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Takes trace memory contents loaded by Trace.FILE.</td>
</tr>
<tr>
<td>FlowTrace</td>
<td>The trace works as a program flow trace. This option is usually not required.</td>
</tr>
<tr>
<td>CORE</td>
<td>The goto operation takes the specified core number into account. Only available for SMP multicore tracing.</td>
</tr>
</tbody>
</table>

Description of Buttons in the Trace Goto Dialog

<table>
<thead>
<tr>
<th>Previous / Next</th>
<th>Go to the previous / next user-defined trace bookmark. Trace bookmarks are created with Trace.BookMark.</th>
</tr>
</thead>
<tbody>
<tr>
<td>First / Last</td>
<td>Go to the first / last trace record.</td>
</tr>
<tr>
<td>Trigger</td>
<td>Go to the trigger record.</td>
</tr>
<tr>
<td>Ref</td>
<td>Go to the reference point, which has been set with the Trace.REF command. You can also set the reference point by right-clicking in a Trace.* window and pointing to Set Ref in the Trace popup menu.</td>
</tr>
</tbody>
</table>

©1989-2020 Lauterbach GmbH
Examples

The Trace.List window is always opened with the Track option. Thanks to the Track option, the subsequent Trace.GOTO command scrolls to the desired trace record in the Trace.List window.

Example 1: Go to a <record_number>.

;open the Trace.List window
Trace.List /Track

;go to this <record_number> in the Trace.List window
Trace.GOTO   -12000.

Example 2: Go to a trace <bookmark>.

;create a trace bookmark named 'BM1' for record -14000.
Trace.BookMark "BM1"   -14000.

;open the Trace.List window
Trace.List /Track

;go to this <bookmark> in the Trace.List window
Trace.GOTO    "BM1"

Example 3: Go to a <time> index.

;      <first_column>  <other_columns>
Trace.List   TIME.ZERO   DEFAULT   /Track

;go to this <time> in the Trace.List window
Trace.GOTO   -5.000ms

See also

- CIProbe
- <trace>.REF
- <trace>.TRACK
- Trace
- Analyzer.RECORD.DATA()
- Analyzer.RECORDS()
- <trace>.BookMark
- <trace>.state
- BookMark
- Analyzer.RECORDADDRESS()
- Analyzer.RECORD OFFSET()
- Analyzer.REF()
The **LA.IMPORT** command group is used to load trace data from a file into TRACE32 and to analyze it just like data recorded with a TRACE32 trace tool.

The trace data can be obtained by the application software itself or by another tool or by TRACE32 in a previous debug session in which the processing could not be performed for some reasons.

Trace data successfully obtained and analyzed by TRACE32 can be stored by **<trace>.SAVE** and re-viewed by using the **<trace>.LOAD** command. This is the more convenient way because **<trace>.SAVE** stores a lot of additional information used for the analysis. **LA.IMPORT** imports only the trace raw data. For proper processing you need to inform the debugger about all the trace-relevant circumstances.

All kind of trace postprocessing is only possible with the trace method 'LA' (Logic Analyzer). Therefore you need to use **LA.IMPORT** and ‘LA.’ command group for all analysis commands or maybe better switch the trace method to ‘LA’ (**Trace.METHOD LA**) and use the command group ‘Trace.’ for all further operations.

**LA.IMPORT** supports different kinds of trace data and formats. Therefore different commands are provided. For command descriptions, see **LA.IMPORT.ETB, LA.IMPORT.flow, LA.IMPORT.TraceFile, LA.IMPORT.TracePort, LA.IMPORT.VCD**, below.

Most trace data is stored in the file in the timely order the data had been generated.

An exception is the on-chip trace buffer, which is typically used as a circular ring buffer overwriting the trace data all the time until the point of interest is reached.
If this buffer is saved into a file, you need to know the wrap pointer for being able to get the data in a timely order. `LA.IMPORT.WRAP` and `LA.IMPORT.GUESSWRAP` will deal with this concern.

### Process:

**Circular Ring Buffer**
- Save (e.g. by external tool)
- Import (`LA.IMPORT.ETB`)
- Re-order (`LA.IMPORT.WRAP`)
- Display (`LA.List`)

### Result:

#### Records in trace file
- 0
- 1
- ...
- 996 (newest record)

#### Records in LA.List window
- 3 (oldest record)
- 2
- 1

#### Wrap pointer
- 997 (oldest record)
- 998
- 999

For post processing trace data loaded by `LA.IMPORT` you need to take the following steps:

1. Start TRACE32 to run as simulator (config.t32 -> `PBI=SIM`). You neither need a debugger hardware nor a target. You can run TRACE32 as debugger as well, but for the postprocessing this is not needed.

2. Adjust all trace relevant settings like for a real target by running the start-up script you used for generating the trace data. For postprocessing an ETMv4 even further setups might be needed which normally the debugger would read out from the ETM module (ETM.COND, ETM.INSTPO, ETM.QE).

   If the start-up script is **not** available, then try this:
   - At best selecting the chip you are debugging (`SYStem.CPU ...`) is sufficient.
   - For trace data coming from a ARM CoreSight system, all commands describing the trace system on the chip are required (`SYStem.CONFIG ...`).
   - Further all settings for the trace sources done at recording time are needed (e.g. ETM. ...).

3. Load your target application (`Data.LOAD ...`).
4. Import the trace raw data (`LA.IMPORT ...`).
5. Now you can use all trace display and analysis functions, e.g.

   ```
   LA.List TP TPC TPINFO DEFault List.NoDummy.OFF ; with diagnostics
   ```

©1989-2020 Lauterbach GmbH
**Example:**

```
SYStem.CPU CortexA15

SYStem.CONFIG COREDEBUG.Base 0x82010000
SYStem.CONFIG.ETM.Base 0x8201c000
SYStem.CONFIG.FUNNEL.Base 0x80040000
SYStem.CONFIG.FUNNEL.ATBSrc ETM 0
SYStem.CONFIG.ETB.Base 0x80010000

ETM.PortMode.Wrapped
ETM.TraceID 0x55

SYStem.Up

Data.LOAD.Elf myfile.elf

Trace.METHOD.LA
Trace.IMPORT.ETB mydata.bin
Trace.IMPORT.GUESSWRAP

Trace.List TP TPC TPINFO DEFAULT List.NoDummy.OFF
```

**See also**

- `<trace>.IMPORT.CoreByteStream`
- `<trace>.IMPORT.flow`
- `<trace>.IMPORT.STP`
- `<trace>.IMPORT.TraceFile`
- `<trace>.IMPORT.VCD`
- `<trace>.IMPORT.WRAP`
- `<trace>.IMPORT.ETB`
- `<trace>.IMPORT.GUESSWRAP`
- `<trace>.IMPORT.STPByteStream`
- `<trace>.IMPORT.TracePort`
- `<trace>.IMPORT.WRAP`

---

**<trace>.IMPORT.CoreByteStream**

Import pure single core trace data

**Format:**

```
LA.IMPORT.CoreByteStream <file>
```

Imports pure single core trace data (for x86 IPT traces).

**See also**

- `<trace>.IMPORT`
**<trace>.IMPORT.ETB**

**Import on-chip trace data**

<table>
<thead>
<tr>
<th>Format:</th>
<th>LA.IMPORT.ETB &lt;file&gt;</th>
</tr>
</thead>
</table>

Imports a pure binary trace data file obtained from an on-chip trace buffer like ARM CoreSight ETB, ETF, ETR.

You additionally need to use LA.IMPORT.WRAP or LA.IMPORT.GUESSWRAP if the following conditions apply:

- The on-chip trace buffer was used as a circular ring buffer.
- The on-chip trace data was stored as is, it was not read out in the timely order starting from the write pointer position.

LA.Mode FlowTrace will automatically be set when using this command.

---

**See also**

- <trace>.IMPORT
- ‘Release Information’ in ‘Release History’

---

**<trace>.IMPORT.flow**

**Import bus trace data**

<table>
<thead>
<tr>
<th>Format:</th>
<th>LA.IMPORT.flow &lt;file&gt;</th>
</tr>
</thead>
</table>

Re-imports a file that has been exported with <trace>.EXPORT.flow. This bus trace data comes from capturing the fetched instructions and data accesses done on an external bus to figure out the program behavior. It works only if no cache is used and if the bus accesses can be captured. Nowadays this method is rarely used.

---

**See also**

- <trace>.IMPORT

---
Guess wrap pointer

Format: **LA.IMPORT.GUESSWRAP** [<record_number>]

Reformats external trace data loaded to TRACE32 in a timely order. The external trace data of a circular ring buffer is loaded to TRACE32 using **LA.IMPORT.ETB**. The command **LA.IMPORT.GUESSWRAP** scans the loaded trace data and guesses where the wrap pointer might have been.

Optionally, you can pass a record number where the search for the wrap pointer shall start. Without a parameter it starts from the beginning.

Use **LA.IMPORT.WRAP** if you know where the wrap pointer is.

See the figures in the introduction to **<trace>.IMPORT**.

See also

- **<trace>.IMPORT**
- **<trace>.IMPORT.WRAP**

Import STP recording from file (nibble)

Format: **LA.IMPORT.STP** <file>

Imports an STP trace from <file> to process it within TRACE32. One trace record is generated per nibble.

In order to unwrap the trace information for processing, TRACE32 needs to know the following information:
- STM base address and the STP protocol version.

If TRACE32 is aware of the chip characteristic, setting up the chip is sufficient.

Example:

```
SYstem.CPU OMAP4430APPl
LA.IMPORT.STP my_recording.stp
STMLA.List
```
Otherwise the following setup has to be done.

```plaintext
SYStem.CONFIG.STM.Base DAP:0xd4161000 ; any base address != 0x0 is fine
SYStem.CONFIG.STM.Mode STPv2 ; specify the STP protocol version
LA.IMPORT.STP my_recording.stp
STMLA.List
```

See also
- `<trace>.IMPORT`

### `<trace>.IMPORT.STPByteStream`

**Import STP recording from file (byte)**

Format:

```
LA.IMPORT.STPByteStream <file>
```

Same as **LA.IMPORT.STP**, but one trace record is generated per byte.

See also
- `<trace>.IMPORT`
- 'Release Information' in 'Release History'

### `<trace>.IMPORT.TraceFile`

**Import trace data where processing has failed**

Format:

```
LA.IMPORT.TraceFile <file>
```

Re-imports trace data stored by `<trace>.SAVE` for re-processing. This is useful if processing was not possible when the trace recording was made. For example if you had no access to the target code at that moment.

Only the trace raw data will be extracted from the saved (*.ad) file.

**LA.Mode FlowTrace** will automatically be set when using this command.

See also
- `<trace>.IMPORT`
- 'Release Information' in 'Release History'
**Import off-chip trace data**

**<trace>.IMPORT.TracePort**

Imports a pure binary trace data file from an external trace port like an ARM CoreSight TPIU. Unlike on-chip trace data, off-chip trace data includes synchronization packages and depend on the port size of the trace port.

**LA.Mode FlowTrace** will automatically be set when using this command.

---

**See also**
- <trace>.IMPORT
- 'Release Information' in 'Release History'

---

**<trace>.IMPORT.VCD**

Imports a VCD (Value Change Dump) file, which is an industrial standard format for waveforms (not for program trace). It is used for visualizing and analyzing the captured signals in the <trace>.Timing window.

---

**See also**
- <trace>.IMPORT
**Format:**  
```
LA.IMPORT.WRAP <record_number>
```

Reformats external trace data loaded to TRACE32 in a timely order. The external trace data of a circular ring buffer is loaded to TRACE32 using `LA.IMPORT.ETB`.

You pass the `<record_number>` of the first trace record in time (wrap pointer). This is the write pointer location of a circular ring buffer the moment the data has been stored.

**NOTE:** On a CoreSight trace, the write pointer points to a 32-bit value. You need to multiply this value by 4 because each CoreSight trace record is 8 bit in size.

Use `LA.IMPORT.GUESSWRAP` if you do not know where the wrap pointer is.

See the figures in the introduction to `<trace>.IMPORT`.

See also
- `<trace>.IMPORT`
- `<trace>.IMPORT.GUESSWRAP`
Show the state of the input channel.

```
B::fdx.INCHANNEL

<table>
<thead>
<tr>
<th>state</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISableChannel</td>
<td></td>
</tr>
<tr>
<td>ENableChannel</td>
<td></td>
</tr>
</tbody>
</table>
```

```
hostfifo used done
```

### <trace>.InChannel

**Format:** `<trace>.InChannel`

…

### <trace>.Init

**Format:** `<trace>.Init`

The contents of the trace memory/streaming file is erased. All user setups, like the trace mode or trace memory size, remain unchanged.

If the chip includes an onchip trigger unit, counters and trigger levels are cleared. The detailed behavior strongly depends on the onchip trigger unit.

The trace is in OFF state, after a `Trace.Init` was executed.

**TRACE32-ICE and TRACE32-FIRE**

The contents of the trace memory is erased. The trigger unit is returned to its initial settings (the counters, the flags and the trigger levels are cleared). All user setups, like the trace mode or trace memory size, remain unchanged.

**See also**

- `CIProbe`
- `<trace>.Arm`
- `<trace>.AutoInit`
- `<trace>.state`
- `Trace`
- "Emulator Functions’ in ‘FIRE User’s Guide’"
- "Release Information’ in ‘Release History’"
### Sampling configuration for probes JKLMO

**PowerIntegrator only**

<table>
<thead>
<tr>
<th>Format:</th>
<th><strong>Integrator.JKLMO</strong> <em>&lt;option&gt;</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;option&gt;:</strong></td>
<td><strong>250MHZ</strong></td>
</tr>
<tr>
<td></td>
<td><strong>State</strong></td>
</tr>
<tr>
<td></td>
<td><strong>StatePLL</strong></td>
</tr>
<tr>
<td></td>
<td><strong>CLKJ</strong></td>
</tr>
<tr>
<td></td>
<td><strong>CLKK</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Falling</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Rising</strong></td>
</tr>
<tr>
<td></td>
<td><strong>SAMPLE</strong></td>
</tr>
</tbody>
</table>

**250MHZ**
Timing Mode 250 MHz.

**State**
State Mode, clocked by CLKJ or CLKK.

**StatePLL**
State PLL Mode, clocked by CLKJ or CLKK.

**CLKJ**
Clock J select for State-Mode or State-PLL-Mode.

**CLKK**
Clock K select for State-Mode or State-PLL-Mode.

**Falling**
sampling on falling edge of selected clock CLKJ or CLKK.

**Rising**
sampling on rising edge of selected clock CLKJ or CLKK.

**SAMPLE**
sampling delay of selected clock CLKJ or CLKK (-3 … +6 ns in steps of 250 ps), State-PLL-Mode only.

### <trace>.JOINFILE

**Concatenate several trace recordings**

<table>
<thead>
<tr>
<th>Format:</th>
<th><strong>&lt;trace&gt;.JOINFILE</strong> <em>&lt;file&gt;</em> [ <em>&lt;records&gt;</em> ] [ <em>&lt;option&gt;</em> ]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;records&gt;:</strong></td>
<td><strong>&lt;string&gt;</strong></td>
</tr>
<tr>
<td><strong>&lt;option&gt;:</strong></td>
<td><strong>ZIP</strong></td>
</tr>
</tbody>
</table>

Concatenates several trace recordings to increase the volume of trace information to be analyzed.
The reference point is automatically set to the start of the last added trace recording.

Time gaps between the trace recording result in a large Time.Back time (see screenshot above). The option **TIMEGAP <time>** allows a seamless concatenation with regards to the timestamp.

```plaintext
Trace.SAVE my_joinfile.ad ; save current trace contents to file
Trace.FILE my_joinfile.ad ; load trace contents from file
... ; run program to fill the trace
Trace.JOINFILE my_joinfile /TIMEGAP 0.1us ; append current trace contents to loaded trace contents
... ; run program to fill the trace
Trace.JOINFILE my_joinfile /TIMEGAP 0.1us ; append current trace contents
Trace.Chart..Symbol /FILE ; display timing for concatenated trace
...
Trace.SAVE ; close loaded trace file
```

```plaintext
; use record numbers to specify the trace recording to be added
Trace.JOINFILE my_joinfile (4665.)--(5168.)

; use bookmarks to specify record range
Trace.JOINFILE my_joinfile "start" "end" /TIMEGAP 0.1us
```
-only

Select trigger level manually

Format:  

<trace>.LEVEL

Only valid for Ha120 and SA120.

Selects the level of the analyzer trigger unit manually.
Opens a window showing the recorded trace data starting at the record `<record>` or for a range of trace records `<record_range>` (e.g. \((-10000.)\)--\((-2000.)\)).

The columns of the `<trace>.List` window can be defined using the `<items>`. The order of the columns in the window is according to the order of the `<item>` parameters given (with a few exceptions like the `run` column that always appears at the very left).
Note that the default columns are hidden, when you manually specify the columns you want to display. The default columns can be included again in the user-defined column display using the option DEFault. Example:

```
Trace.List List.address DEFault
```

For details on the available columns, see further down.

For trace modes other than RTS, the trace contents can only be displayed if the trace is in OFF or break state. Please refer to the `<trace>.state` command for more information.

For target-specific information and options see:

- Processor Architecture Manuals
- Target Guides FIRE
- Target Guides ICE

Description of Buttons in the `<trace>.List` Window

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Open a <code>&lt;trace&gt;.state</code> window, to configure the trace.</td>
</tr>
<tr>
<td>Goto</td>
<td>Open a <code>&lt;trace&gt;.GOTO</code> dialog box, to move the cursor to a specific record.</td>
</tr>
<tr>
<td>Find</td>
<td>Open a <code>&lt;trace&gt;.Find</code> dialog box, to search for specific entries in the trace.</td>
</tr>
<tr>
<td>Chart</td>
<td>Display the program execution time at different symbols as a time chart. See the <code>&lt;trace&gt;.Chart.sYmbol</code> command.</td>
</tr>
<tr>
<td>Profile</td>
<td>Open a <code>&lt;trace&gt;.PROfileChart.sYmbol</code> window.</td>
</tr>
<tr>
<td>MIPS</td>
<td>Open a MIPS.PROfileChart.sYmbol window.</td>
</tr>
<tr>
<td>More/Less</td>
<td>Switch step-by-step from full display (all CPU cycles including dummies) to HLL display and vise versa.</td>
</tr>
</tbody>
</table>

If no parameters are specified, a predefined set of items will appear in the window. By selecting items, specific items can be displayed in any order defined by the user. It is possible to remove a selection from the list by appending the keyword OFF. The display format of the entries can be changed by the `%<format>` options.
In the case of an SMP system, the following options are provided:

<table>
<thead>
<tr>
<th><strong>Option</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE</strong></td>
<td>Displays trace memory contents loaded with <code>Trace.FILE</code>.</td>
</tr>
<tr>
<td><strong>FlowTrace</strong></td>
<td>The trace works as a program flow trace. This option is usually not required.</td>
</tr>
<tr>
<td><strong>BusTrace</strong></td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td><strong>Track</strong></td>
<td>Track the <code>&lt;trace&gt;.List</code> window with other trace list windows (tracking to record number or time possible).</td>
</tr>
<tr>
<td><strong>Mark <code>&lt;item&gt;</code></strong></td>
<td>Bold print all cycles on a yellow background which contain the specified item.</td>
</tr>
<tr>
<td><strong>NorthWestGravity</strong></td>
<td>With <strong>NorthWestGravity</strong>: The record numbering in the top left corner stays fixed as you resize the <code>&lt;trace&gt;.List</code> window.</td>
</tr>
<tr>
<td></td>
<td>Without <strong>NorthWestGravity</strong>: The record numbering scrolls as you resize the window.</td>
</tr>
<tr>
<td><strong>Raw</strong></td>
<td>Displays all channels as raw hexadecimal values (where applicable)</td>
</tr>
<tr>
<td><strong>TimeZero</strong></td>
<td>Use timestamp of first entry in listing as global reference (item <code>Time.Zero</code>).</td>
</tr>
</tbody>
</table>

```plaintext
Trace.FILE test1 ; load trace file
Trace.List /File ; display trace listing, source for the ; trace data is the loaded file
Trace.List /Mark Address sieve ; mark all trace lines which contain the ; address sieve
```

In the case of an SMP system, the following options are provided:

<table>
<thead>
<tr>
<th><strong>Option</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SplitCORE</strong></td>
<td>Displays the trace recording of all cores side by side.</td>
</tr>
<tr>
<td><strong>CORE <code>&lt;number&gt;</code></strong></td>
<td>Displays the trace recording of the specified core.</td>
</tr>
</tbody>
</table>
### Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ascii</td>
<td>Displays single bytes as ASCII characters</td>
</tr>
<tr>
<td>BINary</td>
<td>Displays single bytes in binary values</td>
</tr>
<tr>
<td>Decimal</td>
<td>Displays single bytes in decimal values</td>
</tr>
<tr>
<td>Hex</td>
<td>Displays single bytes in hex values</td>
</tr>
<tr>
<td>HighLow</td>
<td>Displays single bits as 'H' or 'L' character</td>
</tr>
<tr>
<td>LEN &lt;size&gt;</td>
<td>Specifies the width of non numeric fields (e.g. symbols)</td>
</tr>
<tr>
<td>Signed</td>
<td>Displays single bytes signed</td>
</tr>
<tr>
<td>TimeAuto</td>
<td>Displays time values in a floating display format (short)</td>
</tr>
<tr>
<td>TimeFixed</td>
<td>Displays time values in a fixed point format (long format)</td>
</tr>
<tr>
<td>Timing</td>
<td>Displays single bits as vertical timing</td>
</tr>
<tr>
<td>Unsigned</td>
<td>Displays single bytes unsigned</td>
</tr>
</tbody>
</table>

### Examples:

```plaintext
; display trace listing, limit the symbol names to 20 characters
Trace.List Address CYcle Data.L %LEN 20. sYmbol TIme.Back

; display trace listing, show the external trigger input 0 as vertical timing
Trace.List %TIMING T.0 DEFault
```

### The following <items> define the columns shown in the <trace>.List windows

<table>
<thead>
<tr>
<th>&lt;items&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFault</td>
<td>Default trace display.</td>
</tr>
<tr>
<td></td>
<td>The default trace display can be configured with the command <strong>SETUP.ALIST</strong>.</td>
</tr>
<tr>
<td>ALL</td>
<td>Select all available channels (superset of DEFault)</td>
</tr>
<tr>
<td>CPU</td>
<td>Set of channels describing the CPU state (similar to the original setting of DEFault but no source code display).</td>
</tr>
<tr>
<td>LINE</td>
<td>Set of channels which contains all CPU control lines.</td>
</tr>
</tbody>
</table>
### Run

Gives various information about the execution of the current record.

- **GO**: the first instruction that was executed by the CPU after starting program execution with **Go**. 
  (TRACE32-ICE only, sampling of the first instruction can be switched off by **Analyzer.Mode PREPOST OFF**)
- **BRK**: Indicates that the program execution was stopped.
- **T**: Indicates a trigger event.
- **f**: Foreground program
- **b**: Background program
- **ft**: Trigger event occurred in the foreground program
- **bt**: Trigger event occurred in the background program
- **0,1,2,3 ...**: in SMP systems, the run column indicates the number of the core that executed the given code; additionally, the background color of the records changes to highlight the relevant core (light red, light green, ...).

### Address

Start address of each displayed **block of executed opcodes**; for displaying the address of each single opcode, use the channel **List.Address**.

### sYmbol

Symbolic address with path and offset
(as find item will search on all processor busses)

### sYmbolN

Symbolic address without path but with offset

### sYmbollnline

Inline symbol name with path.

### sYmbollnlineN

Inline symbol name without path.

### AAddress

Physical (absolute) CPU address

### AAddress.0--31

Physical address bits A0..A31

### PAddress

This column displays the address of the instruction that was executed before a read or write access was performed (**B**).

Prestore address (**E**), refer to **Analyzer.Mode Prestore** for more information.
Poststore address (**F**), refer to **Analyzer.Mode Poststore** for more information.
<p>| <strong>PsYmbol</strong> | This column display the address of the instruction that was executed before a read or write access was performed (B). Symbolic prestore address with path and offset (E), symbolic poststore address with path and offset (F). |
| <strong>FAddress</strong> | Flowtrace execution address (when flowtrace available) |
| <strong>FsYmbol</strong> | Symbolic flowtrace execution address |
| <strong>BAddress</strong> | Bus address, same like physical address, but also displayed when the bus is not transferring data |
| <strong>Var</strong> | Symbolic display of data accesses to HLL variables |
| <strong>CYcle</strong> | Bus cycle |
| <strong>Data</strong> | CPU data full width |
| <strong>Data.B</strong> | CPU data single byte |
| <strong>Data.B0</strong> | CPU data lower byte |
| <strong>Data.W0</strong> | CPU data lower word |
| <strong>Data.T0</strong> | CPU data lower triple |
| <strong>Data.0..31</strong> | CPU data bit 0 to 31 |
| <strong>Data.0--7</strong> | CPU data bits 0 to 7 as single bits (8 bit processor) |
| <strong>Data.0--15</strong> | CPU data bits 0 to 15 as single bits (16 bit processor) |
| <strong>Data.0--31</strong> | CPU data bits 0 to 31 as single bits (32 bit processor) |
| <strong>Data.sYmbol</strong> | Display the data value symbolically |
| <strong>BData</strong> | Like Data, but always displays the data even when the bus is idle |
| <strong>List.Address</strong> | Lists the address for each individual opcode (instead of the start address of blocks of executed opcodes) |
| <strong>List.Asm</strong> | Disassembled mnemonics |
| <strong>List.Mix</strong> | Disassembled mnemonics and HLL source |
| <strong>List.Hll</strong> | HLL source only, dequeuing based on disassembler |</p>
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List.HllOnly</td>
<td>HLL source only no dequeueing</td>
</tr>
<tr>
<td>List.NoFetch</td>
<td>Suppresses the display of op-fetches</td>
</tr>
<tr>
<td>List.NoPFetch</td>
<td>Suppresses the display of prefetch cycles</td>
</tr>
<tr>
<td>List.NoCycle</td>
<td>Suppresses the display of more than one cycle between lines</td>
</tr>
<tr>
<td>List.Label</td>
<td>Label of disassembled mnemonic</td>
</tr>
<tr>
<td>List.Comment</td>
<td>Comments to disassembled mnemonics</td>
</tr>
<tr>
<td>List.Queue</td>
<td>Start address of disassembled mnemonic</td>
</tr>
<tr>
<td>List.TASK</td>
<td>Displays OS Awareness information (system-calls etc.)</td>
</tr>
<tr>
<td>List.Reorder</td>
<td>Reorders bus cycles logically (only some processors)</td>
</tr>
<tr>
<td>List.NoDummy</td>
<td>Suppresses the display of dummy cycles (where applicable)</td>
</tr>
<tr>
<td>List.Bondout</td>
<td>Display internal bondout information (where applicable)</td>
</tr>
<tr>
<td>List.TTime</td>
<td>Display time information in assembler or HLL lines</td>
</tr>
<tr>
<td>List.CTS</td>
<td>Display CTS information (Context Tracking System)</td>
</tr>
<tr>
<td>List.SOURCE-FILE</td>
<td>Display source file name for each line</td>
</tr>
<tr>
<td>TTime</td>
<td>Time marker (default TTime.Fore)</td>
</tr>
<tr>
<td>TTime.Fore</td>
<td>Time marker, relative time to next record</td>
</tr>
<tr>
<td>TTime.Back</td>
<td>Time marker, relative time to previous record</td>
</tr>
<tr>
<td>TTime.Zero</td>
<td>Time marker, relative to global reference</td>
</tr>
<tr>
<td>TTime.REF</td>
<td>Time marker, relative to reference point</td>
</tr>
<tr>
<td>TTime.Trigger</td>
<td>Time marker, relative to trigger point</td>
</tr>
<tr>
<td>TTime.FUNC</td>
<td>Time spent in a function (*1)</td>
</tr>
<tr>
<td>TTime.FUNCEX</td>
<td>Time spent in calls (*1)</td>
</tr>
<tr>
<td>TTime.FUNCIN</td>
<td>Time spent in code of function (*1)</td>
</tr>
<tr>
<td>TTime.MARKAB</td>
<td>Time relative back to the last marker A</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>Time.MARKAF</td>
<td>Time relative forward to the next marker A</td>
</tr>
<tr>
<td>Time.MARKBB</td>
<td>Time relative back to the last marker B</td>
</tr>
<tr>
<td>Time.MARKBF</td>
<td>Time relative forward to the next marker B</td>
</tr>
<tr>
<td>Time.MARKCB</td>
<td>Time relative back to the last marker C</td>
</tr>
<tr>
<td>Time.MARKCF</td>
<td>Time relative forward to the next marker C</td>
</tr>
<tr>
<td>Time.MARKDB</td>
<td>Time relative back to the last marker D</td>
</tr>
<tr>
<td>Time.MARKDF</td>
<td>Time relative forward to the next marker D</td>
</tr>
<tr>
<td>CLOCKS.Back</td>
<td>Number of clocks relative time to previous record</td>
</tr>
<tr>
<td>CLOCKS.Fore</td>
<td>Number of clocks relative time to next record</td>
</tr>
<tr>
<td>CLOCKS.Trigger</td>
<td>Number of clocks relative to trigger point</td>
</tr>
<tr>
<td>CLOCKS.REF</td>
<td>Number of clocks relative to reference point</td>
</tr>
<tr>
<td>CLOCKS.Zero</td>
<td>Number of clocks relative to global zero point</td>
</tr>
<tr>
<td>FUNC</td>
<td>Function nesting display (*1)</td>
</tr>
<tr>
<td>FUNCVar</td>
<td>Function nesting plus variables</td>
</tr>
<tr>
<td>FUNCNR</td>
<td>Record number associated with this entry/exit point (*1)</td>
</tr>
<tr>
<td>IGNORE</td>
<td>Record ignored or used for performance/nesting analysis</td>
</tr>
<tr>
<td>LeVel</td>
<td>Trigger unit logical level</td>
</tr>
<tr>
<td>MARK.all</td>
<td>Display markers</td>
</tr>
<tr>
<td>MARK.A</td>
<td>Display marker A</td>
</tr>
<tr>
<td>FLAG.all</td>
<td>Flags of the trigger unit in a short form</td>
</tr>
<tr>
<td>FLAG.0</td>
<td>Flag 0 of the trigger unit</td>
</tr>
<tr>
<td>Trigger.A</td>
<td>External trigger input A (E)</td>
</tr>
<tr>
<td></td>
<td>External trigger input EXTA (bit 0) and EXTB (bit 1) (F)</td>
</tr>
</tbody>
</table>
(1): The trace must be the same as for the command `<trace>.STATistic.Func`. The combination of the `FUNC` keyword with the `List.TASK` keyword makes the function nesting display task sensitive.

### FLOW ERROR Diagnosis

![FLOW ERROR Diagnosis](image)

**FLOWERROR**  
Display flow error column

### Flow Trace Decoding

![Flow Trace Decoding](image)

**FAddress**  
To decompress the recorded trace information the program code starting at FAddress is read.

**FsYmbol**  
Symbolic address of FAddress.
### Trace Raw Data and Packet Decoding

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCOUNT</td>
<td>To decompress the recorded trace information FCOUNT number of byte is read.</td>
</tr>
<tr>
<td>FLen</td>
<td>(deprecated)</td>
</tr>
</tbody>
</table>

**TP**

*All raw trace data* as recorded at the trace port.
For multicore systems the stream of trace data may contain information for *multiple cores* (e.g. in “wrapped mode” for CoreSight systems).

**TPC**

Raw trace data pertaining to a *single core*.
For multicore systems this data is extracted from the overall trace data stream.

**TPINFO**

Information obtained by decoding a single trace packet (which may consist of multiple bytes).

**TSINFO**

Timestamp calculation background information, required for the diagnosis of complex timing scenarios.
Context ID/Ownership Trace Packet Decoding

<table>
<thead>
<tr>
<th>task</th>
<th>If a Context ID or ownership packet is decoded and if it is assignable to a task, the “task” cycle type and the task name is displayed. The displayed data value is a TRACE32 internal value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceid</td>
<td>If a Context ID or ownership packet is decoded and if it can not be assigned to a task or any other protocol-specific content such as service, intr etc. the cycle type “traceid” and the packet content is displayed.</td>
</tr>
</tbody>
</table>

If the machine ID is encoded, the machine name is also displayed ("sender" in the screenshot below).

See also
- <trace>.BookMark
- <trace>.Timing
- Task
  - Analyzer.RECORD.ADDRESS()
  - Analyzer.RECORD.OFFSET()
  - Analyzer.RECORD.DATA()
  - Analyzer.REF()
- <trace>.state
- <trace>.View
- CIProbe
  - Analyzer.RECORD.DATA()
  - Analyzer.RECORDS()
▲ ‘How to use the PROTOanalyzer’ in ‘DigRF Protocol Analyzer’
▲ ‘How to use the PROTOanalyzer’ in ‘FlexRay Protocol Analyzer’
▲ ‘How to use the PROTOanalyzer’ in ‘LIN Bus Protocol Analyzer’
▲ ‘Release Information’ in ‘Release History’
▲ ‘Displaying the Trace’ in ‘Training FIRE Analyzer’
▲ ‘Emulator Functions’ in ‘FIRE User’s Guide’
The command **Trace.ListNesting** is mainly used to investigate issues in the construction of the call tree for the nesting function run-time analysis. Typical commands for the nesting function run-time analysis are the commands **Trace.STATistic.Func** or **Trace.STATistic.TREE**.

<table>
<thead>
<tr>
<th>&lt;option&gt;</th>
<th>For a description of the generic options, see &lt;trace&gt;.List.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE &lt;number&gt;</td>
<td>Filters the <strong>Trace.ListNesting</strong> window by the specified core. Processing is done for all cores, but only the specified core is displayed. All other cores are temporarily hidden in the window.</td>
</tr>
<tr>
<td>SplitCORE</td>
<td>Displays the trace recording of the cores side by side in the <strong>Trace.ListNesting</strong> window.</td>
</tr>
</tbody>
</table>

The **Trace.ListNesting** window provides the nesting details. If a function entry point is selected, the path to the function exit is highlighted.

If the function exit is located far apart, you can use the Down Arrow (v) to jump to the function exit.
The interrupt nesting is marked specially (see screenshot below).

Code optimizations are the main reason for issues in the construction of the call tree. TRACE32 indicates these issues as PROBLEMs or WORKAROUNDS.

**PROBLEMs**

- A PROBLEM is a point in the trace recording that TRACE32 cannot integrate into the current nesting.
- PROBLEMs are marked with (!) in the Trace.ListNesting window. The name of the expected function is shown.
- PROBLEMs are ignored in the construction of the call tree.
- PROBLEMs may affect the construction of the call tree, so it is important to inspect them. The Statistic Markers can be used to solve a PROBLEM.

To inspect a PROBLEM, proceed as follows:

1. Go to the start of the trace recording.
2. Use the Find... command from the Edit menu. Type (!) as find item.
3. Open a Trace Listing to inspect the problem in detail.
WORKAROUND

- A WORKAROUND is a point in the trace recording that TRACE32 can not integrate into the current nesting.
- TRACE32 attempts to integrate this point into the function nesting, by deriving information from previous scenarios in the nesting.
- WORKAROUNDS are marked with (?) in the Trace.ListNesting window.
- WORKAROUNDS may affect the construction of the call tree, if the derived information is wrong. It is recommended to inspect the WORKAROUNDS.

To inspect a WORKAROUND, proceed as follows:

1. Go to the start of the trace recording.
2. Use the Find… command from the Edit menu. Type (?) as find item.
3. Open a Trace Listing to inspect the problem in detail.

See also

- <trace>.state
- Trace
- "Release Information’ in ‘Release History’
<trace>.ListVar

Displays a list of all variable recorded if it is used without parameters.

The option Mark allows to mark the specified variable access.

; mark trace entry when a 0x0 is written to variable mstatic1
Trace.ListVar /Mark Address Var.RANGE(mstatic1) CYcle Write Data 0x0
Format 1 represents the standard syntax, in which the variable names follow the `%<format>` parameter. The following options provide a representation in which variable values can be better compared:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Split</strong></td>
<td>Each specified variable gets its own column. If a variable is accessed its value is displayed. Write accesses are printed in black, read accesses are printed in gray.</td>
</tr>
<tr>
<td><strong>SplitFill</strong></td>
<td>Each specified variable gets its own column. Whenever one of the specified variables is displayed, the current values of all other specified variables are displayed as well. Write accesses are printed in black, everything else is printed in gray.</td>
</tr>
</tbody>
</table>

Examples for Format 1:

```plaintext
//Display all accesses to the variable mstatic1
Trace.ListVar %DEFAULT mstatic1

//Display all accesses to the listed variables
Trace.ListVar %DEFAULT mstatic1 fstatic fstatic2

//Display all accesses to the listed variables, but display the values of each variable in a separate column
Trace.ListVar %DEFAULT mstatic1 fstatic fstatic2 /Split

//Display all accesses to the listed variables, but display the values of each variable in a separate column fill in the current value of the not accessed variable to each line
Trace.ListVar %DEFAULT mstatic1 fstatic vlong /SplitFill
```
Format 2 represents the advanced syntax, here it is possible to restrict the display to the specified 
<record_range> or <time_range>.

Examples for Format 2:

```
Trace.ListVar (-14874903.)--(-14874761.)
Trace.ListVar 1.8s--10.8s
Trace.ListVar (-14874903.)--(-14874761.) vfloat
Trace.ListVar 1.8s--10.8s mstatic1 fstatic fstatic2 /SplitFill
```

See also

- `<trace>.state`
- `Trace`

▲ 'Release Information' in 'Release History'
Loads trace data from a file into the debugger. Typically `<trace>.LOAD` is used to analyze data in a simulator or to compare different recordings.

The command loads the data into the “normal” trace buffer i.e. the same buffer that is filled when recording data using an analyzer (e.g. via PowerTrace, PowerProbe, PowerIntegrator etc.). As the standard trace commands work on this buffer, they automatically work on the loaded data. To highlight that loaded data is displayed, windows are marked by a red label `LOAD` label in the bottom-left corner.

To save trace data, use the command `<trace>.SAVE`.

| `<file>` | The default extension for the file name is `*.ad`. |

**NOTE:** There is a similar but slightly different command `<trace>.FILE`. It loads the trace data into a dedicated `file trace buffer`. To have trace commands (e.g. `Trace.List`) work on the `file trace buffer`, they need to be invoked with the parameter `/FILE`.

An example for working on loaded trace data:

```
Trace.LOAD test4 ; load trace contents from file
Data.LOAD.Elf demo.elf /NoCODE ; load symbol information for the ; post-processing
Trace.List ; display loaded trace contents
Trace.Chart.Symbol ; symbol analysis of trace
Trace.STATistic.Func ; function run-time analysis
```
The TRACE display and analysis commands are re-directed to the selected trace method if:

- **Trace.LOAD** is executed without the parameter `<file>`.

```plaintext
Trace.LOAD ; Re-direct trace display and analysis commands to the selected trace method
```

- A trace configuration command is executed.

```plaintext
Trace.Init ; the trace configuration command Trace.Init re-directs the trace display and analysis commands to the selected trace method
```

- The program execution is started while **Trace.AutoArm** is set to ON.

If the **Trace.METHOD Probe** or **Trace.METHOD Integrator** was selected, when the trace contents were saved, the option `/Config` can be used to re-activate the **Probe/Integrator** and **NAME** settings.

```plaintext
Trace.METHOD Probe ; select the trace method Probe for the PowerProbe
...
Trace.SAVE probetest1 ; save the trace contents to the file probetest1
...
QUIT ; end TRACE32
```

```plaintext
Trace.LOAD probetest1 /Config ; load the trace contents from the file probetest1
Trace.List ; use a TRACE32 instruction set simulator to postprocess the PowerProbe trace data
; load the Probe settings and NAMEs
```

**See also**
- `<trace>.FILE`
- `<trace>.SAVE`
- `<trace>.state`
- Trace
- 'Emulator Functions’ in ‘FIRE User’s Guide’
- 'Release Information’ in 'Release History’

©1989-2020 Lauterbach GmbH

General Commands Reference Guide T 254
Combine two trace files into one. This is useful for traces recorded for different cores working in AMP mode.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZIP</td>
<td>File is compressed with the gzip archive format.</td>
</tr>
<tr>
<td>NoCompress</td>
<td>Obsolete.</td>
</tr>
<tr>
<td>QuickCompress</td>
<td>Obsolete.</td>
</tr>
<tr>
<td>Compress</td>
<td>Obsolete.</td>
</tr>
<tr>
<td>TIMEGAP &lt;time&gt;</td>
<td>Allows a seamless concatenation with regards to the timestamp</td>
</tr>
</tbody>
</table>

**NOTE:** The ZIP or Compress options have meanwhile become obsolete because the resulting file is compressed by default.
Selects the trace method you want to use. This allows you to work with a trace method other than the one suggested by TRACE32.

For information about how TRACE32 makes its suggestion, see “What to know about the TRACE32 default settings for Trace.METHOD”, page 111.

<table>
<thead>
<tr>
<th>Trace Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzer</td>
<td>Trace RAM is provided by a TRACE32 tool (e.g. PowerTrace, ICE, FIRE), but not CombiProbe.</td>
</tr>
<tr>
<td>ART</td>
<td>Advanced Register Trace. For more information refer to “ART - Advanced Register Trace” (debugger_user.pdf).</td>
</tr>
<tr>
<td>CAnalyzer</td>
<td>Trace RAM is provided by the CombiProbe.</td>
</tr>
<tr>
<td>FDX</td>
<td>Fast Data eXchange. The target application needs to write the required trace information to a small ring buffer (min. size 2 trace records). The contents of the ring buffer is transferred to the TRACE32 software while the program execution is running and saved there for later display. If the on-chip debug unit provides a Debug Communications Channel (DCC) the required trace information can be transferred directly to the TRACE32 software.</td>
</tr>
<tr>
<td>HAnalyzer</td>
<td>Trace RAM is provided by the host. This method is used for targets that provide a specifically implemented trace channel over interfaces like USB3.</td>
</tr>
<tr>
<td>Integrator</td>
<td>The Lauterbach logic analyzer PowerIntegrator is used to record the trace information.</td>
</tr>
<tr>
<td>Trace Methods</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>IProbe</strong></td>
<td>The Lauterbach IProbe logic analyzer within the Power Trace II is used to record signals.</td>
</tr>
<tr>
<td><strong>LA</strong></td>
<td>LA (Logic Analyzer). Trace information not recorded by TRACE32 can be loaded and processed. This requires that the TRACE32 software is familiar with the format of the trace information.</td>
</tr>
<tr>
<td><strong>LOGGER</strong></td>
<td>The target application can write the required trace information to target RAM. TRACE32 loads the trace information from the target RAM for display and processing.</td>
</tr>
<tr>
<td><strong>Onchip Onchip2</strong></td>
<td>The trace information is saved in the first/second onchip trace buffer provided by the chip.</td>
</tr>
<tr>
<td><strong>Probe</strong></td>
<td>The Lauterbach logic analyzer PowerProbe is used to record the trace information.</td>
</tr>
<tr>
<td><strong>SNOOPer</strong></td>
<td>Snooper. For details, see “Application Note for the SNOOPer Trace” (app_snooper.pdf).</td>
</tr>
<tr>
<td><strong>NONE</strong></td>
<td>A dummy trace method indicating that the trace feature, including the Trace.* commands, is not yet operational. The only command exceptions are Trace.METHOD and Trace.state. Select the trace method you want to use, using either the Trace.METHOD command, the Trace.state window, or a PRACTICE script (*.cmm). For more information including illustrations, see “What to know about the TRACE32 default settings for Trace.METHOD”, page 111.</td>
</tr>
</tbody>
</table>

See also

- `<trace>.state`
- `CAalyzer`
- `IProbe`
- `Probe`
- `Trace.METHOD.ART()`
- `Trace.METHOD.CAnalyzer()`
- `Trace.METHOD.FDX()`
- `Trace.METHOD.IProbe()`
- `Trace.METHOD.Integrator()`
- `Trace.METHOD.LA()`
- `Trace.METHOD.LOGGER()`
- `Trace.METHOD.ONCHIP()`
- `Trace.METHOD.PROBE()`
- `Trace.METHOD.SNOOPer()`
- `ART`
- `Analyzer`
- `HAnalyzer`
- `Integrator`
- `LOGGER`
- `Onchip`
- `SystemTrace`
- `Trace.METHOD.FDX()`
- `Trace.METHOD.LA()`
- `Trace.METHOD.LOGGER()`
- `Trace.METHOD.PROBE()`
- `Trace.METHOD.SNOOPer()`

▲ ‘Trace Functions’ in ‘General Function Reference’
▲ ‘Release Information’ in ‘Release History’
**Format:**  
<trace>.Mode [<mode>]

**<mode>:**  
- **Fifo**  
- **Stack**  
- **Leash**  
- **STREAM**  
- **PIPE**  
- **RTS**  
- **FAST**  
- **Create**  
- **64Bit**  
- **Change**  
- **Compress**  
- **Prestore [ON | OFF] (deprecated) Use ETM.DataTracePrestore instead.**  
- **SLAVE [ON | OFF]**  
- **PrePost [ON | OFF]**  
- **MasterBrk [ON | OFF]**  
- **SlaveBrk [ON | OFF]**  
- **HT [Trace | Prestore | Selective | OFF]**  
- **100MHZ**  
- **DCC**  
- **Memory**  
- **DetailTrace**  
- **EventTrace**  
- **FlowTrace**  
- **LoopTrace**  
- **SPB [ON | OFF]**  
- **RPB [ON | OFF]**

Selects the trace operation mode. The most common operation modes are:

- **Fifo**  
  If the trace is full, new records will overwrite older records. The trace records always the last cycles before the break.

- **Stack**  
  If the trace is full recording will be stopped. The trace always records the first cycles after starting the trace.

- **Leash**  
  Stops the program execution when trace is nearly full.
STREAM

The trace data is immediately conveyed to a file on the host after it was placed into the trace. This procedure extends the size of the trace memory to up to several T Frames.

STREAM mode can only used for the following TRACE32 trace tools:
- PowerTrace Serial
- PowerTrace II
- CombiProbe/CombiProbe 2
- uTrace
- PowerIntegrator (Probe A-E only)
- TRACE32 POWERTRACE/ ETHERNET supports STREAM mode for some trace protocols. If it is not supported, the command Trace.Mode STREAM is blocked.

STREAM mode 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.

The streaming file is placed into the TRACE32 temp directory (OS.PresentTemporaryDirectory()) by default and is named <trace32_instance_id>streama.t32 (OS.ID()). If you explicitly want to specify a location for the streaming file use the command <trace>.STREAMFILE <file>.

Note that the contents of the streaming file are in a proprietary format and not intended for use in external applications. See Trace.EXPORT for details on how to export trace data for external applications.

Please be aware that the streaming file is deleted as soon as you de-select the STREAM mode or when you quit TRACE32.

PIPE

The trace data is immediately conveyed to the host and distributed to user-defined trace sinks. Not supported with PowerTrace Ethernet 256/512MB. See <trace>.PipeWRITE.

RTS

The RTS radio button is only an indicator that shows if Real-time Profiling is enabled. For enabling RTS use the command RTS.ON.

Further operation modes:

100MHz
Port analyzer only

64Bit
LOGGER only

BusTrace
BusTrace mode.

ClockTrace
Clock trace mode.

©1989-2020 Lauterbach GmbH
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>Compressed information transfer for FDX trace.</td>
</tr>
<tr>
<td>Create</td>
<td>Create LOGGER software trace control block by debugger.</td>
</tr>
<tr>
<td>DetailTrace</td>
<td>Address and data of all cycles except for fetches and free cycles are recorded.</td>
</tr>
<tr>
<td>E</td>
<td>Dualport access (E).</td>
</tr>
<tr>
<td>EventTrace</td>
<td>Recording starts after trigger event.</td>
</tr>
<tr>
<td>FlowTrace</td>
<td>FlowTrace mode.</td>
</tr>
<tr>
<td>HT OFF</td>
<td>The Hypertrace unit is turned off.</td>
</tr>
<tr>
<td>HT Prestore</td>
<td>The Hypertrace samples the last 64 (32) bus cycles before a selective trace with the regular trace is made.</td>
</tr>
<tr>
<td>HT Selective</td>
<td>The Hypertrace samples the same records like the regular trace. This mode can be used to make large selective address traces or function nesting analysis.</td>
</tr>
<tr>
<td>HT Trace</td>
<td>The Hypertrace unit samples all cycles (this is the default).</td>
</tr>
<tr>
<td>LATCH</td>
<td>Latch transients.</td>
</tr>
<tr>
<td>LoopTrace</td>
<td>Flow trace inhibiting redundant entries to capture memory.</td>
</tr>
<tr>
<td>MasterBrk</td>
<td>When activated, the break state of the HA120 unit will cause all other traces to enter break state (e.g. TA32).</td>
</tr>
<tr>
<td>PCP</td>
<td>PCP trace.</td>
</tr>
<tr>
<td>Poststore</td>
<td>tbd.</td>
</tr>
<tr>
<td>PostTrace</td>
<td>tbd.</td>
</tr>
<tr>
<td>PrePost</td>
<td>Forces a recording of the opcode fetch cycle at the beginning and at the end of program execution. The recording does not depend on how the trace trigger unit was programmed. The special markers GO and BRK in the list window emphasize these forced recordings.</td>
</tr>
</tbody>
</table>

©1989-2020 Lauterbach GmbH
Prestore
(deprecated)

Samples the last opcode-fetch address before a selective data access. The size of the trace memory is halved automatically. By using this mode is it possible to get information about which program has made an access to a variable without tracing the complete program flow.

RPB
TriCore only

When enabling a bus trace, all read and write accesses on this bus will be traced by default, including data values and addresses. This trace is independent of Onchip.Mode DataTrace which only affects TriCore and PCP. The recorded data will be mixed into the program flow at the point it was placed into the trace memory by the silicon, e.g. wd-rpb marks a write access on remote peripheral bus. There is no support for data cycle assignment as it is for TriCore LMB bus trace.

SLAVE OFF

Separates the trace from the program execution, i.e. trace is recording even when the program execution is stopped (very rarely used command).

SLAVE ON

Ties the trace to the execution of the program, i.e. trace and the filter/trigger work only during user program execution (very rarely used command).

SlaveBrk

When activated, the break of other traces (e.g. TA32) will cause the HA120 unit to enter break state.

NOTE: The break state is not shown in the status window!

SPB
TriCore only

When enabling a bus trace, all read and write accesses on this bus will be traced by default, including data values and addresses. This trace is independent of Onchip.Mode DataTrace which only affects TriCore and PCP. The recorded data will be mixed into the program flow at the point it was placed into the trace memory by the silicon, e.g. rd-spb marks a read access on system peripheral bus. There is no support for data cycle assignment as it is for TriCore LMB bus trace.

See also

- <trace>.state
- Trace
- CIProbe
- COVere.METHOD
- Analyzer.MODE.FLOW()
- ‘On-chip Trace’ in ‘StarCore Debugger and Trace’
- ‘CPU specific Commands’ in ‘TriCore Debugger and Trace’
- ‘HCS12 On-chip Debug Module’ in ‘FIRE Emulator for HC12/MCS12’
- ‘HCS12 On-chip Debug Module’ in ‘FIRE Emulator for HC12/MCS12’
- ‘Release Information’ in ‘Release History’
- ‘CPU specific Trace Commands’ in ‘Simulator for TriCore’
<trace>.MUX

Select channels

Format: <trace>.MUX [A | B | AUTO]

Selects which channels are connected to the port analyzer. This option is only available when the emulation head has more than 64 port analyzer channels. The AUTO selection is only available in port analyzer version 2.

See also
- Trace

<trace>.OFF

Switch off

Format: <trace>.OFF

Disables both trace memory and the trigger unit. The trace memory can be read and the trigger unit be programmed.

See also
- <trace>.state
- CIProbe
- Trace
- Analyzer.STATE()

▲ 'Emulator Functions' in 'FIRE User's Guide'

<trace>.PC

Display PC in real-time

Format: <trace>.PC [ON | OFF]

A certain hardware on the trace allows the displaying of the PC (program counter), while the CPU is running in real time. Without parameter the PC is displayed in the message line. It is displayed in the state line continuously, with the option ON. This command is useful if the program is running in loops (e.g. waiting for input).

See also
- <trace>.state
- Trace
- Analyzer.PC()

▲ 'Emulator Functions' in 'FIRE User's Guide'
### Unload all DLLs

**Format:**

```
<trace>.PipePROTO (deprecated)
Use <trace>.CustomTraceLoad "" instead.
```

### Send command to DLLs

**<trace>.PipePROTO.COMMAND**

**See also**

- <trace>.PipePROTO.COMMAND
- <trace>.PipePROTO.load
- <trace>.state
- Trace

**Format:**

```
<trace>.PipePROTO.COMMAND [cmd_line_args] (deprecated)
Use <trace>.CustomTrace.<label>.COMMAND instead.
```

### Define a user-supplied DLL as trace sink

**<trace>.PipePROTO.load**

**See also**

- <trace>.PipePROTO

**Format:**

```
<trace>.PipePROTO.load <dll_name> [cmd_line_args] (deprecated)
Use <trace>.CustomTraceLoad instead.
```

Loads a user supplied DLL (or “shared object” under Unix) as trace sink. Up to 8 DLLs can be loaded as trace sinks simultaneously. The debugger software will pass each received message to all loaded DLLs. The DLLs can then filter and process the messages in any possible way.
**<trace>.PipeWRITE**

Connect to a named pipe to stream trace data

```
Format:   <trace>.PipeWRITE [<name>]
```

Connect to a named pipe to stream the raw trace data to an external application. If `<name>` is omitted, the debugger disconnects from the named pipe.

**Example:** (for Windows)

```
Trace.Mode PIPE ; switch to PIPE mode
Trace.PipeWRITE \\.\pipe\ptrace ; connect to named pipe
; run test
Go
... ; trace data now streamed to
Break ; external application
Trace.PipeWRITE ; disconnect from named pipe
```

**See also**
- Trace

---

**<trace>.PortFilter**

Specify utilization of trace memory

```
Format:   <trace>.PortFilter ON | OFF | PACK | MAX | AUTO
```

If the trace information is conveyed to the host computer at the recording time, it is advantageous to reduce the amount of data to be conveyed. This goal can be achieved by the following:

- No recording of idle cycles (applies only if the on-chip trace logic generates idle cycles).
- No generation of the TRACE32 tool time stamp, if it is not required for the intended analysis.

The command `Trace.PortFilter` allows the following configurations:

<table>
<thead>
<tr>
<th>ON</th>
<th>Idle cycles are partly not recorded.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This setting is recommended for most accurate TRACE32 tool time stamps. (See also: ETM.TimeMode)</td>
</tr>
<tr>
<td>OFF</td>
<td>All generated trace information is recorded (diagnose purpose only).</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| **PACK** | No idle cycles are recorded. Caveats: The accuracy of the TRACE32 tool time stamp is reduced. This setting is the default with Trace.Method CAnalyzer (CombiProbe/µTrace).
This setting is recommended for trace mode **STREAM**. |
| **MAX** | No idle cycles are recorded and no TRACE32 tool time stamp is generated. This setting is recommended if the trace mode **STREAM** is used for Trace-based Code Coverage. This setting is automatically activated when the command **RTS.ON** is executed (only ETMv3 and later). |
| **AUTO** | Best setting is done automatically by TRACE32 (default). With real-time streaming (**Trace.Mode STREAM**): PortFilter operates in PACK mode With real-time profiling (**RTS.ON**): PortFilter operates in MAX mode In all other cases:  
• Analyzer (PowerTrace) operates in ON mode  
• CAAnalyzer (CombiProbe/µTrace) operates in PACK mode. |

**See also**
- **RTS.ON**
- 'Trace-based Code Coverage' in 'Nexus Training'
Embedded cores in Xilinx FPGAs [Zynq]

**<trace>.PortSize**

Set external port size

| Format: | <trace>.PortSize 1 | 2 | 3 | … | 16 | AUTO |

Informs the debugger that the externally visible port size differs from the internal port size setting of **TPIU.PortSize** and sets the specified external port size. Use this command if there is application-specific logic between the TPIU and the analyzer, for example in the programmable logic part of an FPGA SoC.

The external port size value refers to the number of data pins that are physically connected to the analyzer.

The internal port size value refers to the setting that will be programmed into the target's TPIU.

**AUTO**

(default) The external port size value of **<trace>.PortSize** equals the internal port size value of **TPIU.PortSize**.

1 … 16 Use the specified number of data pins as the external port size.

**See also**

- **TPIU.PortSize**
- "Introduction" in 'Debugging Embedded Cores in Xilinx FPGAs [Zynq]'

---

**<trace>.PortType**

Specify trace interface

| Format: | <trace>.PortType TPIU | STM | SWV (CombiProbe) |

| <trace>.PortType TPIU | TPIUX2 | TPIUX3 | TPIUX4 | STM | RTP | TPIU+RTP (Preprocessor AutoFocus II) |

| <trace>.PortType HSSTP | SETM3 (Preprocessor Serial) |

Inform TRACE32 PowerView about the trace port interface type provided by your target. This might be necessary for the following TRACE32 trace tools:

**TRACE32 CombiProbe:**

| TPIU (default) | CombiProbe is connected to TPIU. |
| STM | CombiProbe is connected to STM interface. |
| SWV | CombiProbe is connected to Serial Wire Viewer interface. |

©1989-2020 Lauterbach GmbH
TRACE32 Preprocessor AutoFocus II:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TPIU</strong></td>
<td>TRACE32 AutoFocus II Preprocessor is connected to TPIU (default). Also supported by LA-7991 PP-ARM-ETM-AF.</td>
</tr>
<tr>
<td><strong>TPIUX2</strong></td>
<td>TRACE32 AutoFocus II is connected to a trace port interface that provides 2 ETMv3 interfaces, multicore chip without TPIU (NEC Triton only).</td>
</tr>
<tr>
<td><strong>TPIUX3</strong></td>
<td>TRACE32 AutoFocus II is connected to a trace port interface that provides 3 ETMv3 interfaces, multicore chip without TPIU (NEC Triton only).</td>
</tr>
<tr>
<td><strong>TPIUX4</strong></td>
<td>TRACE32 AutoFocus II is connected to a trace port interface that provides 4 ETMv3 interfaces, multicore chip without TPIU (NEC Triton only).</td>
</tr>
<tr>
<td><strong>STM</strong></td>
<td>TRACE32 AutoFocus II Preprocessor is connected to STM interface. Also supported by LA-7991 PP-ARM-ETM-AF.</td>
</tr>
<tr>
<td><strong>RTP</strong></td>
<td>TRACE32 AutoFocus II Preprocessor is connected to Ram Trace Port interface.</td>
</tr>
<tr>
<td><strong>TPIU+RTP</strong></td>
<td>TRACE32 AutoFocus II Preprocessor is connected to a trace port interface that includes a TPIU and a Ram Trace Port interface.</td>
</tr>
</tbody>
</table>

TRACE32 Preprocessor Serial:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HSSTP</strong></td>
<td>TRACE32 Preprocessor Serial is connected to a HSSTP interface (default).</td>
</tr>
<tr>
<td><strong>SETM</strong></td>
<td>TRACE32 Preprocessor Serial is connected to a SETM interface.</td>
</tr>
</tbody>
</table>

See also
- `<trace>.state`
- `Trace`
The contents of a trigger unit counter can be displayed as a function of time. Time counters are displayed in percent and event counters as events/s. (see also Count.PROfile).

### Format:
```
<trace>.PROfile <counter> [<gate>]
```

### <gate>:
- 0.1s
- 1.0s
- 10.0s

---

**See also**
- <trace>.state
- Trace

©1989-2020 Lauterbach GmbH
The `<trace>.PROfileChart` command group displays distributions versus time graphically. For example, the command group can be used to show the processing ratio of selected tasks versus time.

See also

- `<trace>.state`
- `<trace>.Chart`
- `<trace>.PROfileChart.DIStance`
- `<trace>.PROfileChart.DURation`
- `<trace>.PROfileChart.MODULE`
- `<trace>.PROfileChart.Rate`
- `<trace>.PROfileChart.TASK`
- `<trace>.PROfileChart.STATistic`

▲ 'Release Information' in 'Release History'
### Format: \texttt{<trace>.PROfileChart.COUNTER} [\%<format>] [<record_range>] [<scale>] [\%<items>] [\%<option>]

- **<format>:**
  - \texttt{ZeroUp. [width]} | \texttt{Up. [width]} | \texttt{Down. [width]} | \texttt{Frequency. [width]} | \texttt{POWER. [width]}

- **<width>:**
  - \texttt{DEFault} | \texttt{Byte} | \texttt{Word} | \texttt{Long} | \texttt{Quad} | \texttt{TByte} | \texttt{HByte}

- **<items>:**
  - \texttt{DEFault} | \texttt{ALL} | \texttt{<cpu>} | \texttt{<signals>} | \texttt{Port[.<subitem>] | MARK[.<marker>] | ENERGY.Abs | POWER[.OFF] | SAMPLE[.OFF] | SPARE[.OFF]}

- **<option>:**
  - \texttt{FILE} | \texttt{FlowTrace} | \texttt{BusTrace} | \texttt{RecScale} | \texttt{TimeScale} | \texttt{TimeZero} | \texttt{TimeREF}
  - \texttt{<draw_options>}
    - \texttt{InterVal <time>} | \texttt{Ratio}
    - \texttt{Filter <filter_item>}
    - \texttt{Sort <item>}
    - \texttt{Track}
    - \texttt{ZoomTrack}

- **<draw_option>:**
  - \texttt{Vector} | \texttt{Steps} | \texttt{Color}

Shows the time profiles of a counter that is traced as data value.

### See also
- \texttt{<trace>.PROfileChart}
<trace>.PROfileChart.DISTance

Time interval for a single event

Format:

<trace>.PROfileChart.DISTance [<record_range>] [<scale>] [/<option>]
<trace>.Chart.DISTance (deprecated)

<option>:

FILE
FlowTrace | BusTrace | CTS
Track
RecScale | TimeScale | TimeZero | TimeREF
Filter <item>

Display the time interval for a single event graphically. For the general options for all <trace>.PROfileChart commands refer to <trace>.Chart.

Example for TRACE32 ICD and TRACE32 PowerTrace:

If no selective tracing is possible, use the option /Filter to filter out the event of interest.

B::Go
B::Break
B::Trace.PROfileChart.DISTance /FILTER Address InterruptEntry

Example for TRACE32-ICE or TRACE32-FIRE:

Perform a selective trace on the event of interest.

Analyzer.ReProgram
(  ADDR AlphaB InterruptEntry ; program the analyzer to sample all
   Sample.enable IF AlphaBreak ; InterruptEntry
)  
Go  ; measurement
...  
Break
Trace.PROfileChart.DISTance ; display the time interval between
  ; the sampled InterruptEntry
 ; graphically
<table>
<thead>
<tr>
<th>time</th>
<th>7.000ms</th>
<th>-236.000ms</th>
<th>-235.000ms</th>
<th>-234.000ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.000us</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36.000us</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32.000us</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28.000us</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.000us</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**See also**
- `<trace>.PROfileChart`
Shows a graphical representation of the specified trace item as a percentage of a timeslice.

Examples:

; Display distribution of data value for AVG_QADC
Trace.PROfileChart.DistriB Data.L /Filter Address AVG_QADC

; Print distribution of data value written for AVG_QADC for the
; record range (-2000.)--(-1000.)
WinPrint.Trace.PROfileChart.DistriB (-2000.)--(-1000.) Data.L /Filter
Address V.RANGE(AVG_QADC)

See also
- <trace>.PROfileChart
- MIPS.PROfileChart
Graphical display of time intervals between two events.

**TRACE32 PowerTrace**

In order to use the command `Trace.STATistic.DURation`:

- Check if both events are exported by a trace packet. Information reconstructed by TRACE32 is not analyzed.
- Alternatively use a `TraceEnable` breakpoint export the event as a trace packet.

The options `FilterA` and `FilterB` provide you with the means to describe your event.

- **FilterA** `<item>` Specify the first event.
- **FilterB** `<item>` Specify the second event.
Trace.Mode Leash

Break.Set 0x9cb0 /Program /TraceEnable

Break.Set 0x9e3c /Program /TraceEnable

Go

WAIT !STATE.RUN()

Trace.STATistic.DURation /FilterA Address 0x9cb0 /FilterB Address 0x9e3c

Trace.PROfileChart.DURation /FilterA Address 0x9cb0
/FilterB Address 0x9e3c

Displays min and max duration per 10 pixels
Displays min and max duration per 10 pixels (with a higher resolution)
Perform a selective trace on both events. Mark the first event in the trace with A and the second event with B.

```
Analyzer.ReProgram
(
    ADDR AlphaBreak InterruptEntry
    ADDR BetaBreak InterruptExit
)
Sample.enable IF AlphaBreak | BetaBreak
Mark.A IF AlphaBreak
Mark.B IF BetaBreak

E::Go
E::Break
```

E::Trace.PROfileChart.DURation

- **ATOA**
  Display the time interval from A to A.

- **BTOA**
  Display the time interval from B to A.

- **BTOB**
  Display the time interval from B to B.
  If no selective tracing is possible and more specific events should be displayed, it is also possible to use the options:

By default, the time is taken from the A marked event to the B marked event.

**See also**

- `<trace>.PROfileChart`
Group profile chart

Format:  

```
<trace>.PROfileChart.GROUP [<record_range>] [<scale>] [/<option>]
```

<option>:

- FILE
- FlowTrace | BusTrace | CTS
- Track
- RecScale | TimeScale | TimeZero | TimeREF
- Filter <item>

Analyzes the group behavior and displays the result as a color chart with fixed time intervals.

Example:

```
GROUP.Create "INPUT" \jquant2 \jquant1 \jidctred \jinput /AQUA
GROUP.Create "JPEG" \jdapimin \jdcolor \jddctmgr \jdoeffct /NAVY
Go
Break
Trace.PROfileChart.GROUP
```

See also

- `<trace>.PROfileChart`
- 'Release Information' in 'Release History'

©1989-2020 Lauterbach GmbH
<trace>.PROfileChart.LINE [<record_range>] [<scale>] [/<option>]

<option>:
  FILE
  FlowTrace | BusTrace
  TASK | SplitTASK | MergeTASK
  Track | ZoomTrack
  RecScale | TimeScale | TimeZero | TimeREF
  INCremental | FULL
  FILTER <item>
  Sort <item>

Analyzes the dynamic program behavior for high-level code lines and displays the result as a color chart with fixed time intervals.

Trace.PROfileChart.Line is based on a flat function run-time analysis.

<record_range> For parameter descriptions and examples, see Parameters.

<option> For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

See also
  ■ <trace>.PROfileChart
Module profile chart

Format:  
<trace>.PROfileChart.MODULE [<record_range>] [<scale>] [/<option>]

<option>:  
FILE
FlowTrace | BusTrace
TASK | SplitTASK | MergeTASK
Track | ZoomTrack
RecScale | TimeScale | TimeZero | TimeREF
INCremental | FULL
FILTER <item>
Sort <item>

Analyzes the dynamic program behavior for modules and displays the result as a color chart with fixed time intervals.

Trace.PROfileChart.MODULE is based on a flat function run-time analysis.

<record_range>  
For parameter descriptions and examples, see Parameters.

<option>  
For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

See also

- <trace>.PROfileChart
Program profile chart

Format:  
<trace>.PROfileChart.PROGRAM [<record_range>] [<scale>] [/<option>]

<option>:  
FILE  
FlowTrace | BusTrace  
TASK | SplitTASK | MergeTASK  
Track | ZoomTrack  
RecScale | TimeScale | TimeZero | TimeREF  
INCremental | FULL  
FILTER <item>  
Sort <item>

Analyzes the dynamic execution behavior broken down by program and displays the result as a color chart with fixed time intervals.

Trace.PROfileChart.PROGRAM is based on a flat function run-time analysis.

<record_range>  
For parameter descriptions and examples, see Parameters.

<option>  
For the general options for all <trace>.Chart commands, refer to <trace>.Chart.

See also  
- <trace>.PROfileChart
Graphical display of the event frequency over the time. For the general options for all `<trace>.PROfileChart` commands refer to `<trace>.Chart`.

**Example:**

Display the **TARGET FIFO OVERFLOW** (FIFOFULL) rate over the time.

```
Trace.PROfileChart.Rate /Filter FIFOFULL
```
Example for TRACE32-ICD and TRACE32-PowerTrace:

If no selective tracing is possible, use the option `/Filter` to filter out the data of interest.

```
B::Go
B::Break
B::B::Trace.PROfileChart.Rate /FILTER Address InterruptEntry
```

Example for TRACE32-ICE and TRACE32-FIRE:

Perform a selective trace on the data of interest.

```
Analyzer.ReProgram
(
  ADDR AlphaBreak InterruptEntry ; sample all InterruptEntry
  Sample.Enable IF AlphaBreak
)
Go ; collect data
Break
Trace.PROfileChart.Rate ; Display event frequency of
                          ; InterruptEntry graphically
```

```
E::Trace.PROfileChart.Rate
```

<table>
<thead>
<tr>
<th>events/s</th>
<th>0s</th>
<th>-40.000s</th>
<th>-30.000s</th>
<th>-20.000s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>144</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>112</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See also
- `<trace>.PROfileChart`
Analyzes the dynamic program behavior and displays the result as a color chart with fixed time intervals.

**Trace.PROfileChart.sYmbol** is based on a flat function run-time analysis.

Push the **Profile** button to get information on the dynamic behavior of the program.

To draw the **Trace.PROfileChart.sYmbol** graphic, TRACE32 PowerView partition the recorded instruction flow information into time intervals. The default interval size is 10.us.
For each time interval rectangles are draw that represent the time ratio the executed functions consumed within the time interval. For the final display this basic graph is smoothed.

<table>
<thead>
<tr>
<th>Fine</th>
<th>Decrease the time interval size by the factor 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse</td>
<td>Increase the time interval size by the factor 10</td>
</tr>
</tbody>
</table>

The time interval size can also be set manually.

```
Trace.PROfileChart.Symbol /InterVal 5.ms ; change the time interval size to 5.ms
```
Color Assignment - Basics

- The tooltip at the cursor position shows the color assignment and the used interval size.

- Use the control handle on the right upper corner of the Trace.PROfileChart.sYmbol window to get a color legend.
Color Assignment - Statically or Dynamically

<table>
<thead>
<tr>
<th>FixedColors</th>
<th>Colors are assigned fixed to functions (default). Fixed color assignment has the risk that two functions with the same color are drawn side by side and thus may convey a wrong impression of the dynamic behavior.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlternatingColors</td>
<td>Colors are assigned by the recording order of the functions again and again for each measurement.</td>
</tr>
</tbody>
</table>

See also

- `<trace>.PROfileChart`
<trace>.PROfileChart.TASK

Dynamic task behavior graphically (flat)

Format:  
<trace>.PROfileChart.TASK [<string> | <range> | <value>] [/<option>]

<option>:
FILE  | FlowTrace  | BusTrace
RecScale | TimeScale | TimeZero | TimeREF
<draw_options>
Interval <time>
Filter <filter_items>
Sort <item>
Track
ZoomTrack

<draw_option>:
Vector  | Steps  | Color

Analyzes the dynamic task behavior and displays the result as a color chart with fixed time intervals. This command requires **OS-ware tracing**.

**Example to analyze CPU load:**

```plaintext
; group all tasks that contain an idle loop to the group "Idle"
; all other tasks are members of the group "other"

; merge the result of all "Idle" group members and
; use white as "Idle" group color
GROUP.CreateTASK "Idle" "Idle_Task" /Merge /WHITE

; merge the result of all "other" group members
GROUP.Merge "other"

; use green as "other" group color
GROUP.COLOR "other" GREEN

; display the CPU load graphically
Trace.PROfileChart.TASK
```
(unknown) represents the time before the first task information was recorded to the trace.

See also

- `<trace>.PROfileChart`
The command group `<trace>.PROfileSTATistic` shows the results of numerical interval analysis in tabular format.

**See also**
- `<trace>.state`
- `Trace`
- `<trace>.Chart`
- `<trace>.PROfileChart`

▲ 'Release Information' in 'Release History'

### `<trace>.PROfileSTATistic.Line`

#### Statistical analysis for HLL lines

Format:  
`<trace>.PROfileSTATistic.Line [<trace_area>] [/<option>]`

#### Format:

`<trace_area>`:
- `<trace_bookmark>` | `<record>` | `<record_range>` | `<time>` | `<time_range>`
  `[<time_scale>]`

Shows the results of numerical interval analysis in tabular format for HLL code lines.
<trace>.Program

Program trigger unit

FIRE / ICE only

Format:  <trace>.Program [<file>]

Opens an editor window, which is used for programming the trigger unit. The program entry in the window is guided by softkeys and the online-help system. To program the trace press the **Compile** button. After successful programming, the file name will be displayed in the trace status window. The default file name is the file currently used by the trigger unit. The file name default extension is `.ts`. This command can be executed by clicking the **Edit** button in the trace state window. If the state of the trace is ARM and this command is executed, the state is switched to **OFF** before programming the trigger unit, automatically. See **Analyzer Programming Guide** for details about the programming language.

![Image of a program window with code snippets]

See also

- `<trace>.ReProgram`
- `<trace>.state`
- Trace
- SETUP.EDITOR

▲ 'Text Editors' in 'PowerView User's Guide'
▲ 'Trigger Programming' in 'Training FIRE Analyzer'
▲ 'Release Information' in 'Release History'
See also
- <trace>.PROTOcol.STATistic
- <trace>.PROTOcol.Draw
- <trace>.PROTOcol.Find
- <trace>.PROTOcol.PROfileChart
- <trace>.state
- <trace>.PROTOcol.Chart
- <trace>.PROTOcol.EXPORT
- <trace>.PROTOcol.List
- <trace>.PROTOcol.STATistic
- Trace

<trace>.PROTOcol.Chart

Graphic display for user-defined protocol

Format:  
<trace>.PROTOcol.Chart <protocol> <parlist> [ <items> … ] [/<options>]

<parlist>:  
<line> | <lines> <options>

<line>:  
<signal>

<lines>:  
<signal>…

=options>:  
<options for ASYCH>
<options for CAN>
<options for I2C>
<options for JTAG>
<options for USB>
<user_defined_options>

<option>:  
FILE
Track
CTS
RecScale
TimeScale
TimeZero
TimeREF
Filter

<items>:  
%<format>
<line>
DEFault | ALL
TIme.Back | TIme.Fore | TIme.REF
TIme.Zero | TIme.Trigger
SyncClock
SPARE
### Options:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE</strong></td>
<td>Display trace memory contents loaded with the command <code>Trace.FILE</code>.</td>
</tr>
<tr>
<td><strong>Track</strong></td>
<td>Track other trace list window (tracks to record number or time)</td>
</tr>
<tr>
<td><strong>RecScale</strong></td>
<td>Record Scaling</td>
</tr>
<tr>
<td><strong>TimeScale</strong></td>
<td>Timed Scaling</td>
</tr>
<tr>
<td><strong>Filter</strong></td>
<td></td>
</tr>
</tbody>
</table>

### Items:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time.REF</strong></td>
<td>Time marker, relative to reference point</td>
</tr>
<tr>
<td><strong>Time.Trigger</strong></td>
<td>Time marker, relative to trigger point</td>
</tr>
<tr>
<td><strong>Time.Zero</strong></td>
<td>Time marker, relative to global reference</td>
</tr>
<tr>
<td><strong>SyncClock</strong></td>
<td>Synchronous clock event</td>
</tr>
</tbody>
</table>

**Example:** Display the user-defined protocol “proto1” on line x.0

```plaintext
PP::Analyzer.PROTOCOL.Chart proto1 x.0
```

**See also**

- `<trace>.PROTOcol`
Format:  
<trace>.PROTOcol.Draw <protocol> <parlist> [ <items> … ] [ / <options> ]

<parlist>:  
<line> | <lines> <options>

<line>:  
<signal>

<lines>:  
<signal> …

<options>:  
=options for ASYCH>  
=options for CAN>  
=options for I2C>  
=options for JTAG>  
,options for USB>  
<user_defined_options>

<option>:  
FILE  
Track  
CTS  
RecScale  
TimeScale  
TimeZero  
TimeREF  
Filter

<items>:  
%<format>  
<line>  
DEFault | ALL  
Time.Back | Time.Fore | Time.REF  
Time.Zero | Time.Trigger  
SyncClock  
SPARE

Options:

FILE  
Display trace memory contents loaded with the command Trace.FILE.

Track  
Track other trace list window  
(tracks to record number or time)

RecScale  
Record Scaling

TimeScale  
Timed Scaling

Filter  
tbd.
Items:

- **TIme.REF**  Time marker, relative to reference point
- **TIme.Trigger**  Time marker, relative to trigger point
- **TIme.Zero**  Time marker, relative to global reference
- **SyncClock**  Synchronous clock event

**Example:** Display the user-defined protocol “proto1” on line x.0

```c
PP::Analyzer.PROTOcol.Draw proto1 x.0
```

**See also**
- `<trace>.PROTOcol`

---

**<trace>.PROTOcol.EXPORT**  Export trace buffer for user-defined protocol

**Format:**

```c
<trace>.PROTOcol.EXPORT.W <protocol> <parlist> <file> [range] …
```

**Example:** Export the user-defined protocol “proto1” on line x.0 to test.lst

```c
PP::a.proto.EXPORT proto1 x.0 test.lst
```
Find in trace buffer for user-defined protocol

Format: \texttt{<trace>.PROTOcol.Find <protocol> <parlist> [<items> ...] [/<options>]}

\texttt{<parlist>}: \texttt{<line> | <lines> <options>}

\texttt{<line>}: \texttt{<signal>}

\texttt{<lines>}: \texttt{<signal> ...}

\texttt{<options>}: \texttt{<options for ASYCH>}
\texttt{<options for CAN>}
\texttt{<options for I2C>}
\texttt{<options for JTAG>}
\texttt{<options for USB>}
\texttt{<user_defined_options>}

\texttt{<option>}: \texttt{FILE}
\texttt{Back}
\texttt{NoFind}

Options:

- **FILE**: Display trace memory contents loaded with the command \texttt{Trace.FILE}.
- **Back**: Search back
- **NoFind**

**Example**: Find in the user-defined protocol “proto1” on line x.0

\texttt{PP::a.proto.f proto1 x.0}

See also

- \texttt{<trace>.PROTOcol}
- \texttt{<trace>.PROTOcol.List}
### Format:

```
<trace>.PROTOcol.List <protocol> <parlist> [{<items>}] [/<options>]
```

### <parlist>:

```
<line> | <lines> <options>
```

### <line>:

```
<signal>
```

### <lines>:

```
<signal> ...
```

### <options>:

```
<options for ASYCH>
<options for CAN>
<options for I"2C>
<options for JTAG>
<options for USB>
<user_defined_options>
```

### <option>:

```
FILE
Track
```

### <items>:

```
%<format>

<line>
DEFault | ALL
Time.Back | Time.Fore | Time.REF
Time.Zero | Time.Trigger
SyncClock
SPARE
```

### <format>:

```
Hex | Decimal | BINary | Ascii
Timing | HighLow
LEN <size>
TimeAuto | TimeFixed
```

### Options:

- **FILE**
  
  Display trace memory contents loaded with the command `Trace.FILE`.

- **Track**
  
  Track other trace list window  
  (tracks to record number or time)

### Formats:

- **Timing**
  
  Display single bits as vertical timing

- **HighLow**
  
  Display single bits as HIGH/LOW value

- **Hex**
  
  Display single bytes in hex values
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Display single bytes in decimal values</td>
</tr>
<tr>
<td>BINary</td>
<td>Display single bytes in binary values</td>
</tr>
<tr>
<td>Ascii</td>
<td>Display single bytes as ascii characters</td>
</tr>
<tr>
<td>LEN &lt;size&gt;</td>
<td>Specify the width of non numeric fields (e.g. symbols)</td>
</tr>
<tr>
<td>TimeAuto</td>
<td>The unit of time is selected automatically.</td>
</tr>
<tr>
<td>TimeFixed</td>
<td>The displayed unit of time is fixed.</td>
</tr>
</tbody>
</table>

**Items:**

- **DEFault** Default selections (see list below)
- **ALL** Select all recorded data (superset of DEFault)
- **Time** Time marker (default Time.Fore)
- **Time.Back** Time marker, relative time to previous record
- **Time.Fore** Time marker, relative time to next record
- **Time.REF** Time marker, relative to reference point
- **Time.Trigger** Time marker, relative to trigger point
- **Time.Zero** Time marker, relative to global reference
- **SyncClock** Synchronous clock event
- **SPARE** Displays an empty block

**Example:** Displays the user-defined protocol “proto1” on line x.0

```
PP::a.proto.l proto1 x.0
```
; JTAG <tck> <tms> <tdi> <tdo> <trst> <initstate>
; when the sampling is started the JTAG state machine is in state
; run-test/idle
Trace.ListProtocol JTAG X.8 X.9 X.4 X.12 X.14 run-test/idle

; CAN <canline> <frequency> DEFault | ALL
; the frequency is defined in Hz
Trace.ListProtocol CAN X.7 1000000. DEFault

; USB <+signal> <-signal>
Trace.ListProtocol USB X.17 X.18

; I2C <scl> <sda>
Trace.ListProtocol I2C X.22 X.23

; asynchronous communication interface
; ASYNC <asylime> <frequency> +|- <parity> <length> <stopbit>

Trace.ListProtocol ASYNC X.6 3600. + EVEN 7 1STOP STRING
Trace.ListProtocol ASYNC X.5 2400. - NONE 5 2STOP CHAR

; special protocols
; TRACE32 offers a API that allows to use special, customer specific
; protocols
Trace.ListProtocol protojtag.dll X.4 X.12 X.14

; examples for special protocols are provided under ~/demo/proto
; on the TRACE32 software DVD

See also
- <trace>.PROTOCOL
- <trace>.PROTOCOL.Find
- <trace>.PROTOCOL.STATistic
- <trace>.REF

▲ 'Release Information' in 'Release History'
### Format:

```plaintext
<trace>.PROTOcol.PROfileChart <protocol> <parlist> [<items> ...] [/<options>]
```

#### <parlist>:

```plaintext
<line> | <lines> <options>
```

#### <line>:

```plaintext
<signal>
```

#### <lines>:

```plaintext
<signal>...
```

#### <options>:

```plaintext
<options for ASYCH>
<options for CAN>
<options for I2C>
<options for JTAG>
<options for USB>
<user_defined_options>
```

#### <option>:

- `FILE`
- `Track`
- `CTS`
- `RecScale`
- `TimeScale`
- `TimeZero`
- `TimeREF`
- `Filter`

#### <items>:

```plaintext
%<format>
```

```plaintext
<line>
```

```plaintext
DEFault | ALL
```

```plaintext
Time.Back | Time.Fore | Time.REF
```

```plaintext
Time.Zero | Time.Trigger
```

```plaintext
SyncClock
```

```plaintext
SPARE
```

### Options:

- **FILE**
  - Display trace memory contents loaded with the command `Trace.FILE`.
- **Track**
  - Track other trace list window (tracks to record number or time)
- **RecScale**
  - Record Scaling
- **TimeScale**
  - Timed Scaling
- **Filter**
## Items:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time.REF</td>
<td>Time marker, relative to reference point</td>
</tr>
<tr>
<td>Time.Trigger</td>
<td>Time marker, relative to trigger point</td>
</tr>
<tr>
<td>Time.Zero</td>
<td>Time marker, relative to global reference</td>
</tr>
<tr>
<td>SyncClock</td>
<td>Synchronous clock event</td>
</tr>
</tbody>
</table>

### See also
- `<trace>.PROTOcol`
### Format:

```
<trace>.PROTOCOL.STATISTIC  <protocol>  <parlist> [ [ <items> ] ] [ <options> ]
```

<table>
<thead>
<tr>
<th><code>&lt;parlist&gt;</code>:</th>
<th><code>&lt;line&gt;</code></th>
<th><code>&lt;lines&gt;</code></th>
<th><code>&lt;options&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;line&gt;</code>:</td>
<td><code>&lt;signal&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;lines&gt;</code>:</td>
<td><code>&lt;signal&gt;</code></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>&lt;options&gt;</code>:</th>
<th><code>&lt;options for ASYCH&gt;</code></th>
<th><code>&lt;options for CAN&gt;</code></th>
<th><code>&lt;options for I2C&gt;</code></th>
<th><code>&lt;options for JTAG&gt;</code></th>
<th><code>&lt;options for USB&gt;</code></th>
<th><code>&lt;user_defined_options&gt;</code></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><code>&lt;option&gt;</code>:</th>
<th>FILE</th>
<th>BEFORE</th>
<th>AFTER</th>
<th>List</th>
<th>Filter</th>
<th>Accumulate</th>
<th>INcremental</th>
<th>FULL</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><code>&lt;items&gt;</code>:</th>
<th><code>%&lt;format&gt;</code></th>
<th><code>&lt;line&gt;</code></th>
<th>DEFault</th>
<th>ALL</th>
<th>TIme.Back</th>
<th>TIme.Fore</th>
<th>TIme.REF</th>
<th>TIme.Zero</th>
<th>TIme.Trigger</th>
<th>SyncClock</th>
<th>SPARE</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><code>&lt;format&gt;</code>:</th>
<th>Hex</th>
<th>Decimal</th>
<th>BINary</th>
<th>Ascii</th>
<th>Timing</th>
<th>HighLow</th>
<th>LEN <code>&lt;size&gt;</code></th>
<th>TimeAuto</th>
<th>TimeFixed</th>
</tr>
</thead>
</table>

### Options:

- **FILE**
  Display trace memory contents loaded with the command `Trace.FILE`.

- **Track**
  Track other trace list window
  (tracks to record number or time)
Example: Display statistics in the user-defined protocol “proto1” on line x.0

PP::a.proto.stat proto1 x.0

See also
- <trace>.PROTOcol
- <trace>.PROTOcol.List
## Protocol specific Options

### Options for ASYNC

<table>
<thead>
<tr>
<th>&lt;line&gt;</th>
<th>&lt;trans&gt;</th>
<th>&lt;rec&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;trans&gt;</td>
<td>&lt;signal&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;rec&gt;</td>
<td>&lt;signal&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;options&gt;</td>
<td>&lt;parlist&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;parlist&gt;</td>
<td>[&lt;baudrate&gt; [&lt;polarity&gt; [&lt;parity&gt; [&lt;bits&gt; [&lt;stopbits&gt; [&lt;disp&gt;]]]]]]]</td>
<td></td>
</tr>
<tr>
<td>&lt;baudrate&gt;</td>
<td>1. ... 100000.</td>
<td></td>
</tr>
<tr>
<td>&lt;polarity&gt;</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>&lt;parity&gt;</td>
<td>NONE</td>
<td>ODD</td>
</tr>
<tr>
<td>&lt;bits&gt;</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>&lt;stopbits&gt;</td>
<td>1STOP</td>
<td>2STOP</td>
</tr>
<tr>
<td>&lt;disp&gt;</td>
<td>CHAR</td>
<td>STRING</td>
</tr>
</tbody>
</table>

### Options for CAN

<table>
<thead>
<tr>
<th>&lt;line&gt;</th>
<th>&lt;signal&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;options&gt;</td>
<td>&lt;frequency&gt;</td>
</tr>
<tr>
<td>&lt;frequency&gt;</td>
<td>frequency in Hz</td>
</tr>
</tbody>
</table>
### Options for I2C

- `<lines>`: `<sck> <sda>`
- `<sck>`: `<signal>`
- `<sda>`: `<signal>`

### Options for JTAG

- `<lines>`: `<tck> <tms> <tdi> <tdo> [<trst>] [<initial_state>]`
- `<tck>`: `<signal>`
- `<tms>`: `<signal>`
- `<tdi>`: `<signal>`
- `<tdo>`: `<signal>`
- `<trst>`: `<signal>`
- `<initial_state>`: Exit2-DR, Exit1-DR, Shift-DR, Pause-DR, Select-IR-scan, Update-DR, Capture-DR, Select-DR-scan, Exit2-IR, Exit1-IR, Shift-IR, Pause-IR, Run-Test/Idle, Update-IR, Capture-IR, Test-Logic-Reset
Options for USB

<lines>:  <+signal> <-signal> [<state>]

 <+signal>:  <signal>
  <-signal>:  <signal>

<state>:  BUSRESET
          GAP
          SYNC
          EOP
          ERRORS
          SOF
          #SETUP
          #PRE
          #IN
          #OUT
          #ACK
          #NACK
          #STALL
          #DATA0
          #DATA1
          #OTHER
Selects the sampling rate in samples/s. Legal values and the resulting sample times are given in the following table:

### PortAnalyzer ICE

<table>
<thead>
<tr>
<th>Rate</th>
<th>Resolution</th>
<th>Total sampling time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000.</td>
<td>100 ns</td>
<td>3.27 ms</td>
</tr>
<tr>
<td>5000000.</td>
<td>200 ns</td>
<td>6.55 ms</td>
</tr>
<tr>
<td>2500000.</td>
<td>400 ns</td>
<td>13.1 ms</td>
</tr>
<tr>
<td>1000000.</td>
<td>1 µs</td>
<td>32.7 ms</td>
</tr>
<tr>
<td>500000.</td>
<td>2 µs</td>
<td>65.5 ms</td>
</tr>
<tr>
<td>250000.</td>
<td>4 µs</td>
<td>131.0 ms</td>
</tr>
<tr>
<td>200000.</td>
<td>5 µs</td>
<td>164.0 ms</td>
</tr>
<tr>
<td>100000.</td>
<td>10 µs</td>
<td>327.0 ms</td>
</tr>
<tr>
<td>50000.</td>
<td>20 µs</td>
<td>655.0 ms</td>
</tr>
<tr>
<td>25000.</td>
<td>40 µs</td>
<td>1.31 s</td>
</tr>
<tr>
<td>20000.</td>
<td>50 µs</td>
<td>1.64 s</td>
</tr>
<tr>
<td>10000.</td>
<td>100 µs</td>
<td>3.27 s</td>
</tr>
<tr>
<td>5000.</td>
<td>200 µs</td>
<td>6.55 s</td>
</tr>
<tr>
<td>2500.</td>
<td>400 µs</td>
<td>13.1 s</td>
</tr>
<tr>
<td>2000.</td>
<td>500 µs</td>
<td>16.4 s</td>
</tr>
<tr>
<td>1000.</td>
<td>1 ms</td>
<td>32.7 s</td>
</tr>
<tr>
<td>500.</td>
<td>2 ms</td>
<td>65.5 s</td>
</tr>
<tr>
<td>250.</td>
<td>4 ms</td>
<td>131.0 s</td>
</tr>
<tr>
<td>200.</td>
<td>5 ms</td>
<td>164.0 s</td>
</tr>
<tr>
<td>100.</td>
<td>10 ms</td>
<td>327.0 s</td>
</tr>
</tbody>
</table>
With a mouse click to the corresponding area in the port analyzer state window that command can be executed, too.

Port.Rate 1000000. ; set to 1 MHz sample rate
Port.Rate 1.us ; same operation, 1 MHz sample rate

<trace>.REF Set reference point for time measurement

Format: <trace>.REF [<time> | <record> | "<trace_bookmark>"

<option>: FILE

Sets the reference point for time measurements using the TIme.REF column of the Trace.List window. The default reference point is always the last record in trace memory. The reference point can also be set via context menu entry Set Ref in Trace.List, Trace.Chart, Trace.Timing etc. windows.

<time> Sets the reference point to the global ZERO point. If the time for each trace event is calculated based on timestamps generated by the processor, the global ZERO point is at the start of the trace recording currently stored in the trace buffer. If the time for each trace event is based on timestamps generated by the trace module, the global ZERO point is set to the start of the first debug session after the start of TRACE32 PowerView.

<record> Sets the reference point to the time index of the specified record number.

<trace_bookmark> Sets the reference point to the time index of the specified bookmark location. You can create trace bookmarks with the <trace>.BookMark command.

Examples:

Trace.REF +2000. ; set reference to record +2000
Trace.REF 100us ; set ref. point to absolute time
Trace.REF "MyRef" ; set ref. point to bookmark "MyRef"

See also
- <trace>.GOTO
- <trace>.PROTOcol.List
- <trace>.state
- <trace>.Timing
- <trace>.View
- Trace
- Analyzer.REF()
<trace>.ReProgram

Program trigger unit

Format:  \(<trace>.\text{ReProgram} [<file>]\)

Programs the trigger unit of the trace with an existing program. The program should be error free, when using this command. To write an error free program, use the command \(<trace>.\text{Program}\).

See also

- \(<trace>.\text{Program}\)
- \(<trace>.\text{state}\)
- Trace

▲ 'Release Information' in 'Release History'
▲ 'Trigger Programming' in 'Training FIRE Analyzer'

<trace>.RESet

Reset command

Format:  \(<trace>.\text{RESet}\)

Resets the trace unit to its default settings.

See also

- \(<trace>.\text{state}\)
- CIProbe
- Trace

▲ 'Emulator Functions' in 'FIRE User's Guide'
Set AutoFocus sample time offset

Use this command to manually configure sample times of the trace channels. It is typically used to restore values previously stored using the store button of the Trace.ShowFocus window, or the STOre ANALYZERFOCUS command.

**Format:** \(<trace>.SAMPLE [\langle channel\rangle] \langle time1\rangle [\langle time2\rangle]\)

Use this command to manually configure sample times of the trace channels. It is typically used to restore values previously stored using the store button of the Trace.ShowFocus window, or the STOre ANALYZERFOCUS command.

**<channel>**
Trace signal to be configured
If the parameter is omitted, all signals are configured with the \(<time>\) setting.

**<time1>**
Parameter Type: Float. The value is interpreted as time in nanoseconds.
Sample time offset to trace clock.
- Positive value: later
- Negative value: earlier

**<time2>**
Parameter Type: Float. The value is interpreted as time in nanoseconds.
For double data rate mode only:
Sample time offset to inverted trace clock (usually this is the sample time relative to the falling edge of the trace clock).
If the parameter is omitted, \(<time1>\) is used for both sample time offsets.

The available AutoFocus settings and features depend on the preprocessor used:

<table>
<thead>
<tr>
<th>Preprocessor:</th>
<th>Features:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoFocus II</td>
<td>configuration:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time resolution:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max. range:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max. inter-channel range:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>individual sample time per channel about 78ps per step</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+/- 63 * 0.078ns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ABS(MAX(time1)-MIN(time1)) &lt; 63*0.078ns</td>
<td></td>
</tr>
<tr>
<td>NEXUS-AutoFocus</td>
<td>configuration:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time resolution:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max. range:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>single sample time applied to all channels about 750ps per step</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+/- 15 * 0.75ns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ABS(time1-time2) &lt; 15*0.75ns</td>
<td></td>
</tr>
</tbody>
</table>

**See also**
- \(<trace>.state\)
- Trace
- 'Release Information’ in 'Release History’

©1989-2020 Lauterbach GmbH
The trace memory contents are stored to the selected file. What is actually saved to the file depends on the Trace.Mode:

- **FlowTrace**: Trace raw data plus decompressed addresses, data and op-codes are saved.
  
  If the program and data flow is output by the CPU in a compressed format, it is decompressed before saving it to a file for postprocessing. By reading the source code information the addresses, data value and op-codes are decompressed.

- All other settings for **FlowTrace**: Only the trace raw data are saved into `<file>` if the information from the external busses, ports etc. are recorded to the trace buffer.

### Parameters

<table>
<thead>
<tr>
<th><code>&lt;file&gt;</code></th>
<th>The default extension of the file name is <code>.ad</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For some TRACE32 devices additional setting are saved to <code>&lt;file&gt;</code>:</td>
</tr>
<tr>
<td></td>
<td>• PowerProbe (<strong>Trace.METHOD Probe</strong>) All Probe settings and all <strong>NAME</strong> settings are saved to <code>&lt;file&gt;</code>.</td>
</tr>
<tr>
<td></td>
<td>• PowerIntegrator (<strong>Trace.METHOD Integrator</strong>) All <strong>Integrator</strong> settings and all <strong>NAME</strong> settings are saved to <code>&lt;file&gt;</code>.</td>
</tr>
</tbody>
</table>
### Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowTrace</td>
<td>Obsolete.</td>
</tr>
<tr>
<td>BusTrace</td>
<td>Save only the trace raw data, if a flow trace is used. This option is helping if a decompression of the program and data trace information is not possible.</td>
</tr>
<tr>
<td>PACK</td>
<td>Save the trace contents is a compact way. PACK is less effective and slower than ZIP. It is only recommended if the option ZIP is not available.</td>
</tr>
<tr>
<td>ZIP</td>
<td>Save the trace contents is a compact way.</td>
</tr>
<tr>
<td>NoCompress</td>
<td>Obsolete.</td>
</tr>
</tbody>
</table>
### Example for `<trace_bookmark>`

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Trace.List</code></td>
</tr>
<tr>
<td><code>Trace.BookMark &quot;First&quot; -123366.;</code></td>
</tr>
<tr>
<td><code>Trace.BookMark &quot;Last&quot; -36675.;</code></td>
</tr>
<tr>
<td><code>BookMark.List</code></td>
</tr>
<tr>
<td><code>Trace.Save testb &quot;First&quot; &quot;Last&quot;</code></td>
</tr>
</tbody>
</table>

*Display trace listing*

*Mark the trace record -123366. with the bookmark "First"*

*Mark the trace record -36675. with the bookmark "Last"*

*List all bookmarks*

*Save trace contents between bookmarks "First" and "Last" to the file testb*

### Example for `<record_range>`

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Trace.List</code></td>
</tr>
<tr>
<td><code>Trace.Save testr (-107032.)--(-21243.)</code></td>
</tr>
</tbody>
</table>

### Example for `<record>` `<record2>`

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Trace.List</code></td>
</tr>
<tr>
<td><code>Trace.Save testv -107032. -21243.</code></td>
</tr>
</tbody>
</table>

### Example for `<timerange>`

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Trace.List TIme.Zero DEFault</code></td>
</tr>
<tr>
<td><code>Trace.Save testt 4.us--1.952ms</code></td>
</tr>
</tbody>
</table>

©1989-2020 Lauterbach GmbH
More Examples

```
Trace.SAVE test4 ; save trace contents to the file test4
QUIT ; end TRACE32

; off-line postprocessing of the trace contents e.g. with a TRACE32 Instruction Set Simulator

Trace.LOAD test4 ; load the saved trace contents

Data.LOAD.Elf demo.elf /NoCODE ; load the symbol information if you like to have HLL information for the trace analysis

Trace.List ; display the loaded trace contents as trace listing

Trace.STATistic.Func ; perform a function run-time analysis on the loaded trace contents;
; HLL information is required

; save trace contents to the file test4
Trace.SAVE test4

... ; use saved trace contents as reference

; load the saved trace contents
Trace.FILE test4

; display the trace contents loaded from the file test4.ad as trace listing
Trace.List /FILE

; compare the current trace contents with the trace contents loaded from test4.ad with regards to the addresses
Trace.ComPare (-27093.)--(-8986.) Address /FILE
```

See also

- `<trace>.EXPORT`
- `<trace>.LOAD`
- `<trace>.state`
- `<trace>.STREAMSAVE`
- 'Emulator Functions' in 'FIRE User's Guide'
- 'Release Information' in 'Release History'
Select trigger/counter line

Format: **Port.Select** [<channel>]

Selects a port line for the frequency counter and the glitch detector. If in the "TSelect" area of the state window "Port" is active, this port will be used to trigger the port analyzer.

For valid channel names refer to the:

- **Processor Architecture Manuals**
- **Target Guides FIRE**
- **Target Guides ICE**

**Examples:**

```
Port.Select Port.20
Port.Select Port.RXD
```

Automatic restart of trace recording

Format: `<trace>.SelfArm` [ON | OFF]

**Trace.SelfArm ON** automatically restarts the trace recording. There are mainly two use cases for this command.

**Snapshot without stopping the program execution**

If stopping the program execution it not advisable, but you are interested in the target state at a specific point of the program execution, proceed as follows:

1. Display the information of interest on the screen.
   Please make sure to display only information that can be updated while the program execution is running.
2. Use a trigger to specify your point of interest.
3. Activate the self-arm mode.

Whenever the trace recording is stopped by the trigger, all information displayed by TRACE32 is updated before the trace recording is automatically restarted.
Example for ARM11:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data.LOAD.Elf armla.Elf /PlusVM</td>
<td>load source code to virtual memory within TRACE32 in order to enable the trace display while program execution is running</td>
</tr>
<tr>
<td>Trace.List</td>
<td>display the information of interest</td>
</tr>
<tr>
<td>Break.Set sieve /TraceTrigger</td>
<td>specify the trigger point</td>
</tr>
<tr>
<td>Trace.Mode AutoInit ON</td>
<td>clear the trace buffer and re-activate the trigger before the trace recording is automatically restarted</td>
</tr>
<tr>
<td>Trace.Mode SelfArm ON</td>
<td>activate the self-arm mode</td>
</tr>
<tr>
<td>Go</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Trace.Mode SelfArm OFF</td>
<td>stop automatic restarting of trace recording</td>
</tr>
</tbody>
</table>

Automated run-time analysis

To automate an incremental run-time analysis, proceed as follows:

1. Prepare the run-time analysis and open a run-time analysis window.
2. Switch the trace to **Leash** mode.
3. Activate the self-arm mode.

Whenever the trace recording/program execution is stopped because the trace buffer is full, the current trace contents is analyzed and the analysis window is updated correspondingly. Afterwards the program execution is restarted.
### Example:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace.STATistic.Func /Accumulate</td>
<td>open a window that performs a continuous nested function run-time analysis</td>
</tr>
<tr>
<td>Trace.Mode Leash</td>
<td>switch the trace to Leash mode</td>
</tr>
<tr>
<td>Trace.Mode AutoInit ON</td>
<td>clear the trace buffer before the trace recording is automatically restarted</td>
</tr>
<tr>
<td>Trace.Mode SelfArm ON</td>
<td>activate the self-arm mode</td>
</tr>
<tr>
<td>Go</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Trace.Mode SelfArm OFF</td>
<td>stop automatic restarting of trace recording</td>
</tr>
</tbody>
</table>

#### See also

- `<trace>.SnapShot`
- `<trace>.state`
- `Trace`

### `<trace>.SET`

**Select line for recording**

Format:  

```
Port.SELECT [channel]
```

Selects a port line for recording.
The **Trace.ShowFocus** command displays the data eyes as they are “seen” by a preprocessor with AUTOFOCUS technology resulting from the following commands:

- **Trace.AutoFocus**
- **Trace.TestFocus**

### Description of Buttons in the Trace.ShowFocus Window

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup …</td>
<td>Open <strong>Trace.state</strong> window to configure the trace.</td>
</tr>
<tr>
<td>Scan</td>
<td>Perform a <strong>Trace.TestFocus</strong> scan.</td>
</tr>
<tr>
<td>Scan+</td>
<td>Perform a <strong>Trace.TestFocus /Accumulate</strong> scan.</td>
</tr>
<tr>
<td>AutoFocus</td>
<td>Perform a <strong>Trace.AutoFocus</strong> scan.</td>
</tr>
<tr>
<td>Data</td>
<td>Open a <strong>Trace.ShowFocusEye</strong> window.</td>
</tr>
<tr>
<td>Clock</td>
<td>Open a <strong>Trace.ShowFocusClockEye</strong> window.</td>
</tr>
<tr>
<td>Store …</td>
<td>Save the current AUTOFOCUS configuration to a file (<strong>STOre &lt;file&gt; AnalyzerFocus</strong>).</td>
</tr>
<tr>
<td>Load …</td>
<td>Load an AUTOFOCUS configuration from a file (<strong>DO &lt;file&gt;</strong>).</td>
</tr>
<tr>
<td>![left arrow]</td>
<td>Move all sampling points 1 * <code>&lt;time_clock&gt;</code> to the left.</td>
</tr>
<tr>
<td>![right arrow]</td>
<td>Move all sampling points 1 * <code>&lt;time_clock&gt;</code> to the right.</td>
</tr>
</tbody>
</table>
Description of the Trace.ShowFocus Window

In the Trace.ShowFocus window the data eyes are white, whereas setup violations are marked as follows:

<table>
<thead>
<tr>
<th>Setup violation</th>
<th>Marking</th>
</tr>
</thead>
<tbody>
<tr>
<td>on the rising edge</td>
<td>High red line</td>
</tr>
<tr>
<td>on the falling edge</td>
<td>Low red line</td>
</tr>
<tr>
<td>on both edges</td>
<td>Grey bar</td>
</tr>
</tbody>
</table>

The x-axis of the Trace.ShowFocus window corresponds to the time axis, whereas the y-axis corresponds to the data channels of the ETM trace port. In the example above we have an 8-bit EMTv1.x architecture with the channels TRACESYNC (TS), PIPESTAT (PS[3:0]) and TRACEPKT (TP[7:0]).
A preprocessor with AUTOFOCUS technology has programmable delays for the clock channel as well as all data channels. With that in mind the x-axis (time-axis) has the following meaning:

<table>
<thead>
<tr>
<th>Data delay greater than clock delay</th>
<th>Negative value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both clock and data delay are zero</td>
<td>Zero</td>
</tr>
<tr>
<td>Clock delay greater than data delay</td>
<td>Positive value</td>
</tr>
</tbody>
</table>

In the example above there is a channel to channel skew of more than 1 ns that has been compensated by choosing individual sampling points for each data channel. The time resolution for clock and data channel adjustment is not necessarily the same. In the example the time resolution for data channel adjustment is relatively coarse (500-600 ps), whereas the clock channel can be adjusted in fine delay steps (78 ps). The actual values are functions of voltage, temperature and process. However they are measured for each Trace.AutoFocus or Trace.TestFocus execution, so the numbers displayed in the Trace.ShowFocus window do have a physical meaning (time unit is ns).

The example shows the Trace.ShowFocus window as it might appear when using the LA-7991 OTP (see Preprocessor for ARM-ETM AutoFocus for details). For the re-programmable version both clock and data delays are typically 270 ps and the Trace.ShowFocus window for the same application might look like this:

![Trace.ShowFocus window](image)

**NOTE:** The NEXUS AutoFocus adapter does not support this feature.

**See also**
- `<trace>.ShowFocusClockEye`
- `<trace>.AutoFocus`
- `<trace>.TestFocus`
- Trace
- `<trace>.TestFocusClockEye`

▲ 'Installation' in 'AutoFocus User's Guide'
▲ 'Installation' in 'ARM-ETM Trace'
The command `Trace.ShowFocusClockEye` displays a graph reflecting the clock waveform. Basically it shows data eyes from a different point of view.

The result of the command `Trace.ShowFocusClockEye` shows a single ETM channel or all ETM channels superimposed. Further are:

- **X-axis:** time range [ns]
- **Y-axis:** voltage range [V]

### Color Legend

<table>
<thead>
<tr>
<th>White area</th>
<th>Data eye.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green area</td>
<td>Setup violation on the rising edge.</td>
</tr>
<tr>
<td>Red area</td>
<td>Setup violation on the falling edge.</td>
</tr>
<tr>
<td>Superimposed area (green and red)</td>
<td>Setup violation on both edges.</td>
</tr>
</tbody>
</table>
### Description of Buttons in the `<trace>.ShowFocusClockEye` Window

<table>
<thead>
<tr>
<th>Setup …</th>
<th>Open <code>Trace.state</code> window to configure the trace.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>Perform a <code>Trace.TestFocusEye</code> scan.</td>
</tr>
<tr>
<td>Scan+</td>
<td>Perform a <code>Trace.TestFocusEye /Accumulate</code> scan.</td>
</tr>
<tr>
<td>AutoFocus</td>
<td>Perform a <code>Trace.AutoFocus</code>.</td>
</tr>
<tr>
<td>Digital</td>
<td>Open a <code>Trace.ShowFocus</code> window scan.</td>
</tr>
<tr>
<td>Channel (previous)</td>
<td>Display <code>Trace.ShowFocusClockEye</code> for a single trace line (previous).</td>
</tr>
<tr>
<td>Channel (next)</td>
<td>Display <code>Trace.ShowFocusClockEye</code> for a single trace line (next).</td>
</tr>
<tr>
<td>▼</td>
<td>Move all sampling points 1 * <code>&lt;time_clock&gt;</code> to the left.</td>
</tr>
<tr>
<td>▲</td>
<td>Move all sampling points 1 * <code>&lt;time_clock&gt;</code> to the right.</td>
</tr>
</tbody>
</table>

### Examples

```plaintext
Trace.ShowFocusEye ; Display data eye with
Trace.ShowFocusEye PS2 ; all trace channels superimposed

; Display data eye for the
; trace channel PS2
```

### NOTE:
The NEXUS AutoFocus adapter does not support this feature.

### See also
- `<trace>.ShowFocus`
- `<trace>.ShowFocusEye`
- `<trace>.AutoFocus`
- `<trace>.state`
- `<trace>.TestFocusEye`
- `Trace`
- `<trace>.TestFocus`
- `<trace>.TestFocusClockEye`

▲ 'Diagnosis' in 'AutoFocus User's Guide'
▲ 'Diagnosis' in 'ARM-ETM Trace'
The command `Trace.ShowFocusEye` displays the data eye as it is 'seen' by a preprocessor with AUTOFOCUS technology or PowerTrace Serial resulting from the command `Trace.TestFocusEye`.

The result of the command `Trace.ShowFocusEye` shows a single trace channel or all trace channels superimposed. The unit of the axis differs for AUTOFOCUS:

- **X-axis:** time range [ns] or [UI]
- **Y-axis:** voltage range [V] or [percentage]

and PowerTrace Serial technology:

- **X-axis:** time range [UI]
- **Y-axis:** voltage range [percentage of eye height]
### Color Legend for AUTOFOCUS

<table>
<thead>
<tr>
<th>Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>White area</td>
<td>Stable data.</td>
</tr>
<tr>
<td>Green area</td>
<td>Setup violation on the rising edge.</td>
</tr>
<tr>
<td>Red area</td>
<td>Setup violation on the falling edge.</td>
</tr>
<tr>
<td>Superimposed area</td>
<td>Setup violation on both edges.</td>
</tr>
<tr>
<td>(green and red)</td>
<td></td>
</tr>
</tbody>
</table>

### Color Legend for PowerTrace Serial

<table>
<thead>
<tr>
<th>Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>White area</td>
<td>Stable data eye.</td>
</tr>
<tr>
<td>Black areas</td>
<td>Unstable data.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>White area</td>
<td>Stable data eye.</td>
</tr>
<tr>
<td>Grey/Black areas</td>
<td>Unstable data.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark Blue area</td>
<td>Stable data eye.</td>
</tr>
<tr>
<td>Blue/Green/Orange/</td>
<td>Unstable data.</td>
</tr>
<tr>
<td>Yellow/Red areas</td>
<td></td>
</tr>
</tbody>
</table>
## Descriptions of Buttons in the `<trace>.ShowFocusEye Window:

<table>
<thead>
<tr>
<th>Button</th>
<th>Action Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup …</td>
<td>Open <code>Trace.state</code> window to configure the trace.</td>
</tr>
<tr>
<td>Scan</td>
<td>Perform a <code>Trace.TestFocusEye</code> scan.</td>
</tr>
<tr>
<td>Scan+</td>
<td>Perform a <code>Trace.TestFocusEye /Accumulate</code> scan.</td>
</tr>
<tr>
<td>AutoFocus</td>
<td>Perform a <code>Trace.AutoFocus</code> scan.</td>
</tr>
<tr>
<td>Digital</td>
<td>Open a <code>Trace.ShowFocus</code> window.</td>
</tr>
<tr>
<td>Channel (previous)</td>
<td>Display <code>Trace.ShowFocusEye</code> for a single trace line (previous).</td>
</tr>
<tr>
<td>Channel (next)</td>
<td>Display <code>Trace.ShowFocusEye</code> for a single trace line (next).</td>
</tr>
<tr>
<td></td>
<td>Move all sampling points 1 * <code>&lt;time_clock&gt;</code> to the left.</td>
</tr>
<tr>
<td></td>
<td>Move all sampling points 1 * <code>&lt;time_clock&gt;</code> to the right.</td>
</tr>
</tbody>
</table>

### Examples

```plaintext
Trace.ShowFocusEye ; Display data eye with
                   ; all trace channels superimposed
Trace.ShowFocusEye PS2 ; Display data eye for the
                        ; trace channel PS2
```

### NOTE:

The NEXUS AutoFocus preprocessor does not support this feature.

### See also
- `<trace>.ShowFocus`
- `<trace>.AutoFocus`
- `<trace>.TestFocusEye`
- `<trace>.TestFocus`
Sets the <size> of trace memory which is used for trace recording. If the command is called with size zero, the trace size will be set to its maximum size.

Reducing the size used for trace recording helps to reduce time needed for trace data download and trace analysis (statistical analysis, trace chart display etc, searching for an event in the trace), because of the smaller amount of recorded data. There is no other benefit besides that.

The maximum trace size for the Advanced Register Trace (ART) and ICE/FIRE port and state analyzers is fixed:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ART</td>
<td>65535.</td>
</tr>
<tr>
<td>ICE State Analyzer</td>
<td>32767.</td>
</tr>
<tr>
<td>ICE Port Analyzer</td>
<td></td>
</tr>
<tr>
<td>FIRE State Analyzer</td>
<td>65535.</td>
</tr>
<tr>
<td>FIRE Port Analyzer</td>
<td></td>
</tr>
<tr>
<td>FIRE State Analyzer</td>
<td>524288.</td>
</tr>
<tr>
<td>512K</td>
<td></td>
</tr>
<tr>
<td>FIRE Port Analyzer</td>
<td></td>
</tr>
</tbody>
</table>

The maximum trace size for logic analyzers PowerProbe and PowerIntegrator depends on module memory size and sample frequency:

<table>
<thead>
<tr>
<th>Module / Mode</th>
<th>&lt;= 100MHz</th>
<th>200MHz</th>
<th>400MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerProbe 128K (LA-7930)</td>
<td>131071.</td>
<td>262142.</td>
<td>524284.</td>
</tr>
<tr>
<td>PowerProbe 256k (LA-7931)</td>
<td>262143.</td>
<td>524286.</td>
<td>1048572.</td>
</tr>
<tr>
<td>PowerIntegrator (LA-7940)</td>
<td>524287.</td>
<td>1048576.</td>
<td>2097152.</td>
</tr>
</tbody>
</table>
The maximum trace size for trace modules PowerTrace and RiscTrace depends on the module’s memory size and a factor depending on the trace protocol:

<table>
<thead>
<tr>
<th>Module</th>
<th>Number of trace records</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICD RiscTrace (LA-7859, LA-7870)</td>
<td>65536</td>
</tr>
<tr>
<td>PowerTrace 256MB (LA-7707)</td>
<td>16777216</td>
</tr>
<tr>
<td>PowerTrace 512MB (LA-7690)</td>
<td>33554432</td>
</tr>
<tr>
<td>PowerTrace II 1G (LA-7692)</td>
<td>67108864</td>
</tr>
<tr>
<td>PowerTrace II 2G (LA-7693)</td>
<td>134217728</td>
</tr>
<tr>
<td>PowerTrace II 4G (LA-7694)</td>
<td>268435456</td>
</tr>
</tbody>
</table>

Trace sizes for several trace protocols:

<table>
<thead>
<tr>
<th>Trace protocol</th>
<th>Factor (size = factor * records)</th>
<th>Trace size PowerTrace 512MB</th>
<th>Trace size PowerTrace II 4G</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETM</td>
<td>4.</td>
<td>134217728.</td>
<td>1073741824.</td>
</tr>
<tr>
<td>NEXUS</td>
<td>1.</td>
<td>33554432.</td>
<td>268435456.</td>
</tr>
</tbody>
</table>

See also
- `<trace>.state`
- `<trace>`
- `Analyzer.SIZE()`
- `Analyzer.RECORDS()`
- ‘Emulator Functions’ in ‘FIRE User’s Guide’
- ‘Release Information’ in ‘Release History’

**<trace>.SLAVE**

Select slave mode

Format: `<trace>.SLAVE`

The trace memory of the port analyzer is used as extension to the SA120 or HA120 state analyzer unit. All other setups of the port analyzer are not valid.
<trace>.SnapShot

Format:  
<trace>.SnapShot
<trace>.TEST (deprecated)

Restart the trace capturing. Effectively the same as executing the commands **Trace.OFF, Trace.Init** and **Trace.Arm**.

Most often used to restart the trace recording:

- After the trace capturing was stopped by a trigger (e.g. by a TraceTrigger breakpoint).
- When the trace works in **Stack** mode and trace capturing was stopped because the trace buffer was full.

See also

- <trace>.SelfArm
- <trace>.state
- Trace

<trace>.SPY

Adaptive stream and analysis

Format:  
<trace>.SPY

**Trace.SPY** allows display intermediate trace analysis results while streaming (see **<trace>.Mode STREAM**). If the average data rate at the trace port is high, the analysis time is reduced, and vise versa.

The **Trace.SPY** command required that trace decoding is possible while the program execution is running. This is possible, if the core architecture in use provides run-time access to memory or if the object code is loaded to the TRACE32 virtual memory. It is recommended to load the object code to the TRACE32 Virtual Memory in any case, because the trace analysis is then faster.

The **Trace.SPY** command only works if the trace is currently in Arm mode.

Example:

```
...
Data.LOAD.Elf demo.elf /VM ; copy object code to TRACE32
; Virtual Memory
Analyzer.Mode STREAM ; select trace mode STREAM
Go ; start the program execution
...
```
IF Analyzer.STATE()! = 1
    PRINT "No switch to SPY mode possible"
Trace.SPY ; enable analysis of streaming file
Trace.List
Trace.Arm ; switch back to standard recording
...

See also
- Trace
<trace>.state

Display the trace configuration window. The trace methods are displayed at the top of the window. The configuration options below the trace methods adjust to the currently selected trace method, compare screenshot on the left with the screenshot on the right.

A After you have selected the desired trace method (Trace.METHOD), you can work with the commands that start with Trace. This principle is illustrated in the two PRACTICE script snippets below.

B For descriptions of the commands in the Trace.state window, please refer to the <trace>* commands in this chapter. Example: For information about OFF, see <trace>.OFF.

```
Trace.METHOD Analyzer ;Select the trace method, here Analyzer
<your_code>
Trace.List ;The trace listing now refers to the ;method Analyzer
<your_code>

Trace.METHOD SNOOPer ;Select the trace method, here SNOOPer
<your_code>
Trace.List ;The trace listing now refers to the method SNOOPer
<your_code>
```

©1989-2020 Lauterbach GmbH
Using this command the operating mode of the analyzer may be selected. During operation this command displays the current state of the analyzer.

**State Field**

Displays the current port analyzer state.

- **SLAVE**
  - The analyzer is in slave mode to the SA120/HA120 unit.

- **OFF**
  - Indicates that trace memory is deactivated and no trace is done. Memory can be read out.

- **ARM**
  - Indicates that the analyzer is activated (for sampling and triggering events). It cannot be read out.

- **trigger**
  - Indicates that the specified trigger event has been encountered.

- **break**
  - Indicates that the specified trigger event has been encountered and trigger delay time has elapsed. At the same time, the analyzer is set to OFF mode.

**used**

Displays the current recording depth in graphical format. Recording depth display is also updated during analyzer activation, but only in steps of 512 entries. If the sampling is stopped, the correct value will be displayed.
Ports

Displays continuously the state of the ports. The order is physically (logically depending on the emulated microcontroller) just as the order, used in the list command.

With a mouse click to the corresponding area all commands can be executed or selects can be switched on or off.

See also

- <trace>.ACCESS
- <trace>.Arm
- <trace>.AutoFocus
- <trace>.AutoStart
- <trace>.BookMark
- <trace>.ComPare
- <trace>.CustomTraceLoad
- <trace>.DisConfig
- <trace>.EXPORT
- <trace>.Find
- <trace>.FindChange
- <trace>.FLOWSTART
- <trace>.GOTO
- <trace>.List
- <trace>.ListVar
- <trace>.Mode
- <trace>.PC
- <trace>.PortType
- <trace>.PROfileChart
- <trace>.Program
- <trace>.REF
- <trace>.RESet
- <trace>.SAVE
- <trace>.ShowFocus
- <trace>.ShowFocusEye
- <trace>.SnapShot
- <trace>.TDelay
- <trace>.TestFocus
- <trace>.THreshold
- <trace>.TraceCONNECT
- <trace>.View
- <trace>.ZERO
- Trace.METHOD
- Trace.RECORDS()

▲ 'Trace Functions' in 'General Function Reference'
▲ 'Displaying the Trace' in 'Training FIRE Analyzer'
▲ 'HCS12 On-chip Debug Module' in 'FIRE Emulator for HC12/MCS12'
▲ 'Emulator Functions' in 'FIRE User's Guide'
▲ 'Release Information' in 'Release History'
The `<trace>.STATistic` commands can be used for statistical analysis based on the information sampled to the trace buffer.

In contrast to the performance analyzer (PERF commands), the statistical analysis commands provide a higher precision and much more information about the analyzed item, but since the size of the trace buffer is limited, the observation time is limited. Statistic evaluations can be made after the trace memory stops sampling. For short calculation time minimize the trace buffer size.

See also

- `<trace>.STATistic.Address`
- `<trace>.STATistic.AddressDIStance`
- `<trace>.STATistic.AddressDURation`
- `<trace>.STATistic.BondOut`
- `<trace>.STATistic.CHildTREE`
- `<trace>.STATistic.CYcle`
- `<trace>.STATistic.DISTance`
- `<trace>.STATistic.DURation`
- `<trace>.STATistic.FIRST`
- `<trace>.STATistic.Func`
- `<trace>.STATistic.FuncDURation`
- `<trace>.STATistic.FuncDURationInternal`
- `<trace>.STATistic.GROUP`
- `<trace>.STATistic.Ignore`
- `<trace>.STATistic.InterruptIsFunction`
- `<trace>.STATistic.LAST`
- `<trace>.STATistic.LINEage`
- `<trace>.STATistic.MODULE`
- `<trace>.STATistic.ParentTREE`
- `<trace>.STATistic.PROGRAM`
- `<trace>.STATistic.Sort`
- `<trace>.STATistic.TASK`
- `<trace>.STATistic.TASKINTR`
- `<trace>.STATistic.TASKORINTERRUPT`
- `<trace>.STATistic.TASKState`
- `<trace>.STATistic.TASKVSINTR`
- `<trace>.STATistic.TASKVSINTR`
- `<trace>.STATistic.TREE`
- `<trace>.STATistic.Use`
- Analyzer.RECORDS()
Format: `trace>.STATistic.Address <address1> [<address1> ...]`

Displays the time interval between up to 8 program events.

Examples:

```
Trace.STATistic.Address sieve response buffer
Trace.STATistic.Address 0x125c 0x1264 0x1274 0x1290 0x12ac 0x12b8 0x12d8
```

Analysis background:

```
address1
address2
address2
address3
address1
address1
address2
```

**See also**
- `<trace>.STATistic`
- 'Release Information' in 'Release History'
**<trace>.STATistic.AddressDISTance**  
Time interval for single program event

Displays the time interval for a single program event. Without parameter the assignment of classes (16) is done automatically. With arguments the classes can be set up manually.

The following 2 commands are equivalents:

```text
B::Trace.STATistic.AddressDISTance InterruptEntry
B::Trace.STATistic.DISTance /FILTER Address InterruptEntry
```

If selective tracing is possible, use the `/TraceEnable` filter to extend the observation time:

```text
B::Break.Set InterruptEntry /Program /TraceEnable
B::Go
B::Break
B::Trace.STATistic.AddressDISTance InterruptEntry
```

In the case of an SMP system, the following options are provided:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JoinCORE</td>
<td>Analysis is performed for all cores. The core information is discarded.</td>
</tr>
<tr>
<td>SplitCORE</td>
<td>Same as JoinCORE.</td>
</tr>
<tr>
<td>MergeCORE</td>
<td>Same as JoinCORE.</td>
</tr>
<tr>
<td>CORE &lt;number&gt;</td>
<td>Analysis is performed for the specified core.</td>
</tr>
</tbody>
</table>

See also

- `<trace>.STATistic`
- `<trace>.STATisticDistance`

▲ 'Release Information' in 'Release History'
The statistic distribution between two program events is analyzed. This command can be used to analyze the run-time of a single function or interrupt response times.

By default TRACE32 PowerView builds 16 result classes. For a graphical display of the results, use the command `Trace.PROfileChart.DURation.43`
The `<option>` **Number** allows a user-defined number of result classes.

```
Trace.STATistic.AddressDURation func9 sYmbol.EXIT(func9) /Number 6.
```

The parameter `<timemin>` allows to specify the time for the first result class, the parameter `<increment>` allows to specify the increment for the next result class.

```
Trace.STATistic.AddressDURation func9 sYmbol.EXIT(func9) 15.us 1.us
```

Trace filter allow a more effective usage of the trace memory:

```
Trace.Mode Leash

Break.Set func9 /Program /TraceEnable

Break.Set sYmbol.EXIT(func9) /Program /TraceEnable

Go

WAIT !STATE.RUN()

Trace.STATistic.AddressDURation func9 sYmbol.EXIT(func9)
```

In the case of an SMP system, the following options are provided:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JoinCORE</strong></td>
<td>Analysis is performed for all cores. The core information is discarded.</td>
</tr>
<tr>
<td>(default)</td>
<td></td>
</tr>
<tr>
<td><strong>SplitCORE</strong></td>
<td>Same as JoinCORE.</td>
</tr>
<tr>
<td><strong>MergeCORE</strong></td>
<td>Same as JoinCORE.</td>
</tr>
<tr>
<td><strong>CORE &lt;number&gt;</strong></td>
<td>Analysis is performed for the specified core.</td>
</tr>
</tbody>
</table>

**See also**

- `<trace>.STATistic`
- 'Release Information' in 'Release History'
<trace>.STATistic.BondOut

**Bondout mode**

If **ON** is selected, the software will use the bondout busses to capture the function entry and exit data. This command is only available on some emulation targets.

**See also**
- <trace>.STATistic

<trace>.STATistic.ChildTREE

**Show callee context of a function**

**Format:**

```
<trace>.STATistic.ChildTREE <address>
```

Show call tree and run-time of all functions called by the specified function. The function is specified by its start **<address>**.

**Example:**

```
Trace.STATistic.ChildTREE master_selection
```

```
B:Trace.STATistic.ChildTREE master_selection
```

**See also**
- <trace>.STATistic
- <trace>.STATistic.ParentTREE

▲ 'Release Information' in 'Release History'

©1989-2020 Lauterbach GmbH
### <trace>.STATistic.COLOR

Assign colors to function for colored graphics

**Format:**

```
<trace>.STATistic.COLOR FixedColors | AlternatingColors
```

<table>
<thead>
<tr>
<th><strong>FixedColors</strong>&lt;br&gt;(default)</th>
<th>Colors are assigned fixed to functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AlternatingColors</strong></td>
<td>Colors are assigned by the recording order of the functions for each measurement.</td>
</tr>
</tbody>
</table>

**See also**
- <trace>.STATistic
- 'PowerView - Screen Display' in 'PowerView User's Guide'

### <trace>.STATistic.CYcle

Analyze cycle types

**Format:**

```
<trace>.STATistic.CYcle [<timerange>] [<option>]
```

**<option>:**
- FILE | Accumulate
- INCremental | FULL
- IdleThreshold `<clocks>`
- CORE `<number>` | SplitCORE | MergeCORE | JoinCORE (SMP tracing only)

Performs a statistical analysis of the cycle types.

In the case of an SMP system, the following options are provided:

| **MergeCORE**<br>(default) | Analysis is performed independently for each core. The results are summarized and displayed as a single result. |
| **SplitCORE**              | Same as MergeCORE. |
| **JoinCORE**               | Same as MergeCORE. |
| **CORE `<number>`**        | Analysis is performed for the specified core. |
Example based on CoreSight ETMv3 for a Cortex-R4:

```
ETM.DataTrace ON       ; full data trace
ETM.CycleAccurate ON   ; cycle accurate tracing
Trace.CLOCK 450.MHz    ; inform TRACE32 about the core clock
```

### survey

<table>
<thead>
<tr>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>records</td>
<td>Number of records in the trace</td>
</tr>
<tr>
<td>time</td>
<td>Time period recorded by the trace</td>
</tr>
<tr>
<td>clocks</td>
<td>Number of clock cycles recorded by the trace</td>
</tr>
<tr>
<td></td>
<td>(cycle-accurate tracing only)</td>
</tr>
<tr>
<td>flow cycles</td>
<td>Number of flow cycles</td>
</tr>
<tr>
<td></td>
<td>(flow execute + flow read + flow write)</td>
</tr>
<tr>
<td>bus cycles</td>
<td>Number of data read and data write cycles</td>
</tr>
<tr>
<td>cpi</td>
<td>Average clocks per instruction</td>
</tr>
<tr>
<td></td>
<td>(clocks/instr)</td>
</tr>
<tr>
<td></td>
<td>(cycle-accurate tracing only)</td>
</tr>
<tr>
<td>details</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>flow execute</td>
<td>Number of cycles that executed/not executed instructions</td>
</tr>
<tr>
<td>flow read</td>
<td>Number of cycles that performed a read access</td>
</tr>
<tr>
<td>flow write</td>
<td>Number of cycles the performed a write access</td>
</tr>
<tr>
<td>bus fetch</td>
<td>—</td>
</tr>
<tr>
<td>bus read</td>
<td>Number of data read cycles</td>
</tr>
<tr>
<td>bus write</td>
<td>Number of data write cycles</td>
</tr>
<tr>
<td>instr</td>
<td>Number of executed/not executed instruction</td>
</tr>
<tr>
<td>slot instr</td>
<td>—</td>
</tr>
<tr>
<td>fail cond</td>
<td>Number of conditional instructions that failed (failed branch instructions not included)</td>
</tr>
<tr>
<td>pass cond</td>
<td>Number of conditional instructions that passed (taken branches not included)</td>
</tr>
<tr>
<td>fail branch</td>
<td>Number of failed branch instructions</td>
</tr>
<tr>
<td>dir branch</td>
<td>Number of direct branches</td>
</tr>
<tr>
<td>indir branch</td>
<td>Number of indirect branches</td>
</tr>
<tr>
<td>load instr</td>
<td>Number of load instructions</td>
</tr>
<tr>
<td>store instr</td>
<td>Number of store instructions</td>
</tr>
<tr>
<td>modify instr</td>
<td>Number of modify instructions (swp, swpb)</td>
</tr>
<tr>
<td>traps</td>
<td>Number of traps</td>
</tr>
<tr>
<td>interrupts</td>
<td>Number of interrupts</td>
</tr>
<tr>
<td>idles</td>
<td>Number of “wait for interrupt” (coprocessor instruction or WFI instruction) or number of times that 1000. clock cycles passed without a broadcast of trace information.</td>
</tr>
<tr>
<td></td>
<td>The option <strong>IdleThreshold</strong> allows to modify the number of clock cycles that need to pass for a idle detection.</td>
</tr>
<tr>
<td>trace gaps</td>
<td>Number of trace gaps (<strong>FIFOFULL</strong>, filtered trace information …)</td>
</tr>
</tbody>
</table>
<trace>.STATistic.DatasYmbol  Analyze pointer contents numerically

Format:  

<trace>.STATistic.DatasYmbol [<list_item> …] [%<format>] [/<option>]

<list_item>:  
DEFault | ALL
Total | MIN | MAX | AVeRage
Count | Ratio | BAR
CountRatio | CountBar

<format>:  
DEFault | LEN
TimeAuto | TimeFixed

<option>:  
FILE
FlowTrace | BusTrace
BEFORE | AFTER | ALL
Accumulate
INCremental | FULL
Sort <item>
Filter <item>
Track

The command Trace.STATistic.DatasYmbol analyzes the contents of a pointer numerically.

If a full program and data trace is analyzed, the following command is recommended:

; analyze the contents of the pointer vpchar numerically
Trace.STATistic.DatasYmbol /Filter Address vpchar
A more effective usage of the trace memory is possible, if only write accesses to the pointer are recorded in the trace.

```plaintext
; set a filter to record only write cycles to the pointer vpchar to the
; trace
Var.Break.Set vpchar /Write /TraceEnable

...

; analyze the contents of the pointer
Trace.STATistic.Datasymbol

; analyze the contents of the pointer, sort the result by symbol names
Trace.STATistic.Datasymbol /Sort symbol
```

See also

- `<trace>.STATistic`
- 'Release Information' in 'Release History'
<trace>.STATistic.DIStance

Time interval for a single event

Displays the time interval for a single event. Without parameter the assignment of classes (16) is done automatically. With arguments the classes can be set up manually.

```plaintext
<trace>.STATistic.DIStance [<timemin>] [<increment>] [/<option>]
```

- **FILE**
- **FlowTrace** | **BusTrace**
- **Accumulate**
- **INCremental** | **FULL**
- **Sort** `<item>`
- **Track**
- **NoMerge**
- **Filter** `<item>`
- **Number** `<number>`
- **LOG**
- **LINear**

<table>
<thead>
<tr>
<th>samples: 73.</th>
<th>avr: 39.636us</th>
<th>min: 0.500us</th>
<th>max: 461.125us</th>
</tr>
</thead>
<tbody>
<tr>
<td>up to</td>
<td>count</td>
<td>ratio</td>
<td>1%</td>
</tr>
<tr>
<td>&lt; 0.000</td>
<td>0</td>
<td>0.000%</td>
<td></td>
</tr>
<tr>
<td>40.960us</td>
<td>55</td>
<td>75.342%</td>
<td></td>
</tr>
<tr>
<td>81.920us</td>
<td>10</td>
<td>13.698%</td>
<td></td>
</tr>
<tr>
<td>122.880us</td>
<td>1</td>
<td>1.369%</td>
<td></td>
</tr>
<tr>
<td>163.840us</td>
<td>2</td>
<td>2.739%</td>
<td></td>
</tr>
<tr>
<td>204.800us</td>
<td>2</td>
<td>2.739%</td>
<td></td>
</tr>
<tr>
<td>245.760us</td>
<td>0</td>
<td>0.000%</td>
<td></td>
</tr>
<tr>
<td>286.720us</td>
<td>1</td>
<td>1.369%</td>
<td></td>
</tr>
<tr>
<td>327.680us</td>
<td>0</td>
<td>0.000%</td>
<td></td>
</tr>
</tbody>
</table>

**Basic <options>**

- **FILE**
  - Displays trace memory contents loaded with `Trace.FILE`.

- **FlowTrace**
  - The trace works as flow trace. This option is usually not required.

- **BusTrace**
  - The trace works as a bus trace. This option is usually not required.

- **Accumulate**
  - By default only the current trace contents is analyzed by the statistic functions. The option `/Accumulate` allows to add the current trace contents to the already displayed results.

- **INCremental**
  - Intermediate results are displayed while the TRACE32 software analyses the trace contents (default).
**Example for TRACE32-ICD and TRACE32-PowerTrace:**

If no selective tracing is possible, use the option `/Filter` to filter out the event of interest.

```
B::Go
B::Break
B::Trace.STATistic.DISTance /FILTER Address InterruptEntry
```

If selective tracing is possible, use the `/TraceEnable` filter to extend the observation time:

```
B::Break.Set InterruptEntry /Program /TraceEnable
B::Go
B::Break
B::Trace.STATistic.DISTance /FILTER Address InterruptEntry
```
Example for TRACE32-ICE or TRACE32-FIRE:

Perform a selective trace on the event of interest

```
Analyzer.ReProgram ; selective trace
(
    ADDR AlphaBreak InterruptEntry ; program the analyzer to sample
    Sample.enable IF AlphaBreak , all InterruptEntry
)
Go
...
Break
Trace.STATistic.DISTance ; measurement
; analyze the time interval
; between the sampled
; InterruptEntry
```

See also

- `<trace>.STATistic.AddressDISTance`
- `<trace>.STATistic`
- 'Release Information' in 'Release History'
- 'Statistic Functions' in 'Training FIRE Analyzer'
- 'Statistic Functions' in 'Training ICE Analyzer'
Distribution analysis

The statistic distribution of any data is displayed if `<item>` is specified. Displayed are the number of occurrences and the time after the events, i.e. the time an event is assumed to be valid. Without `<item>` the statistic is based on the symbolic addresses.

### Format:

```
<trace>.STATistic.DistriB [%<format>] [<items> …] [/<option>]
```

### `<options>`:

- **FILE**
- **FlowTrace** | **BusTrace**
- **Accumulate**
- **INCremental** | **FULL**
- **Sort `<item>`**
- **Track**
- **NoMerge**
- **BEFORE** | **AFTER**
- **List [ `<list_item>` …]**
- **Filter `<item>`**

### Basic `<options>` Description

<table>
<thead>
<tr>
<th>Basic <code>&lt;options&gt;</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE</strong></td>
<td>Displays trace memory contents loaded with <code>Trace.FILE</code>.</td>
</tr>
<tr>
<td><strong>FlowTrace</strong></td>
<td>The trace works as flow trace. This option is usually not required.</td>
</tr>
<tr>
<td><strong>BusTrace</strong></td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td><strong>Accumulate</strong></td>
<td>By default only the current trace contents is analyzed by the statistic functions. The option <code>/Accumulate</code> allows to add the current trace contents to the already displayed results.</td>
</tr>
<tr>
<td><strong>INCremental</strong></td>
<td>Intermediate results are displayed while the TRACE32 software analyses the trace contents (default).</td>
</tr>
<tr>
<td><strong>FULL</strong></td>
<td>The result is displayed after the TRACE32 software finished the analysis.</td>
</tr>
<tr>
<td><strong>Track</strong></td>
<td>Track the <code>Trace.STATistic</code> window with other trace list windows (tracking to record number or time possible).</td>
</tr>
<tr>
<td><strong>NoMerge</strong></td>
<td>(For diagnosis purpose only).</td>
</tr>
</tbody>
</table>
The **List** option defines which values are calculated and displayed. The `<list_items>` can be arranged by pushing the **Config** button in the **Trace.STATistic.Distrib** window.

<table>
<thead>
<tr>
<th><code>&lt;list_item&gt;</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>Total time the event was true.</td>
</tr>
<tr>
<td><strong>MIN, MAX</strong></td>
<td>Minimum and maximum time the event was true.</td>
</tr>
<tr>
<td><strong>AVerage</strong></td>
<td>Average time the event was true.</td>
</tr>
<tr>
<td><strong>Count</strong></td>
<td>Number of occurrences of the event.</td>
</tr>
<tr>
<td><strong>Ratio, BAR.log, BAR.LINEar</strong></td>
<td>Ratio of time spent in events to total measurement time in percent and as graphical bars.</td>
</tr>
<tr>
<td><strong>CRatio, CBAR.log, CBAR.LINEar</strong></td>
<td>Ratio of count to total count in percent and as graphical bars.</td>
</tr>
</tbody>
</table>

All time displays depend on the options **AFTER** or **BEFORE**.
Example for TRACE32-ICD and TRACE32-PowerTrace:

If no selective tracing is possible, use the option /Filter to filter out the event of interest.

```
B::Go
B::Break
B::Trace.STATistic.DistriB Data.B /Filter Address V.RANGE(flags[3])
```

Example for TRACE32-ICE or TRACE32-FIRE:

Perform a selective trace on the event of interest.

```
Analyzer.ReProgram
(
  ADDR AlphaBreak V.RANGE(flags[3])
  Sample.enable IF AlphaBreak
)
Go
...
Break
Trace.Chart.DistriB Data.B
```

; selective trace
; program the analyzer to
; sample all accesses to the
; variable flags[3]

; measurement

; display the time spent in
; different states

```
<table>
<thead>
<tr>
<th>class</th>
<th>samples: 620.</th>
<th>total: 9.456ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>min</td>
</tr>
<tr>
<td>d.b=0x0</td>
<td>6.583ms</td>
<td>8.975us</td>
</tr>
<tr>
<td>d.b=0x1</td>
<td>2.854ms</td>
<td>6.550us</td>
</tr>
</tbody>
</table>
```

See also
- <trace>.STATistic
- ‘Release Information’ in ‘Release History’
### Format

```
<trace>.STATistic.DURation [<timemin>] [<increment>] [/<option>]
```

### Option:

<table>
<thead>
<tr>
<th>Option</th>
<th>Basic Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Displays trace memory contents loaded with Trace.FILE.</td>
</tr>
<tr>
<td>FlowTrace</td>
<td>The trace works as flow trace. This option is usually not required.</td>
</tr>
<tr>
<td>BusTrace</td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td>Accumulate</td>
<td>By default only the current trace contents is analyzed by the statistic functions. The option /Accumulate allows to add the current trace contents to the already displayed results.</td>
</tr>
<tr>
<td>INCremental</td>
<td>Intermediate results are displayed while the TRACE32 software analyzes the trace contents (default).</td>
</tr>
<tr>
<td>FULL</td>
<td>The result is displayed after the TRACE32 software finished the analysis.</td>
</tr>
<tr>
<td>Number</td>
<td>Define the number of classes.</td>
</tr>
<tr>
<td>LOG</td>
<td>Display the bars in the result display in a logarithmic format (default).</td>
</tr>
<tr>
<td>LINear</td>
<td>Display the bars in the result display in a linear format.</td>
</tr>
</tbody>
</table>

Analyzes the statistic distribution between two events.
Example for TRACE32-PowerTrace

To determine the time interval between two instructions (addresses) `Trace.STATistic.AddressDURation` is more suitable.

This example analyzes how long it takes when the contents of a variable changes from 0x0 to 0x1.

```
Var.Break.Set flags /Write /TraceEnable
Trace.STATistic.DURation /FilterA Data 0x0 /FilterB Data 0x1
```

In order to use the command `Trace.STATistic.DURation`:

- Check if both events are exported by a trace packet. Information reconstructed by TRACE32 is not analyzed.
- Alternatively use a `TraceEnable` breakpoint export the event as a trace packet.

The options `FilterA` and `FilterB` provide you with the means to describe your event.

- **FilterA <item>** Specify the first event.
- **FilterB <item>** Specify the second event.
Analyzer.ReProgram

(  
ADDR AlphaBreak InterruptEntry
ADDR BetaBreak InterruptExit
Sample.enable IF AlphaBreak || BetaBreak
Mark.A IF AlphaBreak
Mark.B IF BetaBreak
)

E::Go
E::Break

E::Trace.STATistic.DURation

E::Trace.STATistic.DURation 40.us 40.us

By default the time interval between the first event and the second event is displayed (ATO-B). This can be changed by the following options:

**ATO-A** Display the time interval from A to A.
**BTO-A** Display the time interval from B to A.
**BTO-B** Display the time interval from B to B.
...

See also
- `<trace>.STATistic`
- ‘Statistic Functions’ in ‘Training FIRE Analyzer’
- ‘Statistic Functions’ in ‘Training ICE Analyzer’
The `Trace.STATistic` commands analyze the complete trace contents by default. The command `Trace.STATistic.FIRST` allows to freely select a start point for the statistic analysis.

**Example for `<value>`:**

```plaintext
Trace.List ; display trace listing
Trace.STATistic.FIRST -123366. ; select trace record -123366. ; as start point for the trace ; analysis
Trace.STATistic.LAST -36675. ; select trace record -36675. ; as end point for the trace ; analysis
Trace.STATistic.Func ; perform a function run-time ; analysis
```
Example for `<time>`:

```
Trace.List TIME.Zero DEFAULT ; display trace listing
Trace.STATistic.FIRST 0.3us ; select trace record with time
                             ; stamp 0.3 µs (zero time)
                             ; as start point for the trace
                             ; analysis
Trace.STATistic.Func ; perform a function run-time
                             ; analysis between the specified
                             ; start point and the end of the
                             ; trace buffer
```

See also
- `<trace>.STATistic`
- `<trace>.STATistic.LAST`
- 'Release Information' in 'Release History'
### Nesting function runtime analysis

**Format:**

```
<trace>.STATistic.Func [%<format>] [<list_items> ...] [/<option>]
<trace>.STATistic.TASKFunc
```

(as an alias if a TRACE32 OS Awareness is used)

**<format>:**

- DEFault
- LEN
- TimeAuto
- TimeFixed

**<list_item>:**

- DEFault
- ALL
- Total
- MIN
- MAX
- AVerage
- Count
- NAME
- TASK
- Internal
- IAVerage
- IMIN
- IMAX
- InternalRatio
- InternalBAR
- External
- EAVerage
- EMIN
- EMAX
- INTR
- INTRMAX
- INTRCount
- ExternalTASK
- ExternalTASKMAX
- TASKCount
- TotalRatio
- TotalBAR

**<option>:**

- FILE
- FlowTrace
- BusTrace
- Accumulate
- INCremental
- FULL
- Sort <item>
- Track
- NoMerge
- IncludeOwn
- IncludeTASK
- IncludeINTR
- INTRROOT
- INTRTASK
- TASK <task>
- TASK !<task>

**<task>:**

- <task_magic>
- <task_id>
- <task_name>

Analyzes the function nesting and calculates the time spent in functions and the number of function calls.
Considerations

Please be aware that any gap in the trace recording (FIFOFULL) might result in an incorrect analysis results.

The trace can be tested for FIFOFULLs as follows:

; Process the complete trace contents
Trace.FLOWPROCESS

IF A.FLOW.FIFOFULL() != 0
    PRINT "Trace.STATistic.Func not possible due to FIFOFULL errors."

If it is not possible to eliminate the FIFOFULLs, it is recommended to use the command Trace.STATistic.sYmbol.
In order to prepare the results for the command **Trace.STATistic.Func**, TRACE32 postprocesses the program flow recorded by the PowerTrace to find:

- **Function entries**
  The execution of the first instruction of an HLL function is regarded as function entry.
  Additional identifications for function entries are implemented depending on the processor architecture and the used compiler.

- **Function exits**
  A RETURN instruction within an HLL function is regarded as function exit.
  Additional identifications for function exits are implemented depending on the processor architecture and the used compiler.

- **Entries to interrupt service routines (asynchronous)**
  If an interrupt was identified, the following entry to an HLL function is regarded as entry to the interrupt service routine.
  Interrupts are identified as follows:
  - The trace port broadcasts the occurrence of an interrupt (e.g. PPC4xx).
  - An entry to the vector table is detected and the vector address indicates an asynchronous/hardware interrupt (e.g. ARM9).
  - If the vector table base address is configurable the usage of the command **SYStem.Option VECTORS** might be necessary (e.g. MPC5xxx).

  If an interrupt is detected in the trace, it is marked as in the screenshot below.

- **Exits of interrupt service routines**
  A RETURN / RETURN FROM INTERRUPT within the HLL interrupt service routine is regarded as exit of the interrupt service routine.
• Entries to TRAP handlers (synchronous)

If an entry to the vector table was identified and if the vector address indicates a synchronous interrupt/trap the following entry to an HLL function is regarded as entry to the trap handler.

If a TRAP is detected in the trace, it is marked as in the screenshot below.

• Exits of TRAP handlers

A RETURN / RETURN FROM INTERRUPT within the HLL trap handler is regarded as exit of the trap handler.
Interpretation of the Result

Number of analyzed functions: 1390
Total measurement time: 20.586s
Total time in interrupt service routines over the total measurement time: 59.741ms
Some additional explanations with regards to the function name (column range):

- **(root)**: is the root of the analyzed function nesting.
- **HLL interrupt service routines**: HLL interrupt service routines are indicated in the analysis as shown below:
  
  ```
  \where\|\text{ums\_bute\_build\|intr\_os\_wrapper\|intr\_os\_prologue60}
  ```

- **HLL trap handler**: HLL trap handler are indicated in the analysis as shown below:
  
  ```
  \|\_\text{ArmVectorSwi}
  ```

If **Trace.STATistic.TASKFunc** was performed instead of **Trace.STATistic.Func**, because TRACE32 detected an RTOS, the following function names will appear:

- **<function>@<task_name>**: The name of the task in which the function is called is appended to the function name.
  
  ```
  \|\text{rom\|Div64\|UDiv64\|Timer\_Task}
  ```

- **(root)@<task_name>**: is the root of the analyzed function nesting for the task `<task_name>`.
- **(root)@(root)**: program section where no task-assignment is possible (e.g. measurement started within a task) are summarized here.
The following description of the `<list_item>` that provide the analysis results is kept quite general. An accurate description is given together with the Analysis Options.

<table>
<thead>
<tr>
<th><code>&lt;list_item&gt;</code></th>
<th>Default Display</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total</strong></td>
<td>The total time within the function.</td>
</tr>
<tr>
<td><strong>MIN</strong></td>
<td>The shortest measured time it took to execute the function. The time includes the execution times of all subfunction calls. The time used for interrupt requests is not included, unless the window is opened with option IncludeINTR. If the function was never executed completely, the MIN time is not displayed.</td>
</tr>
<tr>
<td><strong>MAX</strong></td>
<td>The longest measured time it took to execute the function. The time includes the execution times of all subfunction calls. The time used for interrupt requests is not included, unless the window is opened with option IncludeINTR.</td>
</tr>
<tr>
<td><strong>AVeRage</strong></td>
<td>The average time it took to execute the function. The time includes the execution times of all subfunction calls. The time used for interrupt requests is not included, unless the window is opened with option IncludeINTR.</td>
</tr>
<tr>
<td><strong>Count</strong></td>
<td>Number of calls of the function. If a function is never completely executed, no number of calls is displayed.</td>
</tr>
</tbody>
</table>

If function entries or exits are missing, this is display in the following format:

<times within the function>. (<number of missing function entries>/<number of missing function exits>).

Example: count 2.(2/0)

**Interpretation examples:**

1. 950. (0/1): 950. times within the function, 1 function exit is missing.
2. 9. (1/0): 9. times within the function, 1 function entry is missing.
3. 11. (1/1): 11. times within the function, 1 function entry and 1 function exit is missing.
4. 9. (0/3): 9. times within the function, 3 function exits missing.

If the number of missing function entries or exits is higher the 1. the analysis performed by the command `Trace.STATistic.Func` might fail due to nesting problems. A detailed view to the trace contents is recommended.

In some cases a further treatment of the trace contents might help. For more information refer to `Adjusting the Measurement`.
<table>
<thead>
<tr>
<th><strong>&lt;list_item&gt;</strong></th>
<th><strong>Time only in Function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internal</strong></td>
<td>Total time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines, other tasks …</td>
</tr>
<tr>
<td><strong>IAVeRage</strong></td>
<td>Average time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines, other tasks …</td>
</tr>
<tr>
<td><strong>IMIN</strong></td>
<td>Shortest between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines, other tasks …</td>
</tr>
<tr>
<td><strong>IMAX</strong></td>
<td>Longest time spent in the function between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines, other tasks …</td>
</tr>
<tr>
<td><strong>InternalRatio</strong></td>
<td>$\frac{\text{internal_time_of_function}}{\text{total_measurement_time}}$ as a numeric value.</td>
</tr>
<tr>
<td><strong>InternalBAR</strong></td>
<td>$\frac{\text{internal_time_of_function}}{\text{total_measurement_time}}$ graphically.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&lt;list_item&gt;</strong></th>
<th><strong>Time in Sub-Functions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>External</strong></td>
<td>Total time spent within called sub-functions, TRAP handlers, interrupt service routines, other tasks …</td>
</tr>
<tr>
<td><strong>EAVeRage</strong></td>
<td>Average time spent within called sub-functions, TRAP handlers, interrupt service routines, other tasks …</td>
</tr>
<tr>
<td><strong>EMIN</strong></td>
<td>Shortest time spent within called sub-functions, TRAP handlers, interrupt service routines, other tasks …</td>
</tr>
<tr>
<td><strong>EMAX</strong></td>
<td>Longest time spent within called sub-functions, TRAP handlers, interrupt service routines, other tasks …</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>&lt;list_item&gt;</strong></th>
<th><strong>Interrupt Times</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTR</strong></td>
<td>Total time the function was interrupted.</td>
</tr>
<tr>
<td><strong>INTRMAX</strong></td>
<td>Max. time 1 function pass was interrupted.</td>
</tr>
<tr>
<td><strong>INTRCount</strong></td>
<td>Number of interrupts that occurred during the function run-time.</td>
</tr>
</tbody>
</table>
The `<list_items>` can be arranged as shown in the following examples:

```
Trace.STATistic.Func INTR DEFault
Trace.STATistic.Func Internal IAVeRage Count INTR InternalRatio
```

The `<list_items>` can also be arranged by pushing the **Config** button in the Trace.STATistic.Func window.
<table>
<thead>
<tr>
<th>Basic &lt;options&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Displays trace memory contents loaded with Trace.FILE.</td>
</tr>
<tr>
<td>FlowTrace</td>
<td>The trace works as flow trace. This option is usually not required.</td>
</tr>
<tr>
<td>BusTrace</td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td>Accumulate</td>
<td>By default only the current trace contents is analyzed by the statistic functions. The option /Accumulate allows to add the current trace contents to the already displayed results.</td>
</tr>
<tr>
<td>INCremental</td>
<td>Intermediate results are displayed while the TRACE32 software analyses the trace contents (default).</td>
</tr>
<tr>
<td>FULL</td>
<td>The result is displayed after the TRACE32 software finished the analysis.</td>
</tr>
<tr>
<td>Track</td>
<td>Track the Trace.STATistic window with other trace list windows (tracking to record number or time possible).</td>
</tr>
<tr>
<td>NoMerge</td>
<td>(For diagnosis purpose only).</td>
</tr>
</tbody>
</table>
### TASK Option

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK <code>&lt;task&gt;</code></td>
<td>Performs a nesting function run-time analysis only on specified task. See also “What to know about the Task Parameters” (general_ref_t.pdf).</td>
</tr>
<tr>
<td>TASK <code>!&lt;task&gt;</code></td>
<td>Excludes a specified task from nesting function run-time analysis. This option can be useful if the nesting analysis for the specified task is problematic. See also “What to know about the Task Parameters” (general_ref_t.pdf).</td>
</tr>
</tbody>
</table>

![Graphical representation of function analysis](image_url)
<table>
<thead>
<tr>
<th>&lt;option&gt;</th>
<th>Configuration of the Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(default)</td>
<td>Function run-times are calculated without interrupts.</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Start of measurement**
- **Entry to func1**
- **Exit of func1**
- **Entry to func1**
- **func2**
- **TRAP1**
- **func3**
- **interrupt 1**
- **Exit of func1**
- **Entry to func1**
- **Exit of func1**
- **End of measurement**

©1989-2020 Lauterbach GmbH
<table>
<thead>
<tr>
<th>&lt;option&gt;</th>
<th>Configuration of the Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>IncludeINTR</td>
<td>Function run-times include times in interrupts. In other words, interrupts are treated as sub-functions.</td>
</tr>
</tbody>
</table>
<option> Configuration of the Analysis (RTOS)

| IncludeOWN +  | Function run-times without interrupts and without times in other tasks (default). Interrupts are assigned to (root)@ (root) |

**Diagram:***
- Start of measurement
- First entry to TASK1
- Entry to func1 in TASK1
- func2 in TASK1
- func2 in TASK1
- func3 in TASK1
- TRAP1 in TASK1
- func4 in TASK1
- func4 in TASK1
- interrupt1 in TASK1
- Exit of func1 in TASK1
- Entry to func1 in TASK1
- Exit of func1 in TASK1
- Total of (root)@root
- Total of (root)@TASK1
- Internal of func1@TASK1
- External of func1@TASK1
- Total of func1@TASK1
- Internal of func1@TASK1
- External of func1@TASK1
- First task switch recorded to trace

©1989-2020 Lauterbach GmbH
Include TASK + INTRROOT

Configuration of the Analysis (RTOS)
Function run-times without interrupts but with times in other tasks.
Interrupts are assigned to (root)@(root)
<table>
<thead>
<tr>
<th>Option</th>
<th>Configuration of the Analysis (RTOS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IncludeOWN +</td>
<td>Function run-times without interrupts and without times in other tasks (default).</td>
</tr>
<tr>
<td>INTRTASK</td>
<td>Interrupts are assigned to (root) @ <code>&lt;task_name&gt;</code></td>
</tr>
</tbody>
</table>

Diagram:
- Start of measurement
- First entry to TASK1
- Entry to func1 in TASK1
- func2 in TASK1
- TASK2
- func2 in TASK1
- func3 in TASK1
- TRAP1 in TASK1
- func4 in TASK1
- TASK3
- func4 in TASK1
- interrupt1 in TASK1
- Exit to func1 in TASK1
- Entry to func1 in TASK1
- Last exit of TASK1
- Exit of func1 in TASK1
### Sorting

<table>
<thead>
<tr>
<th>/Sort &lt;item&gt;</th>
<th>Sorting the Analysis Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>Sorting by program flow (default)</td>
</tr>
<tr>
<td>Nesting</td>
<td>Sorting by nesting</td>
</tr>
<tr>
<td>Address</td>
<td>Sorting by addresses</td>
</tr>
<tr>
<td>sYmbol</td>
<td>Sorting by names</td>
</tr>
<tr>
<td>TotalRatio/Ratio</td>
<td>Sorting by TotalRatio</td>
</tr>
<tr>
<td>Count</td>
<td>Sorting by Count</td>
</tr>
<tr>
<td>Window Global</td>
<td>(ineffectual)</td>
</tr>
</tbody>
</table>

The sorting can also be arranged by pushing the **Config** button in the **Trace.STATistic.Func** window.

If **All Windows** is selected, the selected sorting method is applied to all **Trace.STATistic** and **Trace.Chart** windows.

See also **Trace.STATistic.Sort**.
Adjusting the Measurement

- **Trace.STATistic.FIRST/ Trace.STATistic.LAST**
  The Trace.STATistic commands analyze the complete trace contents by default. The command Trace.STATistic.FIRST allows to freely select the start point for the statistic analysis; the command Trace.STATistic.LAST allows to freely select the end point for the statistic analysis.

- **sYmbol.NEW.MARKER FENTRY / FEXIT**
  If the function nesting analysis can’t identify code sections as HLL functions (e.g. assembler function, unusual function exits) these code sections can be marked manually as functions by using the marker FENTRY and FEXIT.

Example 1:

```
; mark the entry of the assembler function ass_int as function entry
sYmbol.New.MARKER FENTRY ass_int

; mark the exit of the assembler function ass_int as function exit
sYmbol.New.MARKER FEXIT ass_int+0x15F

; list the marker
sYmbol.List.MARKER
```

Since func3 is the HLL function executed after an interrupt occurred, it is regarded as interrupt service routine.

Since ass_int is now marked as a function, it is correctly identified as interrupt service routine.
Example 2:

Since interrupt1 is the HLL function executed after an interrupt occurred, it is regarded as interrupt service routine. The assembler code from ass_int is added to the time in func2.

Since ass_int is now marked as function, it is correctly identified as interrupt service routine. interrupt1 is a sub-function called by ass_int now.
If the KERNEL is using special methods to call/end KERNEL functions, this might annoy the function nesting analysis. In such a case it is recommended to exclude the KERNEL from the function nesting by using the markers KENTRY/KEXIT.

Example:

The KERNEL is manipulating the return address on the stack in order to return quickly into TASK1. This behavior will annoy the function nesting analysis.

The usage of the markers KENTRY/KEXIT excluded the KERNEL from the function nesting in order to get a correct function nesting.
Advanced example for RTOS RTXC on a StarCore CPU:

```plaintext
; mark all interrupt service routines as kernel entries
sYmbol.ForEach "sYmbol.NEW.MARKER KENTRY ** "_isr_*"

; mark all RTE instructions in the specified program range as kernel exit
Data.Find P:RTXCProlog--P:RTXCProlog_end %Word 0x9f73
WHILE FOUND()
{
  sYmbol.NEW.MARKER KEXIT P:TRACK.ADDRESS()
  Data.Find
}

sYmbol.List.MARKER
```

**Procedure for Measurement for TRACE32-ICE and TRACE32-FIRE**

**Mark the Functions**

The measurement is based on a selective trace of all function entries and exits. In order to perform this measurement all function entries have to be marked with an Alpha breakpoint, and all function exits have to be marked with a Beta breakpoint.

These breakpoints can be set automatically with the command `Break.SetFunc`, if HLL-functions are loaded. Assembler functions or e.g. loops within HLL-functions can also be marked manually.

The entry points of interrupt routines should be marked with Alpha and Charly (or Beta on ECC8) breakpoints. Thus the analysis ignores interrupt function times and takes care about double fetches caused by interrupted programs.

If all interrupt routines are located in a specific memory range or within a specific HLL module the interrupt routines can be marked automatically by using the command `Break.SetFunction <range>|<module>` /INTR.

If there is a name convention for interrupt routines use the `sYmbol.ForEach` command to mark the interrupt routines.
Programming of the Trigger Unit

The analyzer is programmed to record accesses to ALPHA or BETA breakpoints and mark the records in the trace buffer as follows:

- Function entries are marked with an A marker.
- Function exits are marked with a B marker.
- Interrupt entries are marked with an A and a C marker.
- Interrupt exits are marked with a B marker.

```c
Analyzer.ReProgram
{
    Sample.Enable IF AlphaBreak||BetaBreak
    Mark.A if AlphaBreak
    Mark.B if BetaBreak
    Mark.C if CharlyBreak
} ; not on ECC8
```

Recording

The interesting program flow is recorded. The trace memory should be cleared before starting the recording. This is automatically done if Analyzer.AutoInit is ON.

Check the trace by viewing the records with `<trace>.List MARK Default`.

The display of the nesting is possible with the command Analyzer.List FUNC FUNCR (this command uses the same strategy to determine function entries and exits).

If the nesting of the functions is not correct (each function-entry must have a corresponding function-exit), the results will be incorrect and an error message will be displayed in the statistic window. In this case use the command `<trace>.STATistic.PreFetch` to solve the problem. If still single records are wrong, the command `<trace>.STATistic.Ignore` can be used to remove them from the measurement.
### Options

<table>
<thead>
<tr>
<th>Basic &lt;options&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Displays trace memory contents loaded with <code>Trace.FILE</code>.</td>
</tr>
<tr>
<td>FlowTrace</td>
<td>The trace works as flow trace. This option is usually not required.</td>
</tr>
<tr>
<td>BusTrace</td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td>Accumulate</td>
<td>By default only the current trace contents is analyzed by the statistic functions. The option <code>/Accumulate</code> allows to add the current trace contents to the already displayed results.</td>
</tr>
<tr>
<td>INCremental</td>
<td>Intermediate results are displayed while the TRACE32 software analyses the trace contents (default).</td>
</tr>
<tr>
<td>FULL</td>
<td>The result is displayed after the TRACE32 software finished the analysis.</td>
</tr>
<tr>
<td>Track</td>
<td>Track the <code>Trace.STATistic</code> window with other trace list windows (tracking to record number or time possible).</td>
</tr>
<tr>
<td>NoMerge</td>
<td>(For diagnosis purpose only).</td>
</tr>
</tbody>
</table>

**CTS**

Use **Context Tracking System** to fill trace gaps and then perform the `<trace>.STATistic.FUNC` command. This is only useful for the ARM-ETM, SH4 and NEXUS.

**INTR**

The time spent in interrupts is included to the measurement like a function call.
The `<list_items>` can be arranged by pushing the `Config` button in the `<trace>.STATistic.FUNC` window.

<table>
<thead>
<tr>
<th>NAME</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Displays the function name. Interrupt functions are marked with ➔.</td>
</tr>
<tr>
<td>Time</td>
<td>Time between entry and exit summed over the complete measurement time. By default the time spent in interrupt routines is taken out of the measurement.</td>
</tr>
<tr>
<td>MIN</td>
<td>Shortest time from function entry to exit.</td>
</tr>
<tr>
<td>MAX</td>
<td>Longest time from function entry to exit.</td>
</tr>
<tr>
<td>AVerAge</td>
<td>Average time from function entry to exit.</td>
</tr>
<tr>
<td>Internal</td>
<td>Time spent within the function (without called sub functions).</td>
</tr>
<tr>
<td>IAVerAge</td>
<td>Average time spent in the function (without called sub functions).</td>
</tr>
<tr>
<td>IMIN</td>
<td>Shortest time spent in the function (without called sub functions).</td>
</tr>
<tr>
<td>IMAX</td>
<td>Longest time spent in the function (without called sub functions).</td>
</tr>
<tr>
<td>External</td>
<td>Time spent within sub functions.</td>
</tr>
<tr>
<td>EAVerAge</td>
<td>Average time spent within sub functions.</td>
</tr>
<tr>
<td>EMIN</td>
<td>Shortest time spent within sub functions.</td>
</tr>
<tr>
<td>EMAX</td>
<td>Longest time spent within sub functions.</td>
</tr>
<tr>
<td>MAXIntr</td>
<td>Maximum time the function was interrupted by an interrupt.</td>
</tr>
<tr>
<td>Count</td>
<td>Number of calls of the function. A negative value in brackets shows the number of not complete entries or exits to the function (i.e. when the measurement is stopped).</td>
</tr>
<tr>
<td>Ratio</td>
<td>Ratio of time spent within the function over the complete measurement time (without called subroutines).</td>
</tr>
<tr>
<td>BAR.log, BAR.LIN</td>
<td>Graphical display of the ratio (linear or logarithmic).</td>
</tr>
<tr>
<td>TRatio</td>
<td>Ratio of time spent within the function over the complete measurement time (called subroutines included).</td>
</tr>
<tr>
<td>TBAR.log, TBAR.LIN</td>
<td>Graphical display of the ratio (linear or logarithmic).</td>
</tr>
</tbody>
</table>
**Format**

- **LEN <size>** Specifies the width of non numeric fields (e.g. symbols)
- **TimeAuto** Adapt the time display. (default)
- **TimeFixed** Display all time information in seconds.

**See also**

- `<trace>.STATistic`
- 'Function Run-Times Analysis' in 'ARM-ETM Training'
- 'Nesting Function Run-Time Analysis - Single' in 'AURIX Trace Training'
- 'Function Run-Times Analysis - Single' in 'Nexus Training'
- 'Release Information' in 'Release History'
**<trace>.STATistic.FuncDURation**

Statistic analysis of single function

Format:  

```
<trace>.STATistic.FuncDURation  <function_name>
```

Analyzes the function runtime between function entry and exit.

- The time spent in called subroutines is *included*.
- The time spent in called interrupt service routine and other tasks is *excluded*.

See also

- `<trace>.STATistic.FuncDURationInternal`
- `<trace>.STATistic`
- 'Release Information' in 'Release History'
- 'Function Run-Times Analysis - Single' in 'Nexus Training'
<trace>.STATistic.FuncDURationInternal

Statistic analysis of single func.

Format:  
<trace>.STATistic.FuncDURationInternal  <function_name>

Analyzes the function runtime between function entry and exit. The time spent in called subroutines, traps, interrupt service routine and other tasks is excluded.

See also

- <trace>.STATistic.FuncDURation
- <trace>.STATistic
- 'Release Information' in 'Release History'
Group run-time analysis

The time spent in groups and the number of calls is calculated (flat statistic).

Example:

```
GROUP.Create "INPUT" \jquant2 \jquant1 \jidctred \jinput /AQUA
GROUP.Create "JPEG" \jdapimin \jcolor \jddctmgr \jcoefct /NAVY
Go
Break
Trace.STATistic.GROUP
```

See also

- `<trace>.STATistic`
- `GROUP.Create`
- 'Release Information' in 'Release History'
Format:  
\texttt{<trace>.STATistic.Ignore [<record> | <range>] [/<options>]} 

\texttt{<option>}:  \texttt{FILE | BusTrace} 

The specified record(s) are ignored in the statistic analysis. This command can be used, when single records (caused by prefetch etc.) confuse the statistic analysis.

**FILE**  
Displays trace memory contents loaded with \texttt{Trace.FILE}. 

**BusTrace**  
The trace works as a bus trace. This option is usually not required.

**Example:** The state of the ignore bit for each record can be displayed in the \texttt{<trace>.List} window by the \texttt{IGNORE} field.

\begin{verbatim}
measurement ...
...
Trace.STATistic.PreFetch ON ; enable prefetching -> will ; ignore most prefetch cases
Trace.STATistic.TREE ; display function nesting ; tree
Trace.List IGNORE Func MARK CPU ; control window for ; nesting analysis ...
Trace.STATistic.Ignore (-1000.)--(-995.) ; ignore six records
\end{verbatim}
Format: \( <\text{trace}>.\text{STATistic.INTERRUPT} \ [\%<\text{format}>] \ [<\text{list}\_\text{item}> \ldots] \ [/<\text{option}>] \)
In order to calculate the results for the *nesting function run-time analysis* the trace recording is post-processed. One important issue in this processing is the identification of interrupt entries and exits.

TRACE32 provides two methods to identify interrupt entries and exits:

- Default: `<trace>.STATistic.InterruptIsFunction OFF`
- Recommended: `<trace>.STATistic.InterruptIsFunction ON`

**Trace.STATistic.InterruptIsFunction OFF**

The screenshot below shows the function nesting for the interrupt.

1. The first HLL function called after the indirect branch to the Interrupt Vector Table is regarded as interrupt service routine (here OSInterruptDispatcher1).
2. The return from interrupt is regarded as the exit of this function (here OSInterruptDispatcher1).

Please be also aware that some trace port protocols require special setups for the Interrupt Vector Table. For details, please refer to your *Processor Architecture Manual*. 
1. Interrupt entry is the point in the trace recording at which the indirect branch to the Interrupt Vector Table occurs.

2. Interrupt exit is the point in the trace recording at which the return from interrupt is executed.

TRACE32 handles the time between interrupt entry and exit as a function. The name given to this function is the label of the interrupt vector address.

Please be aware that method only works if interrupts are exit by regular return from interrupt.

Please be also aware that some trace port protocols require special setups for the Interrupt Vector Table. For details, please refer to your Processor Architecture Manual.

See also
- <trace>.STATistic

<trace>.STATistic.INTERRUPTTREE
Display interrupt nesting

Format: <trace>.STATistic.INTERRUPTTREE
The Trace.STATistic commands analyze the complete trace contents by default. The command Trace.STATistic.LAST allows to freely select an end point for the statistic analysis.

Example for <value>:

Trace.List ; display trace listing
Trace.STATistic.FIRST -123366. ; select trace record -123366.
; as start point for the trace
; analysis
Trace.STATistic.LAST -36675. ; select trace record -36675.
; as end point for the trace
; analysis
Trace.STATistic(Func ; perform a function run-time
; analysis

Format: <trace>.STATistic.LAST <value> | <time> | <string>
Exampe for `<time>`:

```plaintext
Trace.List Time.Zero DEFAULT ; display trace listing
Trace.STATistic.LAST 468.2us ; select trace record with timestamp
                          ; 468.2 µs (zero time) as end point for
                          ; the trace analysis
Trace.STATistic.Func     ; perform a function run-time analysis
                          ; from the beginning of the trace buffer
                          ; to the specified end point
```

See also

- `<trace>.STATistic`
- `<trace>.STATistic.FIRST`

▲ 'Release Information' in 'Release History'
HLL-line analysis

Format:  \(<trace>.STATistic.Line [/<option>]\)

<options>:
- FILE
- FlowTrace | BusTrace
- Accumulate
- INCremental | FULL
- Sort <item>
- Track
- NoMerge
- BEFORE | AFTER
- List <item> ...
- Filter <item>

Analyzes the time spent in HLL lines.

Options

<table>
<thead>
<tr>
<th>Basic &lt;options&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Displays trace memory contents loaded with Trace.FILE.</td>
</tr>
<tr>
<td>FlowTrace</td>
<td>The trace works as flow trace. This option is usually not required.</td>
</tr>
<tr>
<td>BusTrace</td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td>Accumulate</td>
<td>By default only the current trace contents is analyzed by the statistic functions. The option /Accumulate allows to add the current trace contents to the already displayed results.</td>
</tr>
<tr>
<td>INCremental</td>
<td>Intermediate results are displayed while the TRACE32 software analyses the trace contents (default).</td>
</tr>
<tr>
<td>FULL</td>
<td>The result is displayed after the TRACE32 software finished the analysis.</td>
</tr>
</tbody>
</table>
Basic <options> | Description
--- | ---
Track | Track the Trace.STATistic window with other trace list windows (tracking to record number or time possible).
NoMerge | (For diagnosis purpose only).

BEFORE | Display the time before the program entered the listed HLL line, that means how long the program was in the previous HLL line before the listed HLL line was entered.
AFTER | Display the time after the program entered the listed HLL line, that means how long the program was in the listed HLL line (default).
List <items> | Specify the result that should be displayed in the window.
Filter <item> | Filter the HLL lines to analyze only a specific function or module.

The List option defines which values are calculated and displayed. The <list_items> can be arranged by pushing the Config button in the Trace.STATistic.Line window.

<list_item>

**TIme** | Total time in the HLL line.
**MIN, MAX** | Minimum and maximum times in the HLL line.
**AVeRage** | Average time in the HLL line.
**Count** | Number of executions of the HLL line.
**Ratio, BAR.log, BAR.LINear** | Ratio of time spent in the HLL line to total measurement time in percent and as graphical bars.
**CRatio, CBAR.log, CBAR.LINear** | Ratio of count to total count in percent and as graphical bars.

All time displays depend on the options AFTER or BEFORE.
Example for TRACE32-ICD and PowerTrace:

If no selective trace is possible use the option /Filter to filter out the module or function of interest.

```
Go
Break
Trace.STATistic.Line /Filter V.RANGE(sieve)
```

Example for TRACE32-ICE and TRACE32-FIRE:

If only a specific module or function should be analyzed, perform a selective trace on the module or function.

```
ANalyzer.ReProgram
  (  
    ADDR AlphaBreak V.RANGE(module1)
    Sample.Enable IF AlphaBreak
  )
Go
Break
Trace.STATistic.Line
```

See also

- `<trace>.STATistic`
- 'Release Information' in 'Release History'
<trace>.STATistic.LINKage

Per caller statistic of function

<table>
<thead>
<tr>
<th>Format:</th>
<th><code>&lt;trace&gt;.STATistic.LINKage &lt;address&gt; [&lt;items&gt; …] [&lt;option&gt;]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;format&gt;</code>:</td>
<td>DEFault</td>
</tr>
<tr>
<td></td>
<td>TimeAuto</td>
</tr>
<tr>
<td><code>&lt;list_item&gt;</code>:</td>
<td>DEFault</td>
</tr>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td></td>
<td>Count</td>
</tr>
<tr>
<td></td>
<td>NAME</td>
</tr>
<tr>
<td></td>
<td>Internal</td>
</tr>
<tr>
<td></td>
<td>External</td>
</tr>
<tr>
<td></td>
<td>INTR</td>
</tr>
<tr>
<td></td>
<td>ExternalTASK</td>
</tr>
<tr>
<td></td>
<td>TotalRatio</td>
</tr>
<tr>
<td><code>&lt;option&gt;</code>:</td>
<td>FILE</td>
</tr>
<tr>
<td></td>
<td>FlowTrace</td>
</tr>
<tr>
<td></td>
<td>Accumulate</td>
</tr>
<tr>
<td></td>
<td>INCremental</td>
</tr>
<tr>
<td></td>
<td>Sort</td>
</tr>
<tr>
<td></td>
<td>Track</td>
</tr>
<tr>
<td></td>
<td>NoMerge</td>
</tr>
<tr>
<td></td>
<td>IncludeOwn</td>
</tr>
<tr>
<td></td>
<td>INTRROOT</td>
</tr>
</tbody>
</table>

Performs a function run-time statistic for a single function itemized by its callers. The procedure for recording the data is the same as for the `<trace>.STATistic.Func` command.

`<address>` Has to be the function entry address.
Example:

```
Trace.STAT.LINKage alloc_small
```

The function alloc_small was called by the listed 20 functions. The dependency between the run-time of the function allow_small and its callers is analyzed.

See also

- `<trace>.STATistic`
- 'Release Information' in 'Release History'
This command allows to analyze the performance of a single signal. It is mainly used with PowerProbe or PowerIntegrator.

Typical application for the `<trace>.STATistic.Measure`:
- to check the best threshold level for a symmetric signal (e.g. a symmetric clock signal).
- to detect spikes (e.g. a signal has a defined period of 10.ns, detect if there is any much smaller period).

### Format:

```
<trace>.STATistic.Measure [ <record> | <range> ] [ <items> … ] [ <options> ]
```

### recs
The number of records that are analyzed.

### time
The time that is analyzed.

### lead
The time from the beginning of the analysis until the first edge.

### tail
The time from the last edge until the end of the analysis.

### \( \backslash / \) : 
The number of low states.

### \( \backslash \) : 
The number of high states
The analysis can also be activated by selecting the signal in the **Trace.Timing** display and by using the pull-down menu provided via the right mouse button.

It is also possible to analyze only the selected part of the complete recording time.

---

**See also**

- `<trace>.STATistic`
- 'Release Information' in 'Release History'
<trace>.STATistic.MODULE  
Code execution broken down by module

Format:  <trace>.STATistic.MODULE [%<format>] [<list_items> …] [/<option>]

Shows a statistical analysis of the code execution broken down by module.

See also
- <trace>.STATistic
- <trace>.STATistic.PROGRAM

▲ 'Release Information' in 'Release History'

<trace>.STATistic.PAddress  
Which instructions accessed data address

Format:  <trace>.STATistic.PAddress /Filter Address <address>

The command provides a statistic about the instructions that accessed the data address specified by the Filter option.

Example:

Trace.STATistic.PAddress /Filter Address mstatic1

See also
- <trace>.STATistic

▲ 'Release Information' in 'Release History'
Show the call context of a function. The function is specified by its start address.

Example:

```
Trace.STATistic.ParentTREE alloc_small
```

See also
- `<trace>.STATistic`
- `<trace>.STATistic.ChildTREE`
Prefetch detection

This command should only be used when a prefetching CPUs is used!

If ON is selected, the software will try to eliminate the errors in statistics generated by prefetches. The command affects the result of the commands `<trace>.STATistic`, `<trace>.Chart` and `<trace>.List FUNC` (and all related commands).

When the option cycles is used, another software prefetch detection strategy is added. In this case after each marked record there must be a number of consecutive cycles, otherwise the record is treated as a prefetch cycle. This technique can be used to eliminate false entry/exits detections caused by prefetching and interrupts.

If the prefetch problem cannot be solved by this command, an alternative can be tagging the code (command Data.TAGFunc) or by manually removing records from the measurement (command `<trace>.STATistic.Ignore`).

The following typical prefetch conditions are detected:

**Alpha/Beta prefetched direct after Beta**

The beginning of a function may be sampled during leaving the prior function by a prefetch. This may happen, if the breakpoints were set manually or with the command Break.SetFunc /PreFetch. The prefetch will be detected by the software, if an access to Alpha or Beta is made directly after an access to Beta, without any other program fetch cycles between. If the record after the ignored record is also prefetched, it will be ignored too.

**Double fetches after interrupts**

Problems with the prefetch may occur, if interrupts are handled by the program during the analysis. After accessing a tag breakpoint on processors with prefetch it is possible that the interrupt program is entered. After leaving the interrupt program the same breakpoint is read again, causing a nesting error in the analysis. If the PreFetch option is active and the interrupt program is marked with a Charly breakpoint, the software will assume that a second read of a breakpoint tag after the interrupt function was caused by a prefetch. A result may be that recursive functions may cause nesting errors, if the interrupt occurs in such a procedure. A 100% solution is not possible, when working with prefetching CPUs and sampling program fetches. The best solution in this case is to use data tags (see command Break.SetFunc).

**Multiple prefetched Beta through program structure**

If the epilog code of a function contains a loop, which accesses the Beta breakpoint more than once, it will be detected. The last access to the Beta will be taken into account. If the beta breakpoint is accessed by a prefetch, but the function continues this will also be detected.

See also
- `<trace>.STATistic`
<trace>.STATistic.PROGRAM

Code execution broken down by program

Shows a statistical analysis of the code execution broken down by program.

See also
- <trace>.STATistic
- <trace>.STATistic.MODULE

▲ 'Release Information' in 'Release History'

<trace>.STATistic.PsYmbol

Shows which functions accessed data address

Format:  <trace>.STATistic.PsYmbol /Filter <filter>

The command provides a statistic about the functions that accessed the data address specified by the /Filter option.

Trace.STATistic.PsYmbol /Filter sYmbol mstatic1

Trace.STATistic.PsYmbol /Filter sYmbol mstatic1 CYcle Write

Preconditions:
- Has to be implemented for the processor architecture in use.
- Data access has to be clearly assignable to an instruction.
If TRACE32 was able to clearly assign the data access to an instruction can be checked as follows:

```plaintext
Trace.FindAll symbol mstatic1
```

A red cycle type indicates that a clear assignment was not possible.

```plaintext
; PAddress: address of instruction that performed the data access
; PsYmbol: symbolic address of instruction that performed the data access
Trace.List PAddress PsYmbol DEFAULT
```

Both columns are empty if no clear assignment is possible.

See also
- `<trace>.STATistic`
- ‘Release Information’ in ‘Release History’
Specify sorting criterion for the results of the command groups `<trace>.STATistic` and `<trace>.Chart`.

If the command is entered without parameters, a Trace.STATistic.Sort dialog is displayed.

The sorting criterion specified by Trace.STATistic.Sort applies to all `<trace>.STATistic` and `<trace>.Chart` analysis windows (check box All Windows ON).
To specify the sorting criterion for an individual statistic window use the **Config** button of this statistic window or use the **/Sort** option when you enter the command.

```
Trace.Chart.Symbol /Sort TotalRatio ; sort the time chart by the criterion TotalRatio
```

**Default Sorting Criterion**

**OFF** is the default mode for most statistic windows. **OFF** means that the analyzed items are displayed in their recording order.

Statistic windows that are focused on the program's call hierarchy e.g **Trace.STATistic.TREE** use **Nesting** as default mode.
Window vs. Global

<table>
<thead>
<tr>
<th>Global (default)</th>
<th>The sorting criterion is strictly maintained.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td>The sorting criterion is applied. The analyzed items active in the displayed time interval are displayed first, followed by the non-active items. <strong>Window</strong> might be useful if you scroll horizontally.</td>
</tr>
</tbody>
</table>

Trace.Chart.Symbol /Sort Window Symbol
### CoreTogether vs. CoreSeparated (SMP Systems only)

<table>
<thead>
<tr>
<th>CoreTogether (default)</th>
<th>The analyzed items are displayed per core. Additional sorting criteria apply to this per core order.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreSeparately</td>
<td>The core information has no impact on the sorting order.</td>
</tr>
</tbody>
</table>

```plaintext
Trace.Chart.Symbol /ZoomTrack /Sort CoreTogether Symbol
Trace.Chart.Symbol /ZoomTrack /Sort CoreSeparated Symbol
```

![Comparison of CoreTogether and CoreSeparated displays](image-url)
Standard Sorting Criteria

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Sort result by address</td>
</tr>
<tr>
<td>sYmbol</td>
<td>Sort result alphabetically by symbol names</td>
</tr>
<tr>
<td>[&lt;wildcard_list …&gt;]</td>
<td>Sort result by their grouping</td>
</tr>
<tr>
<td>GROUP</td>
<td>Sort analyzed items by their occurrence</td>
</tr>
</tbody>
</table>

Example for sort criterion `sYmbol [<wildcard_list …>]`.

```plaintext
; display items starting with string "SPI" first, then items starting
; with string "SUP" then rest
Trace.STATistic.Sort sYmbol Spi* SUP*
Trace.Chartsymbol
```
Example for sort criterion `GROUP`.

```
GROUP.List
Trace.STATistic.Sort GROUP
Trace.Chart.symbol
```

GROUP other

GROUP my_code

GROUP toms_code

GROUP ralfs_code
## Sorting Criteria for the Nesting Analysis

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nesting</strong></td>
<td>Calling functions are displayed atop of called function.</td>
</tr>
<tr>
<td><strong>InternalRatio</strong></td>
<td>Sort result be internal ratio. InternalRatio: <code>&lt;time_in_function&gt;/total_measurement_time</code> as a numeric value.</td>
</tr>
<tr>
<td><strong>TotalRatio</strong></td>
<td>Sort result by total ratio. InternalRatio: <code>&lt;total_time_of_function&gt;/total_measurement_time</code> as a numeric value, <code>&lt;total_time_of_function&gt;</code> includes called subfunctions and traps.</td>
</tr>
</tbody>
</table>

Example for criterion **Nesting**.

```plaintext
Trace.Chart.Func /ZoomTrack /Sort Nesting
```

![Diagram of nesting analysis](image)
## Sorting Criteria for the Flat Analysis

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ratio</strong></td>
<td>Sort analyzed items by their ratio.</td>
</tr>
<tr>
<td><strong>TotalMAX</strong></td>
<td>Flat analysis with InterVal option only. Sort analyzed items by maximal total time per specified interval.</td>
</tr>
<tr>
<td><strong>RatioMAX</strong></td>
<td>Flat analysis with InterVal option only. Sort analyzed items by maximal ratio per specified interval.</td>
</tr>
</tbody>
</table>

*Trace.STATistic.sYmbol /InterVal 10.ms /Sort RatioMAX*

![Graph with sorting criteria values]

**See also**
- `<trace>-STATistic`
- 'Release Information' in 'Release History'
Flat run-time analysis

The execution time in different symbol regions is displayed. Displayed are the number of entries into the range and the time spent in the range.

```plaintext
Format:  <trace>.STATistic.sYmbol [%<format>] [<list_item> ...] [/<option>]

<format>:  DEFault | LEN
            TimeAuto | TimeFixed

<list_item>:  DEFault | ALL
              Total | MIN | MAX | AVerage
              Count
              NAME | CountRatio | CountBAR
              CountChange | CountFirst | CountALL

<options>:  FILE
            FlowTrace | BusTrace
            Accumulate
            INCremental | FULL
            Sort <item>
            Track
            NoMerge
            SplitTASK
            NoInline
            BEFORE | AFTER
            Address <function1>-<function2> ...
            Address <function_m>---<function_n>
            Filter Address <function1>-<function2> ...
            Filter Address <function_m>---<function_n>
            ALL
```
### Configuration of Count column

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountFirst (default)</td>
<td>Count the occurrence of the start address of a program symbol region or of a function.</td>
</tr>
<tr>
<td>CountChange</td>
<td>Count how often the address range of a program symbol region or of a function was entered.</td>
</tr>
<tr>
<td>CountALL</td>
<td>Count all executed instructions.</td>
</tr>
</tbody>
</table>

### Basic <options> Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Displays trace memory contents loaded with <code>Trace.FILE</code>.</td>
</tr>
<tr>
<td>FlowTrace</td>
<td>The trace works as flow trace. This option is usually not required.</td>
</tr>
<tr>
<td>BusTrace</td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td>Accumulate</td>
<td>By default only the current trace contents is analyzed by the statistic functions. The option <code>/Accumulate</code> allows to add the current trace contents to the already displayed results.</td>
</tr>
<tr>
<td>INCremental</td>
<td>Intermediate results are displayed while the TRACE32 software analyses the trace contents (default).</td>
</tr>
<tr>
<td>FULL</td>
<td>The result is displayed after the TRACE32 software finished the analysis.</td>
</tr>
<tr>
<td>Track</td>
<td>Track the <code>Trace.STATistic</code> window with other trace list windows (tracking to record number or time possible).</td>
</tr>
<tr>
<td>NoMerge</td>
<td>(For diagnosis purpose only).</td>
</tr>
<tr>
<td>SplitTASK</td>
<td>Splits up the results for different tasks.</td>
</tr>
<tr>
<td>NoInline</td>
<td>Inline functions are treated as separate functions. The option <code>NoInline</code> can be used to discard inline functions.</td>
</tr>
<tr>
<td>BEFORE</td>
<td>Display the time before the program entered the listed symbol range, that means how long the program was in the previous symbol range until the listed symbol range was entered.</td>
</tr>
</tbody>
</table>
AFTER Display the time after the program entered the listed symbol range, that means how long the program was in the listed symbol range (default).

Filter … The recorded trace information is first filtered and then analyzed.

; filter the specified functions out of the trace stream
; and then analyze the filtered trace information
Trace.STATistic.Symbol /Filter Address main||func2||func10||func26

Recording (filtered functions are displayed in black)

Analysis result
Perform statistic on specified functions, assign statistic information for all other functions to (other).

The **GROUP** command provides more features to structure your statistic.

```
Trace.STATistic.sYmbol /Address func2||func10||sfpDoubleNormalize
Trace.STATistic.sYmbol /Address func2--func7
```
### The `<list_items>` can be arranged by pushing the Config button in the `<trace>.STATistic.sYmbol` window.

<table>
<thead>
<tr>
<th><code>&lt;list_item&gt;</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>Total time in the symbol range.</td>
</tr>
<tr>
<td><strong>MIN, MAX</strong></td>
<td>Minimum and maximum times in the symbol range.</td>
</tr>
<tr>
<td><strong>A VeRage</strong></td>
<td>Average time in the symbol range.</td>
</tr>
<tr>
<td><strong>Count</strong></td>
<td>Number of entries to the symbol range.</td>
</tr>
<tr>
<td><strong>Ratio, BAR.log, BAR.LINear</strong></td>
<td>Ratio of time spent in the symbol range to total measurement time in percent and as graphical bars.</td>
</tr>
<tr>
<td><strong>CRatio, CBAR.log, CBAR.LINear</strong></td>
<td>Ratio of count to total count in percent and as graphical bars.</td>
</tr>
</tbody>
</table>

All time displays depend on the options **AFTER** or **BEFORE**.

### Example for PowerTrace:

If no selective trace is possible use the option `/Filter` to filter out the module or function of interest.

```
Go
Break
Trace.STATistic.Sort sYmbol ; sort the result alphabetically
Trace.STATistic.sYmbol /Filter Address Y.SECRANGE(\\diab555\.text)
```

### Example for TRACE32-ICE and TRACE32-FIRE:

If only a specific module or function should be analyzed, perform a selective trace on the module or function.

```
Analyzer.ReProgram
{
    ADDR AlphaBreak ; sample only instructions from
    Y.SECRANGE(\\diab555\.text) ; program \diab555
    Sample.Enable IF AlphaBreak
}
Go
Break
Trace.STATistic.Sort Ratio ; sort the result by ratio
Trace.STATistic.sYmbol ; display the result
```

---

**See also**

- `<trace>.Chart.sYmbol`
- `<trace>.STATistic`
- "Release Information" in "Release History"

©1989-2020 Lauterbach GmbH
### Task activity statistic

**Format:**

```
<trace>.STATistic.TASK [%<format>] [<list_items>] [/<option>]
```

| **<format>**       | DEFault | LEN
|--------------------|---------|-------
|                    | TimeAuto | TimeFixed |

| **<list_items>**   | DEFault | ALL
|--------------------|---------|-------
|                    | Total | TotalMIN | TotalMAX
|                    | MIN | MAX | AVeRage
|                    | Count | CountMIN | CountMAX
|                    | Ratio | RatioMIN | RatioMAX
|                    | BAR.log | BAR.LINEar
|                    | TASK | GROUP |

| **<option>**       | FILE
|--------------------|-------
|                    | InterVal | <time> | <event>
|                    | Accumulate |
|                    | INCremental | FULL
|                    | Sort | <item>
|                    | Track |

|                    | FlowTrace | BusTrace (diagnosis only) |
|                    | CORE | SplitCORE (default) | MergeCORE | JoinCORE |
|                    | (SMP Systems only) |

Task run-times are analyzed.

If a core trace is used, “**OS-aware Tracing**” (glossary.pdf) has to be enabled in order to use this command.
Survey

<table>
<thead>
<tr>
<th>task</th>
<th>Number of recorded tasks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>Time period recorded by trace.</td>
</tr>
</tbody>
</table>

Task details

<table>
<thead>
<tr>
<th>column</th>
<th>&lt;list_item&gt;</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>(unknown)</td>
<td>TRACE32 assigns all trace information generated before the first task information to the (unknown) task.</td>
</tr>
<tr>
<td>total</td>
<td>Total</td>
<td>Time period in the task during the recorded time period.</td>
</tr>
<tr>
<td>min</td>
<td>MIN</td>
<td>Shortest time in task.</td>
</tr>
<tr>
<td>max</td>
<td>MAX</td>
<td>Longest time in task.</td>
</tr>
<tr>
<td>avr</td>
<td>AVerRage</td>
<td>Average time in task.</td>
</tr>
<tr>
<td>count</td>
<td>Count</td>
<td>Number of time in task.</td>
</tr>
<tr>
<td>ratio</td>
<td>Ratio</td>
<td>Ratio of time in the task with regards to the total time period recorded.</td>
</tr>
<tr>
<td>(graphical bar)</td>
<td>BAR.LOG</td>
<td>Ratio of time in the task with regards to the total time period recorded graphically.</td>
</tr>
<tr>
<td>group</td>
<td>GROUP</td>
<td>Display of group name assigned by command GROUP.CreateTASK.</td>
</tr>
</tbody>
</table>

The column layout can be configured by using the Config… button of the Trace.STATistic.TASK window or...
InterVal Analysis

The **InterVal** option allows to divide the time period recorded by the trace (total) into time slices. Additional analysis details can be displayed for these time slices.

- **Trace.STATistic.TASK /InterVal <time> | <event>**

  ; divide trace into 10.ms time slices
  Trace.STATistic.TASK /InterVal 10.ms

  ; divide trace in time slices, a new time slice is started when the
  ; function Func cpu0_generateData is entered
  Trace.STATistic.TASK /InterVal sYmbol Func cpu0_generateData
### Survey (InterVal option)

<table>
<thead>
<tr>
<th>column</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>Number of recorded tasks.</td>
</tr>
<tr>
<td>total</td>
<td>Time period recorded by trace.</td>
</tr>
<tr>
<td>intervals</td>
<td>Number of intervals.</td>
</tr>
<tr>
<td>avr</td>
<td>Average interval length.</td>
</tr>
<tr>
<td>min</td>
<td>Shortest interval.</td>
</tr>
<tr>
<td>max</td>
<td>Longest interval.</td>
</tr>
</tbody>
</table>

### Task details (InterVal option)

<table>
<thead>
<tr>
<th>column</th>
<th>item</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>totalmax</td>
<td>TotalMAX</td>
<td>Longest time period in the task within an interval.</td>
</tr>
<tr>
<td>ratiomax</td>
<td>RatioMAX</td>
<td>Highest ratio of time in the task within an interval.</td>
</tr>
<tr>
<td>countmax</td>
<td>CountMAX</td>
<td>Highest number of time in the task within an interval.</td>
</tr>
<tr>
<td>totalmin</td>
<td>TotalMIN</td>
<td>Shortest time period in the task within an interval.</td>
</tr>
<tr>
<td>ratiomin</td>
<td>RatioMIN</td>
<td>Shortest ratio of time in the task within an interval.</td>
</tr>
<tr>
<td>countmin</td>
<td>CountMIN</td>
<td>Shortest number of time in the task within an interval.</td>
</tr>
</tbody>
</table>
Options

Basis Options

FILE
Displays trace memory contents loaded with `Trace.FILE` command.

InterVal
Divide the time period recorded by the trace (total) into time slices and analyze the time slices, see examples.

Accumulate
By default only the current trace contents is analyzed. The option `/Accumulate` allows to add the current trace contents to the already displayed results.

INCremental
Intermediate results are displayed while the TRACE32 software analyzes the trace contents (default).

FULL
The result is displayed after the TRACE32 software finished the analysis.

Sort `<item>`
By default the result is sorted by the recording order. Other sorting criteria are possible. The Config button provides a quick access. For a detailed overview refer to `Trace.STATistic.Sort`.

CLOCKS
The measurement results display the number of clocks instead of time information.

FlowTrace
(Diagnosis only).

BusTrace
(Diagnosis only).

SMP Options

CORE `<n>`
TRACE32 analyzes the result only for the core with the specified number.

SplitCORE
TRACE32 displays the result per core (default).

MergeCORE
Trace information is analyzed independently for each core. The statistic summarizes these results to a single result.

JoinCORE
(Unused).

Formatting

LEN `<size>`
Specifies the width of non numeric fields (e.g. symbols)
**TRACE32-ICE or TRACE32-FIRE**

Before using this command two things must be done:

1. the switch of the RTOS kernel must be marked for recording
2. the trigger unit has to be programmed

This trigger program performs the recording of all accesses to the magic word (a memory location that defines which task is running).

Mark the task switch and program the trigger unit (TASK configured)

If the `TASK` command is configured, the magic word (the memory location that defines which task is running) has to be marked with an AlphaBreak. The trigger program samples all write accesses to the address marked by an AlphaBreak.

Prepare the recording after using a `TASK` configuration

```
; Mark the magic location with an Alpha breakpoint
Break.Set task.config(magic)+ task.config(magicsize)-1) /Alpha

; Program the Analyzer to record only task switches
Analyzer.ReProgram
(  Sample.Enable if AlphaBreak&&Write
 )
```

Mark the task switch and program the trigger unit (TASK not configured)

If the `TASK` command is not configured, the following procedure is necessary:

- Mark the memory location that defines which task is running by an AlphaBreak.
- Sample only write accesses to the address marked by an AlphaBreak to the trace.
- Mark the memory location that defines which task is running in the trace with a C marker.

**General Commands Reference Guide T**
Prepare the recording:

```
Break.Set v.range(current_task) /Alpha

Analyzer.ReProgram
  (Sample.Enable if AlphaBreak&&Write
   Mark.C if AlphaBreak&&Write)
```

**Recording**

The trace memory should be cleared before starting the recording. The command `<trace>.Chart.TASK` displays a time chart of the running tasks.

Please see also the **OS Awareness Manuals** for the supported kernels.

**See also**

- `<trace>.STATistic.TASKFunc`
- `<trace>.STATistic`
- 'OS-Aware Tracing' in 'ARM-ETM Training'
- 'OS-Aware Tracing - Single-Core and AMP' in 'AURIX Trace Training'
- 'OS-Aware Tracing - SMP Systems' in 'AURIX Trace Training'
- 'OS-Aware Tracing' in 'Intel® Processor Trace Training'
- 'OS-Aware Tracing - Single Core' in 'Nexus Training'
- 'OS-Aware Tracing - SMP Systems' in 'Nexus Training'
- 'Release Information' in 'Release History'
The time spent in functions and the number of calls is measured. The application can run under a multitask kernel. Functions that are used by multiple tasks are displayed for each task.
The **INTR** option includes interrupt functions in the measurement like function calls. Without this option the time spend in interrupt functions is taken out of the measurement.

The **INTRTASK** option takes interrupts out of the measurement, but displays the interrupt times for each task in a separate line.

```
E::Trace.STATistic.TASKFunc ALL
<table>
<thead>
<tr>
<th>range</th>
<th>time</th>
<th>min</th>
<th>max</th>
<th>avr</th>
<th>include</th>
</tr>
</thead>
<tbody>
<tr>
<td>(root)@MSG</td>
<td>561.160us</td>
<td>0.000</td>
<td>561.160us</td>
<td>561.160us</td>
<td>561.160us</td>
</tr>
<tr>
<td>(root)@SINK</td>
<td>163.700ms</td>
<td>0.000</td>
<td>163.700ms</td>
<td>163.700ms</td>
<td>163.700ms</td>
</tr>
<tr>
<td>(root)@SRCE</td>
<td>121.730ms</td>
<td>0.000</td>
<td>121.730ms</td>
<td>121.730ms</td>
<td>121.730ms</td>
</tr>
<tr>
<td>MO_r_bench@MEM1</td>
<td>376.848ms</td>
<td>5.341ms</td>
<td>5.954ms</td>
<td>5.541ms</td>
<td>15.797ms</td>
</tr>
<tr>
<td>MO_r_bench@MEM2</td>
<td>229.561ms</td>
<td>5.341ms</td>
<td>6.269ms</td>
<td>5.599ms</td>
<td>9.364ms</td>
</tr>
<tr>
<td>_r_dmy_read@IO1</td>
<td>23.679ms</td>
<td>812.990us</td>
<td>1.480ms</td>
<td>845.698us</td>
<td>23.679ms</td>
</tr>
<tr>
<td>r_dmy_write@IO2</td>
<td>22.548ms</td>
<td>839.985us</td>
<td>1.184ms</td>
<td>835.139us</td>
<td>22.548ms</td>
</tr>
<tr>
<td>MO_r_sieve@MEM1</td>
<td>361.051ms</td>
<td>512.215us</td>
<td>1.083ms</td>
<td>530.957us</td>
<td>361.051ms</td>
</tr>
<tr>
<td>MO_r_sieve@MEM2</td>
<td>220.196ms</td>
<td>512.210us</td>
<td>818.615us</td>
<td>537.065us</td>
<td>220.196ms</td>
</tr>
</tbody>
</table>
```

```
E::Trace.STATistic.TASKFunc ALL
<table>
<thead>
<tr>
<th>range</th>
<th>exclude</th>
<th>eavr</th>
<th>maxintr</th>
<th>count</th>
<th>ratio</th>
<th>1%</th>
</tr>
</thead>
<tbody>
<tr>
<td>(root)@MSG</td>
<td>0.000</td>
<td>0.000</td>
<td>1.249s</td>
<td>1.(-2)</td>
<td>0.022%</td>
<td></td>
</tr>
<tr>
<td>(root)@SINK</td>
<td>0.000</td>
<td>0.000</td>
<td>2.277s</td>
<td>1.(-2)</td>
<td>6.629%</td>
<td></td>
</tr>
<tr>
<td>(root)@SRCE</td>
<td>0.000</td>
<td>0.000</td>
<td>2.323s</td>
<td>1.(-2)</td>
<td>4.929%</td>
<td></td>
</tr>
<tr>
<td>MO_r_bench@MEM1</td>
<td>361.051ms</td>
<td>5.309ms</td>
<td>30.349ms</td>
<td>68.</td>
<td>0.639%</td>
<td></td>
</tr>
<tr>
<td>MO_r_bench@MEM2</td>
<td>220.196ms</td>
<td>5.370ms</td>
<td>113.519ms</td>
<td>41.</td>
<td>0.379%</td>
<td></td>
</tr>
<tr>
<td>_r_dmy_read@IO1</td>
<td>0.000</td>
<td>0.000</td>
<td>127.073ms</td>
<td>28.(-2)</td>
<td>0.958%</td>
<td></td>
</tr>
<tr>
<td>r_dmy_write@IO2</td>
<td>0.000</td>
<td>0.000</td>
<td>119.853ms</td>
<td>27.(-2)</td>
<td>0.913%</td>
<td></td>
</tr>
<tr>
<td>MO_r_sieve@MEM1</td>
<td>0.000</td>
<td>0.000</td>
<td>30.349ms</td>
<td>680.</td>
<td>14.621%</td>
<td></td>
</tr>
<tr>
<td>MO_r_sieve@MEM2</td>
<td>0.000</td>
<td>0.000</td>
<td>113.519ms</td>
<td>410.</td>
<td>8.917%</td>
<td></td>
</tr>
</tbody>
</table>
```
The command **Trace.STATistic.TASKFunc** can be used, if the trace information output by the processor includes data write cycles.

The measurement is based on marking:

- All function entries and exits
- All task switches

**Mark the Functions**

Here for most CPUs it isn’t possible to perform a selective trace. In order to perform the measurement set Alpha breakpoints implemented as software breakpoints to all function entries and Beta breakpoints implemented as software breakpoints to all function exits.

These breakpoints can be set automatically with the command **Break.SetFunc**, if HLL-functions are loaded. Assembler functions or e.g. loops within HLL-functions can also be marked manually.

The entry points of interrupt routines should be marked with Alpha and Charly breakpoints. Thus the analysis ignores interrupt function times and takes care about double fetches caused by interrupted programs.

If all interrupt routines are located in a specific memory range or within a specific HLL module the interrupt routines can be marked automatically by using the command **Break.SetFunction <range>|<module>/INTR**.

If there is a name convention for interrupt routines use the **sYmbol.ForEach** command to mark the interrupt routines.

**Mark the Task Switch (TASK configured)**

If a **TASK** configuration is used, no extra preparations are necessary. If the processor allows to restrict the output to specific data write cycles, it is recommended to restrict the output to write cycles to `task.config(magic)+`(task.config(magicsize)-1) since more information can be sampled into the trace buffer.

**Mark the Task Switch (TASK not configured)**

If no TASK configuration is used, set an Alpha, Beta and Charly breakpoint implemented as software breakpoint to “current_task” (assuming “current_task” to be the variable which holds the ID of the current running task). The data present at this cycle will be used in the statistic to identify the task.

**Break.Set v.range(current_task) /Alpha /Beta /Charly /Soft**

If the time spent in the kernel should be excluded from the measurement the entry of the kernel has to be marked with a Charly breakpoint implemented as software breakpoint and exit of the kernel have to be marked with a Charly and a Beta breakpoint implemented as software breakpoint.
Recording

The complete program flow is recorded and the records of interest (function entries, function exits, interrupt entries and interrupt exits) are marked in the trace. The trace memory should be cleared before starting the recording. This is automatically done if Analyzer.AutoInit is ON.

Check the trace by viewing the record with <trace>.List MARK DEFAULT.

Procedure for Measurement for TRACE32-ICE and TRACE32-FIRE

The measurement is based on selective tracing of:

- all function entries and exits
- all task switches

Mark the Functions

In order to perform this measurement all function entries have to be marked with an Alpha breakpoint, and all function exits have to be marked with a Beta breakpoint. These breakpoints can be set automatically with the command Break.SetFunc, if HLL-functions are loaded. Assembler functions or e.g. loops within HLL-functions can also be marked manually.

The entry points of interrupt routines should be marked with Alpha and Charly (or Beta on ECC8) breakpoints. Thus the analysis ignores interrupt function times and takes care about double fetches caused by interrupted programs.

If all interrupt routines are located in a specific memory range or within a specific HLL module the interrupt routines can be marked automatically by using the command Break.SetFunction <range>|<module> /INTR.

If there is a name convention for interrupt routines use the sYmbol.ForEach command to mark the interrupt routines.

Mark the Task Switch (TASK configured)

If the TASK command is configured, the magic word (the memory location that defines which task is running) has to be marked with an AlphaBreak.

Mark the Task Switch (without TASK configured)

If the TASK command is not configured, the access cycle identifying the next running task has to be traced and marked in the trace by 'A'+ 'B'+C' markers. The data present at this cycle will be displayed in the statistic to identify the task. If the time spent in the kernel should be excluded from the measurement the entry and exit of the kernel have to be traced and marked with 'C' and 'C'+B'.
Programming of the Trigger Unit

The trigger unit is programmed to record accesses to ALPHA, BETA or CHARLY breakpoints and mark the record in the trace buffer as follows:

- Function entries are marked with an A marker.
- Function exits are marked with a B marker.
- Interrupt entries are marked with an A and a C marker.
- Interrupt exits are marked with a B marker.
- Task switches are marked with an A, B and C marker (without TASK configuration only)
- Entries to the kernel are marked with a C marker (without TASK configuration only)
- Exits of the kernel are marked with a B and C marker (without TASK configuration only)

Analyzer.ReProgram
{
    Sample.Enable IF AlphaBreak||BetaBreak||CharlyBreak
    Mark.A IF AlphaBreak
    Mark.B IF BetaBreak
    Mark.C IF CharlyBreak
}

Recording

The trace memory should be cleared before starting the recording. Check the trace by viewing the record with <trace>.List MARK List.TASK DEFault.

A display of the nesting is possible with the command <trace>.List FUNC FUNCR List.TASK (this command uses the same strategy to determine function entries and exits). If the nesting of the functions is not correct (each function entry must have its corresponding function exit), the results will be incorrect and an error message will be displayed in the statistic window. If single records are wrong, the commands <trace>.STATistic.Prefetch or <trace>.STATistic.Ignore can be used to remove them from the measurement.

The command <trace>.STATistic.TASKTREE can also be used to verify the correct function nesting. The command <trace>.Chart.TASKFunc displays a time chart of the function usage.
## Options

<table>
<thead>
<tr>
<th>Basic &lt;options&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE</strong></td>
<td>Displays trace memory contents loaded with <code>Trace.FILE</code>.</td>
</tr>
<tr>
<td><strong>FlowTrace</strong></td>
<td>The trace works as flow trace. This option is usually not required.</td>
</tr>
<tr>
<td><strong>BusTrace</strong></td>
<td>The trace works as a bus trace. This option is usually not required.</td>
</tr>
<tr>
<td><strong>Accumulate</strong></td>
<td>By default only the current trace contents is analyzed by the statistic functions. The option <code>/Accumulate</code> allows to add the current trace contents to the already displayed results.</td>
</tr>
<tr>
<td><strong>INCremental</strong></td>
<td>Intermediate results are displayed while the TRACE32 software analyses the trace contents (default).</td>
</tr>
<tr>
<td><strong>FULL</strong></td>
<td>The result is displayed after the TRACE32 software finished the analysis.</td>
</tr>
<tr>
<td><strong>Track</strong></td>
<td>Track the <code>Trace.STATistic</code> window with other trace list windows (tracking to record number or time possible).</td>
</tr>
<tr>
<td><strong>NoMerge</strong></td>
<td>(For diagnosis purpose only).</td>
</tr>
</tbody>
</table>

| **CTS**        | Use Context Tracking System to fill trace gaps and then perform the `<trace>.STATistic.TASKFunc` command. This is only useful for the ARM-ETM, SH4 and NEXUS. |
| **INTR**       | The INTR option includes interrupt functions in the measurement like function calls. Without this options the time spent in interrupt functions is taken out of the measurement. |
| **INTRTASK**   | The INTRTASK option takes interrupts out of the measurement, but displays the interrupt times for each task in a separate line. |
List items

The `<list_items>` can be arranged by pushing the **Config** button in the `<trace>.STATistic.FUNC` window.

<table>
<thead>
<tr>
<th>NAME</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Displays the function name. Interrupt functions are marked with ➔.</td>
</tr>
<tr>
<td>TIme</td>
<td>Time between entry and exit summed over the complete measurement time. By default the time spent in interrupt routines is taken out of the measurement.</td>
</tr>
<tr>
<td>MIN</td>
<td>Shortest time from function entry to exit.</td>
</tr>
<tr>
<td>MAX</td>
<td>Longest time from function entry to exit.</td>
</tr>
<tr>
<td>AVeRage</td>
<td>Average time from function entry to exit.</td>
</tr>
<tr>
<td>Internal</td>
<td>Time spent within the function (without called sub functions).</td>
</tr>
<tr>
<td>IAVeRage</td>
<td>Average time spent in the function (without called sub functions).</td>
</tr>
<tr>
<td>IMIN</td>
<td>Shortest time spent in the function (without called sub functions).</td>
</tr>
<tr>
<td>IMAX</td>
<td>Longest time spent in the function (without called sub functions).</td>
</tr>
<tr>
<td>External</td>
<td>Time spent within sub functions.</td>
</tr>
<tr>
<td>EAVeRage</td>
<td>Average time spent within sub functions.</td>
</tr>
<tr>
<td>EMIN</td>
<td>Shortest time spent within sub functions.</td>
</tr>
<tr>
<td>EMAX</td>
<td>Longest time spent within sub functions.</td>
</tr>
<tr>
<td>MAXIntr</td>
<td>Maximum time the function was interrupted by an interrupt.</td>
</tr>
<tr>
<td>MAXTask</td>
<td>Time in interrupt routines for each task.</td>
</tr>
<tr>
<td>Count</td>
<td>Number of calls of the function. A negative value in brackets shows the number of not complete entries or exits to the function (i.e. when the measurement is stopped).</td>
</tr>
<tr>
<td>Ratio</td>
<td>Ratio of time spent within the function over the complete measurement time (without called subroutines).</td>
</tr>
<tr>
<td>BAR.log, BAR.LIN</td>
<td>Graphical display of the ratio (linear or logarithmic).</td>
</tr>
<tr>
<td>TRatio</td>
<td>Ratio of time spent within the function over the complete measurement time (called subroutines included).</td>
</tr>
<tr>
<td>TBAR.log, TBAR.LIN</td>
<td>Graphical display of the ratio (linear or logarithmic).</td>
</tr>
</tbody>
</table>
### Formats

**LEN** `<size>`
Specifies the width of non numeric fields (e.g. symbols)

**TimeAuto**
Adapt the time display. (default)

**TimeFixed**
Display all time information in seconds.

**See also**
- `<trace>.STATistic.TASK`
- `<trace>.STATistic`
- 'Release Information' in 'Release History'

---

**<trace>.STATistic.TASKINTR**

**ISR2 statistic (ORTI)**

**Format:**

```
<trace>.STATistic.TASKINTR [%<format>] [<list_item> …] [/<option>]
```

Displays an ORTI based ISR2 runtime statistic.

For parameters see `<trace>.STATistic.Func`

**See also**
- `<trace>.STATistic`
Task analysis with kernel markers (flat)

The command `Trace.STATistic.TASKKernel` refines the command `Trace.STATistic.TASK` for RTOS systems that don’t assign a task ID to the kernel. In such a case no task run-time is calculated for the kernel if the command `Trace.STATistic.TASK` is used.

If the TRACE32 TASK awareness was configured, TRACE32 implies that the kernel writes the identifier of the current task to the address `TASK.CONFIG(magic)`.

```
PRINT TASK.CONFIG(magic)
```
Measurement performed by Trace.STATistic.TASK (no task ID for the kernel):

The refined measurement of Trace.STATistic.TASKKernel requires that the kernel entries and kernel exits are marked by the command `sYmbol.NEW.MARKER`.

```
sYmbol.NEW.MARKER KENTRY os_prologue ; mark the address os_prologue as kernel entry point
sYmbol.NEW.MARKER KEXIT os_epilogue ; mark the address os_epilogue as kernel exit point
sYmbol.List.MARKER ; list all markers
```
Advanced example for RTOS RTXC on a StarCore CPU:

```c
; mark all interrupt service routines as kernel entries
sYmbol.ForEach "sYmbol.NEW.MARKER KENTRY *" "_isr_*"

; mark all RTE instructions in the specified program range as kernel exit
Data.Find P:RTXCProlog--P:RTXCProlog_end %Word 0x9f73
WHILE FOUND()
{
    sYmbol.NEW.MARKER KEXIT P:TRACK.ADDRESS()
    Data.Find
}
sYmbol.List.MARKER
```

If the processor allows to restrict the trace information output to the program flow and specific write access, it is recommended to restrict the output to the program flow plus write cycles to `task.config(magic)`, since more information can be recorded into the trace buffer.

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

Go

Break

Trace.STATistic.TASKKernel
```

See also
- `<trace>.STATistic`
- 'Release Information' in 'Release History'
General Commands Reference Guide T

**<trace>.STATistic.TASKORINTERRUPT**

Statistic of interrupts and tasks

Format:  

```
<trace>.STATistic.TASKORINTERRUPT
```

See also

- `<trace>.STATistic`
- 'Release Information' in 'Release History'

**<trace>.STATistic.TASKSRV**

Analysis of time in OS service routines

Format:  

```
<trace>.STATistic.TASKSRV [%<format>] [items] [option]
```

<option>:  

```
FILE
```

The time spent in OS service routines and the number of calls is measured.

This feature is only available, if an OSEK/ORTI system is used, and if the OS Awareness is configured with the  

**TASK.ORTI** command.

See also

- `<trace>.STATistic`
- 'OS-Aware Tracing - Single-Core and AMP' in 'AURIX Trace Training'
- 'OS-Aware Tracing - SMP Systems' in 'AURIX Trace Training'
- 'Release Information' in 'Release History'
**Format:**
<trace>.STATistic.TASKState [%<format>] [<items> …] [<option>]

**<option>:**
- FILE
- FlowTrace | BusTrace
- Accumulate
- INCremental | FULL
- Sort <item>
- Track

**<list_items>:**
- Time [.all | .UND | .RUN | RDY | WAIT | SUSP]
- MAX [.all | .UND | .RUN | RDY | WAIT | SUSP]
- AVErage [.all | .UND | .RUN | RDY | WAIT | SUSP]
- Count [.all | .UND | .RUN | RDY | WAIT | SUSP]
- Ratio
- BAR.log | BAR.LIN

The time tasks spent in different states is measured.

Before using that function the task state transitions must be sampled by the trace. This feature is highly dependent on the used RTOS kernel, and needs the TASK to be configured. Please see kernel specific “OS Awareness Manuals” manuals for more information.
### E::Trace.STATistic.TASKState

**Total:** 1.954s

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
<th>Run</th>
<th>Rdy</th>
<th>Wait</th>
<th>Susp</th>
</tr>
</thead>
<tbody>
<tr>
<td>(root)</td>
<td>0.000</td>
<td>0.000</td>
<td>1.954s</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>IDLE</td>
<td>252.049ms</td>
<td>78.787ms</td>
<td>1.623s</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>IO1</td>
<td>10.340ms</td>
<td>100.968ms</td>
<td>345.670ms</td>
<td>653.006ms</td>
<td>844.202ms</td>
</tr>
<tr>
<td>IO2</td>
<td>11.186ms</td>
<td>247.481ms</td>
<td>153.662ms</td>
<td>724.730ms</td>
<td>817.127ms</td>
</tr>
<tr>
<td>MEM1</td>
<td>50.364ms</td>
<td>389.512ms</td>
<td>499.453ms</td>
<td>1.014s</td>
<td>0.000</td>
</tr>
<tr>
<td>MEM2</td>
<td>0.000</td>
<td>454.146ms</td>
<td>1.209s</td>
<td>0.000</td>
<td>290.577ms</td>
</tr>
<tr>
<td>MSG</td>
<td>420.354ms</td>
<td>1.654ms</td>
<td>132.875ms</td>
<td>0.000</td>
<td>1.532s</td>
</tr>
<tr>
<td>SINK</td>
<td>20.342ms</td>
<td>371.914ms</td>
<td>341.089ms</td>
<td>1.220s</td>
<td>0.000</td>
</tr>
<tr>
<td>SRCE</td>
<td>27.991ms</td>
<td>303.206ms</td>
<td>7.734ms</td>
<td>1.615s</td>
<td>0.000</td>
</tr>
</tbody>
</table>

### E::Trace.STATistic.TASKState

**Total:** 1.954s

<table>
<thead>
<tr>
<th>Task</th>
<th>Max</th>
<th>Run</th>
<th>Rdy</th>
<th>Wait</th>
<th>Susp</th>
</tr>
</thead>
<tbody>
<tr>
<td>(root)</td>
<td>0.000</td>
<td>0.000</td>
<td>1.954s</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>IDLE</td>
<td>252.049ms</td>
<td>23.346ms</td>
<td>426.751ms</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>IO1</td>
<td>10.337ms</td>
<td>4.598ms</td>
<td>27.738ms</td>
<td>50.386ms</td>
<td>64.353ms</td>
</tr>
<tr>
<td>IO2</td>
<td>11.181ms</td>
<td>11.596ms</td>
<td>17.166ms</td>
<td>47.320ms</td>
<td>66.393ms</td>
</tr>
<tr>
<td>MEM1</td>
<td>50.360ms</td>
<td>22.228ms</td>
<td>29.950ms</td>
<td>87.455ms</td>
<td>0.000</td>
</tr>
<tr>
<td>MEM2</td>
<td>0.000</td>
<td>35.941ms</td>
<td>78.177ms</td>
<td>0.000</td>
<td>68.315ms</td>
</tr>
<tr>
<td>MSG</td>
<td>420.328ms</td>
<td>805.625us</td>
<td>42.000us</td>
<td>0.000</td>
<td>1.229s</td>
</tr>
<tr>
<td>SINK</td>
<td>20.338ms</td>
<td>8.103ms</td>
<td>11.523ms</td>
<td>83.515ms</td>
<td>0.000</td>
</tr>
<tr>
<td>SRCE</td>
<td>27.991ms</td>
<td>3.737ms</td>
<td>1.085ms</td>
<td>92.603ms</td>
<td>0.000</td>
</tr>
</tbody>
</table>

### E::Trace.STATistic.TASKState

**Total:** 1.954s

<table>
<thead>
<tr>
<th>Task</th>
<th>Avr</th>
<th>Run</th>
<th>Rdy</th>
<th>Wait</th>
<th>Susp</th>
</tr>
</thead>
<tbody>
<tr>
<td>(root)</td>
<td>0.000</td>
<td>0.000</td>
<td>1.954s</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>IDLE</td>
<td>252.049ms</td>
<td>7.162ms</td>
<td>147.577ms</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>IO1</td>
<td>10.340ms</td>
<td>1.628ms</td>
<td>4.215ms</td>
<td>31.095ms</td>
<td>20.100ms</td>
</tr>
<tr>
<td>IO2</td>
<td>11.186ms</td>
<td>4.057ms</td>
<td>1.873ms</td>
<td>34.510ms</td>
<td>19.929ms</td>
</tr>
<tr>
<td>MEM1</td>
<td>50.364ms</td>
<td>9.058ms</td>
<td>8.055ms</td>
<td>50.742ms</td>
<td>0.000</td>
</tr>
<tr>
<td>MEM2</td>
<td>0.000</td>
<td>8.410ms</td>
<td>22.397ms</td>
<td>0.000</td>
<td>20.755ms</td>
</tr>
<tr>
<td>MSG</td>
<td>420.354ms</td>
<td>413.531us</td>
<td>33.218us</td>
<td>0.000</td>
<td>306.409ms</td>
</tr>
<tr>
<td>SINK</td>
<td>20.342ms</td>
<td>4.375ms</td>
<td>2.842ms</td>
<td>33.912ms</td>
<td>0.000</td>
</tr>
<tr>
<td>SRCE</td>
<td>27.991ms</td>
<td>3.002ms</td>
<td>54.854us</td>
<td>19.941ms</td>
<td>0.000</td>
</tr>
</tbody>
</table>

### E::Trace.STATistic.TASKState

**Total:** 1.954s

<table>
<thead>
<tr>
<th>Task</th>
<th>C.</th>
<th>Run</th>
<th>Rdy</th>
<th>Wait</th>
<th>Susp</th>
<th>Ratio 1%</th>
<th>Ratio 2%</th>
</tr>
</thead>
<tbody>
<tr>
<td>(root)</td>
<td>1.</td>
<td>0.</td>
<td>1.</td>
<td>0.</td>
<td>0.</td>
<td>0.000%</td>
<td></td>
</tr>
<tr>
<td>IDLE</td>
<td>1.</td>
<td>11.</td>
<td>11.</td>
<td>0.</td>
<td>0.</td>
<td>4.031%</td>
<td></td>
</tr>
<tr>
<td>IO1</td>
<td>1.</td>
<td>62.</td>
<td>82.</td>
<td>21.</td>
<td>42.</td>
<td>5.166%</td>
<td></td>
</tr>
<tr>
<td>IO2</td>
<td>1.</td>
<td>61.</td>
<td>82.</td>
<td>21.</td>
<td>41.</td>
<td>12.664%</td>
<td></td>
</tr>
<tr>
<td>MEM1</td>
<td>1.</td>
<td>43.</td>
<td>62.</td>
<td>20.</td>
<td>0.</td>
<td>19.932%</td>
<td></td>
</tr>
<tr>
<td>MEM2</td>
<td>1.</td>
<td>54.</td>
<td>54.</td>
<td>0.</td>
<td>14.</td>
<td>23.239%</td>
<td></td>
</tr>
<tr>
<td>MSG</td>
<td>1.</td>
<td>4.</td>
<td>4.</td>
<td>0.</td>
<td>5.</td>
<td>0.084%</td>
<td></td>
</tr>
<tr>
<td>SINK</td>
<td>1.</td>
<td>85.</td>
<td>120.</td>
<td>36.</td>
<td>0.</td>
<td>19.031%</td>
<td></td>
</tr>
<tr>
<td>SRCE</td>
<td>1.</td>
<td>101.</td>
<td>141.</td>
<td>81.</td>
<td>0.</td>
<td>15.515%</td>
<td></td>
</tr>
<tr>
<td>Basic &lt;options&gt;</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILE</td>
<td>Displays trace memory contents loaded with <code>Trace.FILE</code>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FlowTrace</td>
<td>The trace works as flow trace. This option is usually not required.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BusTrace</td>
<td>The trace works as a bus trace. This option is usually not required.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulate</td>
<td>By default only the current trace contents is analyzed by the statistic functions. The option <code>/Accumulate</code> allows to add the current trace contents to the already displayed results.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INCremental</td>
<td>Intermediate results are displayed while the TRACE32 software analyses the trace contents (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FULL</td>
<td>The result is displayed after the TRACE32 software finished the analysis.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Track</td>
<td>Track the <code>Trace.STATistic</code> window with other trace list windows (tracking to record number or time possible).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoMerge</td>
<td>(For diagnosis purpose only).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### List items

<table>
<thead>
<tr>
<th><strong>&lt;state&gt;</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>The total time the task was in this state.</td>
</tr>
<tr>
<td>MAX</td>
<td>The maximum time the task was in this state.</td>
</tr>
<tr>
<td>AVerage</td>
<td>The average time the task was in this state. &lt;br&gt;<strong>NOTE:</strong> This value can be wrong if intermediate states exist.</td>
</tr>
<tr>
<td>Count</td>
<td>The number of times a state was entered. &lt;br&gt;<strong>NOTE:</strong> This value can be wrong if intermediate states exist.</td>
</tr>
<tr>
<td>Ratio</td>
<td>The ratio of CPU runtime consumed by this task.</td>
</tr>
<tr>
<td>BAR.log, BAR.LIN</td>
<td>Graphical display of ratio column.</td>
</tr>
</tbody>
</table>

Possible `<state>` are: **UND**efined, **RUN**ing, **ReaDY**, **WAIT**ing, **SUSP**ended.

### See also
- `<trace>.STATistic`
- 'Release Information' in 'Release History'
The results of this command shows a graphical tree of the function nesting. The measurement is done like for the command `Analyzer.STATistic.TASKFunc`.

For a description of `<format>`, `<list_items>`, and `<option>`, refer to `trace>.STATistic.TASKFunc`.
### <trace>.STATistic.TASKTREE

Statistic of interrupts, task-related

<table>
<thead>
<tr>
<th>Range tree</th>
<th>Time</th>
<th>Min</th>
<th>Max</th>
<th>Avr</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>786.129ms</td>
<td>0.000</td>
<td>786.129ms</td>
<td>786.129ms</td>
</tr>
<tr>
<td>IDLE</td>
<td>35.053ms</td>
<td>0.000</td>
<td>35.053ms</td>
<td>35.053ms</td>
</tr>
<tr>
<td>IO1</td>
<td>72.308ms</td>
<td>0.000</td>
<td>72.308ms</td>
<td>72.308ms</td>
</tr>
<tr>
<td>IO2</td>
<td>22.760ms</td>
<td>813.000us</td>
<td>1.287ms</td>
<td>842.989us</td>
</tr>
<tr>
<td>MEM1</td>
<td>437.910ms</td>
<td>0.000</td>
<td>437.910ms</td>
<td>437.910ms</td>
</tr>
<tr>
<td>MEM2</td>
<td>555.178ms</td>
<td>0.000</td>
<td>555.178ms</td>
<td>555.178ms</td>
</tr>
</tbody>
</table>

### <trace>.STATistic.TASKVSINTERRUPT

ISR2 statistic (ORTI), task related

Format:  

<trace>.STATistic.TASKVSINTERRUPT

Displays an ORTI based ISR2 runtime statistic against task runtimes.

### <trace>.STATistic.TASKVSINTR

See also
- <trace>.STATistic
- 'Release Information' in 'Release History'
The results of this command shows a graphical tree of the function nesting. The measurement is done like for the command `Analyzer.STATistic.TASKFunc`.
<trace>.STATistic.Use

Use records

Format: 

<trace>.STATistic.Use [ <record> | <range> ] [ <options> ]

<option>: FILE

The specified record(s) are used for performance and nesting analysis. This command can be used, when some records should be used, which are ignored due to the <trace>.STATistic.Ignore or <trace>.STATistic.PreFetch commands.

See also

■ <trace>.STATistic

▲ 'Release Information' in 'Release History'

<trace>.STATistic.Var

Statistic of variable accesses

Format: 

<trace>.STATistic.Var [ <trace_area> ] [ <option> ]

The command provides a graphical chart of variable accesses.
Example:

; Display a statistic of all variable accesses:
Trace.STATistic.Var /Filter sYmbol mstatic1 /Filter CYcle Write

; Display a statistic of write accesses to the mstatic1 variable
Trace.STATistic.Var /Filter sYmbol mstatic1 /Filter CYcle Write

See also
- <trace>.STATistic
- 'Release Information' in 'Release History'
### <trace>.STREAMCompression

Select compression mode for streaming

<table>
<thead>
<tr>
<th>Format:</th>
<th>&lt;trace&gt;.STREAMCompression OFF</th>
<th>LOW</th>
<th>MID</th>
<th>HIGH</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>Trace information is streamed uncompressed and saved uncompressed to file on hard-disk (diagnose purpose only).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOW (default)</td>
<td>Trace information is streamed compressed (hardware compression for all ETMv3 and higher, software compression otherwise). Trace information is saved to file on hard-disk as received.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MID</td>
<td>Trace information is streamed compressed (hardware compression for all ETMv3s and higher, software compression otherwise). Trace information is zipped before it is saved to file on hard-disk.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HIGH</td>
<td>Trace information is streamed compressed (hardware compression for all ETMv3s and higher, software compression otherwise). Trace information is zipped very compact before it is saved to file on hard-disk.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

```plaintext
Trace.STREAMCompression LOW
Trace.Mode STREAM
```

**See also**

- <trace>.STREAMFILE
- <trace>.STREAMFileLimit
- <trace>.STREAMLOAD
- <trace>.STREAMSAVE
- CIProbe
**<trace>.STREAMFILE**

**Specify temporary streaming file path**

<table>
<thead>
<tr>
<th>Format:</th>
<th>&lt;trace&gt;.STREAMFILE &lt;file&gt;</th>
</tr>
</thead>
</table>

Set the path and file name for the temporary stream file. By default, TRACE32 will choose the standard TEMP folder set in the operating system. The only intention of this command is to select a different path/drive, e.g. a high-capacity drive dedicated for this use case.

The contents of the streaming file are in a proprietary format and not intended for use in external applications. Also it is not possible to use this file for later analysis.

In order to store a trace recording for later analysis, either use **Trace.SAVE** or **Trace.STREAMSAVE**. In order to export trace data for use in external applications, see **Trace.EXPORT**.

**Example:**

```
Trace.STREAMFILE d:\temp\mystream.t32 ; specify the location for your streaming file
Trace.Mode STREAM ; select the trace mode STREAM
```

**See also**

- <trace>.STREAMFileLimit
- <trace>.STREAMLOAD
- CIProbe
- <trace>.STREAMCompression
- <trace>.STREAMSAVE
Sets the maximum size allowed for a streaming file. If the maximum size is exceeded, streaming is stopped.

The limit value is given in bytes and can have a positive or negative sign:

- Positive value: The maximum size of the streaming file in bytes
- Negative value: Specifies the amount of space to leave on the disk before stopping streaming. The maximum file size is calculated based on the amount of available disk space at the time of starting streaming.

The default setting is -1.000.000.000 i.e. stops streaming when less than a GB of space is left on the target storage medium.

NOTE: The file limit and the name of the streaming file must be set before starting streaming. Later changes are not taken into account.

Slow trace ports allow high compression. Fast trace ports allow no compression.

Example:

```
Trace.STREAMCompression LOW ; select software compression
; mode LOW for streaming file
Trace.Mode STREAM ; select the trace mode STREAM
```

See also
- `<trace>.STREAMFILE`
- `<trace>.STREAMLOAD`
- `<trace>.STREAMCompression`
- `<trace>.STREAMSAVE`
Load **streaming file** to TRACE32. In order to display decompressed trace information the target state at the recording time has to be reconstructed within TRACE32. This can be very complicated especially if a software with an operating system that uses dynamic memory management to handle processes/tasks (e.g. Linux) is used.

**Example 1**: Reconstruction of the target state at the recording time for a bar metal Cortex-R4 application.

```plaintext
; specify the target CPU
SYSTEM.CPU TMS570PSFC61

SYStem.Option BigEndian ON

SYStem.Up

; specify ETM setting valid at the recording time
ETM.PortSize 16.
ETM.PortMode Bypass
ETM.DataTrace OFF
ETM.ContextID OFF
ETM.ON

; load source code and debug information
Data.LOAD.ELF I:\EVB\arm\tms570psfc61\demo.axf

; load saved streaming file
Trace.STREAMLOAD C:\T32_ARM\r4_max.sad

Trace.List
```
Example 2: Reconstruction of the target state at the recording time for NEXUS Power Architecture:

; specify the target CPU
SYStem.CPU MPC5646C

; specify the NEXUS setting valid at the recording time
NEXUS.PortSize MDO12
NEXUS.PortMode 1/2
NEXUS.BTM ON
NEXUS.HTM ON
NEXUS.PTCM BL_HTM ON
NEXUS.ON

SYStem.Up

; mapping logical to physical address is 1:1

; load source code and debug information
Data.LOAD.Elf J:\AND\PPC5XXX_OTM\im02_bf1x.elf

; load the OS Awareness
TASK.ORTI J:\AND\PPC5XXX_OTM\im02_bf1x.ort

; load saved streaming file
Trace.STREAMLOAD J:\AND\PPC5XXX_OTM\my_stream

Trace.List

See also
- <trace>.STREAMCompression
- <trace>.STREAMFileLimit
- <trace>.STREAMFILE
- <trace>.STREAMSAVE
Save the **streaming file**. Use **Trace.STREAMLOAD** to load the file for analysis.

The contents of the streaming file are in a proprietary format and not intended for use in external applications. Also it is not possible to use this file for later analysis.

In order to store a trace recording for later analysis, either use **Trace.SAVE** or **Trace.STREAMSAVE**. In order to export trace data for use in external applications, see **Trace.EXPORT**.

| <file> | The default extension for the streaming file is *.sad. |

**See also**
- **<trace>.STREAMCompression**
- **<trace>.STREAMFileLimit**
- **<trace>.SAVE**
- **<trace>.STREAMFILE**
- **<trace>.STREAMLOAD**

### <trace>.TCount

*Set trigger counter*

**Format:**

\[
\text{Integrator.TCount [<value>]} \quad \text{[value]: 0. \ldots 16777215.} \\
\]

Sets the number of trigger events that will be ignored by the trace or logic analyzer, before a trigger event ends the recording (state: break). A counter value zero means that the recording stops immediately after the first trigger. A value of 1 halts the recording at the second trigger event, and so on.

**Trigger Signal**

<table>
<thead>
<tr>
<th>Trigger Counter = 0</th>
<th>Trigger Counter = 1</th>
<th>Trigger Counter = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selects the delay time between trigger point and break (end of recording). Use this command in order to record events that occurred after the trigger point.

The trigger delay may also be defined in percent of the trace buffer size. The delay can be larger (up to 10x) than the total trace buffer size. Logic analyzers also support setting a delay time.

Analyzer.TDelay 40% ; trigger delay is 40% of trace depth.

Selects the delay time between trigger point and break of the port analyzer. The time can be larger than the time for a full sample of the analyzer. The trigger delay time may be defined in percent relating to the total trace time.

```
<table>
<thead>
<tr>
<th>ARM</th>
<th>Pretrigg.Delay</th>
<th>Trigger point</th>
<th>TDelay time</th>
<th>Break</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Trigger system active

Sampling

Format: `<trace>.TDelay <time> | <cycles> | <percent>%`

- `<time>`: 0 … 200.s
- `<percent>`: 0 … 1000%
- `<cycles>`: 0 … 4000000000.

©1989-2020 Lauterbach GmbH
With a mouse click to the corresponding area in the port analyzer state window this command can be executed too.

<table>
<thead>
<tr>
<th>Port.TDelay 10.ms</th>
<th>; the trigger delay is 10 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port.TDelay 50%</td>
<td>; the trigger point is in the mid of the trace</td>
</tr>
<tr>
<td></td>
<td>; memory</td>
</tr>
<tr>
<td>Port.TDelay 99%</td>
<td>; the trigger point is at the beginning to the trace</td>
</tr>
<tr>
<td></td>
<td>; memory</td>
</tr>
<tr>
<td>Port.TDelay 200</td>
<td>; the trigger point 200 record before end of trace</td>
</tr>
</tbody>
</table>

**Trigger delay**

- **0%** Trigger point
- **25%** Trigger point
- **50%** Trigger point
- **75%** Trigger point
- **100%** Trigger point
- **200%** Trigger point

See also

- `<trace>.state`
- `Trace`
By default the trace line termination of the preprocessor is used during a trace capture. Undefinable FLOWERRORs may occur if the output drivers of the CPU are not strong enough. In this case it is recommended to switch the trace line termination OFF.

See also

- `<trace>.state`
- `Trace`
The command **Trace.TestFocus** tests the recording at a high-speed trace port.

The command **Trace.TestFocus** can be used if:

- **The program execution is stopped.**
  
  To test the trace port, the test pattern generator of the trace port is used if available. Otherwise, a test program is loaded and started by TRACE32.

- **The program execution and the trace recording is running.**
  
  Testing the trace port while the application program is running might be helpful to detect trace port problems caused by the application program.

  The trace data from the application program are used to test the trace port. Here a reduced test scenario is processed that checks the correctness of the program flow recording and for short-circuits between the trace port lines. This test requires that the program code is loaded to the virtual memory.

  ```plaintext
  Data.LOAD.Elf arm.elf /PlusVM ; Load the application code
  ; to the target memories and
  ; to the virtual memory of
  ; TRACE32
  ```

- **The program execution is running and the trace recording is stopped.**
  
  To test the trace port, the test pattern generator of the trace port is used if available. Otherwise, the trace data from the application program are used.

  If a test program is used, TRACE32 attempts to load the test program to the memory addressed by the PC or the stack pointer. It is also possible to define an `<address_range>` for the test program.

  ```plaintext
  Trace.TestFocus ; start trace port test
  Trace.AutoFocus 0x24000000++0xffff ; start the test and load
  ; test program to address
  ; 0x24000000
  ```
If TRACE32 is unable to load the test program the following error message is displayed: “Don’t know where to execute the test code”.

By default, the original RAM content is restored after the trace port test and the trace recording is deleted.

**Accumulate**

If the application program varies the CPU clock frequency, this affects also the trace port. In such a case it is recommended to overlay the test results for all relevant CPU clock frequencies by using the option `/Accumulate`.

**Config**

Allows to define a RAM address range for the download of the test program.

**KEEP**

After a trace port test the trace is cleared and any loaded test program is removed from the target RAM.

With the option `/KEEP`, the test trace is not cleared and can be viewed with the `Trace.List` command. If a test program was loaded by TRACE32, it also remains in the target RAM.

**ALTERNATE**

If the trace port provides a test pattern generator, it is always used for the test. The option `/ALTERNATE` forces TRACE32 to use its own test program.

**NoTraceControl**

Informs the TRACE32 software that the trace control signal is not available on the trace connector.

```plaintext
; advise the command Trace.TestFocus to download the test program ; always to the address range 0x24000000++0xfff
Trace.TestFocus 0x24000000++0xfff /Config
```

The result of the command `Trace.TestFocus` can be processed in a PRACTICE script as follows:

```plaintext
Trace.TestFocus

IF FOUND()
    PRINT %ERROR "Trace port test failed"
ELSE
    PRINT "Trace port recording ok"
```
The **Trace.TestFocus** command calls the data eye finder for the current hardware configuration of a preprocessor with AUTOFOCUS technology and verifies the correctness of traced test data. In contrast to **Trace.AutoFocus**, the preprocessor configuration remains unchanged.

A complete trace port test executes the following steps:

1. **The data eye finder is called.** The source for the trace data for the test are the trace port’s pattern generator, a test program or the application program.

2. **When the eye finder is done, the test is started once again to verify the correctness of the trace recording.**

3. **The data eyes resulting from the <trace>.TestFocus command can be viewed in the <trace>.ShowFocus window.**

See also

- <trace>.TestFocusEye
- <trace>.ShowFocus
- <trace>.ShowFocusEye
- Trace
- AUTOFOCUS.FREQUENCY()

▲ ‘Release Information’ in ‘Release History’

---

**<trace>.TestFocusClockEye**

Scan clock eye

Format: `<trace>.TestFocusClockEye [address_range] [l<option>]`

`<option>`: Accumulate | Config | KEEP | ALTERNATE | Utilisation | NoTraceControl

Scans the clock eye. To view the result, use the command **Trace.ShowFocusClockEye**.

**NOTE:** The NEXUS AutoFocus adapter does not support this feature.

For a description of the options, see **Trace.TestFocus**.

See also

- <trace>.TestFocus
- <trace>.ShowFocus
- <trace>.ShowFocusEye
- <trace>.AutoFocus
- <trace>.ShowFocusClockEye
- <trace>.TestFocusEye
- AUTOFOCUS.OK()
<trace>.TestFocusEye

Debugger only

Format: 

<trace>.TestFocusEye [<address_range>] [/<option>]

<option>

Accumulate
Config
KEEP
ALTERNATE
NoTraceControl

Scans the data eye to determine the integrity of the electrical trace signals.

The command Trace.TestFocusEye starts an eye finder to test the quality of the trace signals, if a preprocessor with AUTOFOCUS technology is used. The test result can be displayed with the command Trace.ShowFocusEye. If the result shows that an individual trace signal has a significantly smaller data eye than other signals, the hardware layout should be checked to see if this signal shows any unusual features.

The test procedure and the options used by the command Trace.TestFocusEye are similar to the command Trace.TestFocus.

<option> 

For a description of the options, see Trace.TestFocus.

The command Trace.TestFocusEye can also be used with PowerTrace Serial. For this tool no additional parameters (e.g. <address_range>) or options are available.

See also

■ <trace>.AutoFocus
■ <trace>.ShowFocusClockEye
■ <trace>.state
■ <trace>.TestFocus

▲ 'Release Information’ in 'Release History'
The command `Trace.THreshold` can be used to optimize the threshold level for the trace lines sampled via a TRACE32-Preprocessor (e.g. ARM-ETM, OCDS Level 2, AUD ...). The optimization of the threshold level should result in less errors in the trace recording.

### VCC
The preprocessor and the TRACE32 software measure the VCC of the target. 1/2 VCC is then automatically used as the threshold level for the trace lines. The result is also displayed in the THreshold field of the `<trace>.state` window.

### CLOCK
The threshold level is changed until the duty cycle of the trace clock reaches a ratio of 1:1. This setting is only recommended if the trace clock has a duty cycle of 1:1. The result is displayed in the THreshold field of the `<trace>.state` window.

### `<level>`
The threshold level can be entered directly.

```
Trace.THreshold VCC
Trace.THreshold CLOCK
Trace.THreshold 1.6 ; the unit is Volt
```
Enhanced parameters for the Preprocessor for ARM-ETM with AUTOFOCUS:

\(<\text{clock}>\ <\text{data}>\)

For the Preprocessor for ARM-ETM with AUTOFOCUS different threshold levels can be defined for the clock and the data lines

```
Trace.THreshold 0.86 0.79  ; the unit is Volt
```

See also

- `<trace>.state`
- `Trace`
- `▲ 'Release Information' in 'Release History'`

**<trace>.TimeStamp** Configure timestamp usage of LOGGER trace

LOGGER only

```
Format:  LOGGER.TimeStamp OFF | Up | Down | Rate <rate>
```

Configure timestamps for the LOGGER trace. The LOGGER trace record format includes a timestamp field for up to 48 bit timestamps. The direction and rate information passed by this command is required to convert the timestamp into the time in seconds.

- **OFF** (default)  
  Disable timestamps. Use this setting if the LOGGER target code does not store timestamps in the LOGGER trace records. When this setting is used, the x-direction in chart views is the record number axis instead of the time axis.

- **Up**  
  Enable timestamp counter, counting upwards. Use this setting if the LOGGER target code stores timestamps in the LOGGER trace records and if the timestamp increments with each timer tick.

- **Down**  
  Enable timestamp counter, counting downwards. Use this setting if the LOGGER target code stores timestamps in the LOGGER trace records and if the timestamp decrements with each timer tick.

- **Rate <rate>**  
  Frequency of the timestamp in ticks per second.

- **AllCycles**  
  Set timestamp generation frequency.

  | [ON | OFF] | SH only |
  |---------|---------|
  | OFF (default): Generate a single timestamp for 6 trace cycles. |
  | ON: Generate dedicated timestamps for all trace cycles. |
Example: The timestamp used by the LOGGER target code increments at a rate of 16 million per second (16 MHz):

```
LOGGER.TimeStamp Up
LOGGER.TimeStamp Rate 16000000.
```

### `<trace>.Timing`

Waveform of trace buffer

<table>
<thead>
<tr>
<th>Format:</th>
<th><code>&lt;trace&gt;.Timing</code> <code>[&lt;record_range&gt;]</code> <code>[[&lt;items&gt;]]</code> <code>[&lt;options&gt;]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;option&gt;</code>:</td>
<td>FILE</td>
</tr>
<tr>
<td></td>
<td>Track</td>
</tr>
<tr>
<td></td>
<td>RecScale</td>
</tr>
<tr>
<td></td>
<td>TimeScale</td>
</tr>
<tr>
<td></td>
<td>TimeZero</td>
</tr>
<tr>
<td></td>
<td>TimeREF</td>
</tr>
</tbody>
</table>

Displays the trace memory contents like command `<trace>.List`, but in form of a timing display. As a default the external trigger channels are displayed.

- **FILE**
  Display trace memory contents loaded with `Trace.FILE`.

- **Track**
  The cursor in the `<trace>.Timing` window follows the cursor movement in other trace windows. Default is a time tracking. If no time information is available tracking to record number is performed.
  The zoom factor of the `<trace>.Timing` window is retained, even if the trace content changes.

- **RecScale**
  Display trace in fixed record raster. This is the default.

- **TimeScale**
  Display trace as true time display, time relative to the trigger point.

- **TimeZero**
  Display trace as true time display, time relative to zero.

- **TimeREF**
  Display trace as true time display, time relative to the reference point.

![Example waveform of trace buffer](image)
### Buttons

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Zoom In T</strong></td>
<td>Zooms in Trace by a factor of 2.</td>
</tr>
<tr>
<td><strong>Zoom Out T</strong></td>
<td>Zooms out Trace by a factor of 2.</td>
</tr>
<tr>
<td><strong>Zoom Full T</strong></td>
<td>Display the complete trace buffer in the window.</td>
</tr>
<tr>
<td><strong>Goto …</strong></td>
<td>Open an <code>&lt;trace&gt;.GOTO</code> dialog box.</td>
</tr>
<tr>
<td><strong>Find …</strong></td>
<td>Open an <code>&lt;trace&gt;.Find</code> dialog box.</td>
</tr>
<tr>
<td><strong>Set Ref</strong></td>
<td>Set an analyzer reference point to the current record.</td>
</tr>
<tr>
<td><strong>Set Zero</strong></td>
<td>Set the global time reference to the current record.</td>
</tr>
<tr>
<td><strong>View</strong></td>
<td>Display all information about the current record (<strong>trace&gt;.View</strong>).</td>
</tr>
<tr>
<td><strong>List</strong></td>
<td>Open an <code>&lt;trace&gt;.List</code> window.</td>
</tr>
</tbody>
</table>

### Examples:

```plaintext
; Open Port Analyzer timing window in standard display format
E::Port.Timing

; Open Port Analyzer timing window and display last file loaded with
; " Port.FILE <file>"
E::Port.Timing /FILE

; Open Port Analyzer timing window starting at record -100. in standard
; display format
E::Port.Timing -100. DEFault
```

### See also

- `<trace>.List`
- `<trace>.State`
- `<trace>.REF`
- `<trace>.View`
- `Analyzer.RECORD.ADDRESS()`
- `Analyzer.RECORD.OFFSET()`
- `Analyzer.REF()`

▲ 'Release Information' in 'Release History'
▲ 'Displaying the Trace' in 'Training FIRE Analyzer'
▲ 'Emulator Functions' in 'FIRE User's Guide'
<trace>.TMode

Select trigger mode

Format:  

\<trace>.TMode [High | Low | Rising | Falling]

Selects the trigger condition, edge or level trigger and the corresponding line polarity. The edge trigger is asynchronous and needs a minimum pulse width of 20 ns.

Example:

E::Port.TMode Rising  
E::Port.TSelect Port  
E::Port.Select Port.00 ; trigger on the rising edge of P.00  
E::Port.TDelay 100.us ; sample till 100.us after trigger

See also

- Trace

<trace>.TOut

Enable trigger output line (PowerIntegrator)

Format:

\[\text{Integrator.TOut ON | OFF}\]

Enables or disables the trigger output line of the integrator.

See also

- Trace
Pre-trigger delay

Format: `<trace>.TPreDelay [<time> | <percent> | DEFAULT]`

- `<percent>`: 0 … 1000%
- `<time>`: 0 … 200.s
- `<cycles>`: 0 … 4000000000.

Selects a delay time between the start of the recording and the release of the trigger system. This delay is very useful if previous events before a trigger point are of interest. The trigger event is immediately valid after the arming of the analyzer. If the port analyzer is slave to the emulator, usually no delay will be needed.

| DEFAULT | The trigger predelay is max. 10% of the selected trace depth. |

With a mouse click to the corresponding area in the port analyzer state window that command can be executed, too.

See also
- Trace

©1989-2020 Lauterbach GmbH
If the AutoFocus preprocessor is connected to a Single-Wire-Trace module (i.e. Trace.PortType SWV), then this command is used to specify the bit rate. The `<data_rate>` is then set to 1/(min-bit-size).

**See also**

- Trace
- 'Release Information' in 'Release History'

---

**<trace>.TraceCONNECT**

Select on-chip peripheral sink

**Format:**

```
<trace>.TraceCONNECT <component>
<trace>.TraceCONNECT NONE
<trace>.TraceCONNECT AUTO
```

Default: AUTO.

Selects the on-chip peripheral used as trace sink on the SoC.

**Example:** The two ETFs of an ARM CoreSight based SoC are selected as trace sink.

```
;note that the two approaches to select the first ETF are equivalent:
Onchip.TraceCONNECT ETF1 ; selects the ETF1 as onchip-trace sink
;or
Trace.METHOD Onchip
Trace.TraceCONNECT ETF1 ; selects the ETF1 as onchip-trace sink

;note that the two approaches to select the second ETF are equivalent:
Onchip.TraceCONNECT ETF2 ; selects the ETF2 as onchip-trace sink
;or
Trace.METHOD Onchip
Trace.TraceCONNECT ETF2 ; selects the ETF2 as onchip-trace sink
```
Sets the tracking record to the specified trace bookmark, time, or record number. The blue cursor moves to the specified destination in all Trace.* windows opened with the /Track option. All other Trace.* windows opened without the /Track option do not respond to the <trace>.TRACK command.

Example:

```
; set the tracking record to the record -12000.
Trace.TRACK -12000.

; without /Track: this window does not respond to the Trace.TRACK command
Trace.List %TimeFixed Time.Zero DEFault

; with /Track: this window responds to the Trace.TRACK command, i.e. the blue cursor selects the tracking record -12000.
Trace.List %TimeFixed Time.Zero DEFault /Track

; display only selected trace information about the record currently selected in the Trace.List ... /Track window
Trace.View %TimeFixed Time.Zero DEFault /Track
```
Selects the trigger source for the port analyzer.

- **BusA, BusB, BusC, BusD**: Trigger lines on the trigger bus. This lines may be controlled by the state analyzer, by the timing analyzer or the pattern generator.
- **EXT ON | OFF**: The external trigger input on the ETM connector is turned off by default. `Analyzer.TSESelect EXT` can enable or disable the trigger source.
- **Port**: The port, selected by the `Port.Select` command, is used as trigger signal.
- **Trigger**: The trigger signal of the emulator system is used to trigger the port analyzer. The emulator trigger signal may be generated by different events e.g. timeout, glitch, state analyzer.
- **Break**: Will trigger, when the emulator trigger system has stopped.
- **ALways**: Trigger is immediately executed, after the trigger state is switched to "Arm" and the pre-trigger time has been elapsed. The trigger mode must be switched to HIGH. If set to LOW, a trigger event cannot be generated.

**See also**

- Trace
The command `Integrator.TSYNC.SIMPLE` resets the values of TPreDelay, TWidth, TCount and TDelay to defaults. Command `Integrator.TSYNC.SELect` doesn’t modify these values.

The trigger signal can be generated out of the 204 port channels. Each signal can be qualified as high, low, rising or falling edge.

```
Channel 0
  High
  Low
  Rising
  Falling
  Don’t Care

Channel 203
```

More than 1 edge can be combined to a trigger word. To detect a valid combination of edges, the edges must have a max. skew of 4 ns.

```
Rising

Falling

4 ns max.
```

Edges and state signals can be combined. The state signal must be stable 4 ns before the edge. The sampling of the state signal is guaranteed before the edge is detected.

```
Low

Rising

8 ns min.
```

See also

- Trace
- 'Release Information' in 'Release History'
The state of the trigger unit and the trigger settings are displayed.

- **level**: Indicates the encountered logical level of the current trigger program.
- **setup file**: Indicates the file name containing the trigger unit program. By clicking on that field will open an editor window for modification.
- **symbol, flag, counter**: Indicates the values and symbolic names of the flags and counters of the trigger unit.

### <trace>.TWidth

**Set trigger filter**

Format: `<trace>.TWidth [<value>]`

- `<value>`: 0 ... 2.5 us

The trigger filter time is defined. All trigger events which are shorter than this value are ignored.

#### Input Signal

#### Trigger Signal

![Filter Time Diagram](#)

**Example:**

```
i.s i.a20 high
i.s i.rxd rising
```
Display single record

Displays a single record in a more detailed format. The syntax of the channel definitions is the same as for the `<trace>.List` command. Without arguments all channels are displayed.

Example 1:

```plaintext
;display all information about a specific record, here record -12000.
Trace.View -12000.

;display only selected trace information about a specific record
Trace.View -12000. TIme.ZERO DEFault CORE
```

Example 2:

```plaintext
;open a Trace.List window with all records. Display the ti.zero column
;as the first column, followed by the DEFault columns
Trace.List TIme.ZERO DEFault /Track

;display only selected trace information about the record currently
;selected in the Trace.List window
Trace.View TIme.Zero DEFault CORE /Track
```

See also

- `<trace>.List`
- `<trace>.state`
- Trace
  - Analyzer.RECORD.SHAKE()
  - Analyzer.RECORD.HEADER()
  - Analyzer.RECORD.RECORDS()
- Analyzer.SIZE()
- "Emulator Functions" in "FIRE User's Guide"
- "Release Information" in "Release History"
- "Displaying the Trace" in "Training FIRE Analyzer"
Cross system tracking

The cross system tracking allows to use the **Track** option for all trace windows between multiple emulators. This allows to correlate events in the analyzer in a multiprocessor environment. The communication between the different emulators is done by the **InterCom** system. **InterCom** allows the exchange of data between different emulators. The InterCom name is either the name and port number of a UDP port used by another emulator (see Installation Manuals for details) or the file name used by another emulator (see command **InterCom** for details). Using TCP/IP for communication is preferred. The tracking can be based on two different synchronization techniques: full time correlation between the system with a synch cable or triggering the systems at the same time. When a synch cable is used and the **SYnch** command is turned on, then the timestamps of all systems will show the same time. If the synch cable is not available then the emulators may be triggered by an event that is available on all units. The tracking assumes that the trigger event, sampled by the analyzers of all emulators, was sampled at the same time on all systems. Triggering all systems at the same time can be done by using trigger input and output lines. The bidirectional coaxial trigger A line of SA120 and HA120 units are very useful for triggering in multiprocessor systems.

**NOTE:** As the time bases of the systems are then not correlated, long measurements can lead to small errors, due to inexact local times.

The following examples uses TCP/IP for the cross tracking:

```
emulator 1                       emulator 2
'Analyzer.XTrackste:20002'       'Analyzer.XTrackste:20001'
```

If TCP/IP is not available the communication can by made by a file:

```
emulator 1                       emulator 2
InterCom c:\tmp\emu_1
'Analyzer.XTrackc:\tmp\emu_2'
```

The following analyzer trigger program can serve as a template:

```
Out.A IF <local_trigger_condition> ; trigger all emulators if the
; local trigger condition arrives
Trigger.A IF inouta               ; local trigger if any processors
; in the system triggers
```

See also
- **<trace>.state**
- **Trace**

©1989-2020 Lauterbach GmbH
Use this command to align the zero time point for trace and timing analyzer sources with time bases of different origin.

<time>         Moves the ZERO point by specified time.

<record>       Sets the ZERO point to the time index of the specified record number.

<bookmark>     Sets the ZERO point to the time index of the specified bookmark location.

You can create trace bookmarks with the <trace>.BookMark command.

no parameter   Reset zero time point back to initial location.

The table below shows the different sources for time information. As the different sources are not related, they all have an individual zero time point.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Trace data source</th>
<th>Original zero time point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamps generated by TRACE32 hardware</td>
<td>Analyzer (no processor generated timestamps) PowerProbe, Integrator, IProbe, etc.</td>
<td>Permanently set to beginning of first debug session or trace recording after starting up TRACE32 PowerView. All trace data sources using TRACE32 hardware generated timestamps have a common zero time point.</td>
</tr>
<tr>
<td>Timestamps generated by target processor</td>
<td>Onchip Trace Analyzer Trace (with processor generated timestamps enabled)</td>
<td>Depends on CPU architecture and trace protocol. Starting a new trace recording usually moves the zero time point to a new location.</td>
</tr>
<tr>
<td>Timestamps loaded from files.</td>
<td>Trace.LOAD &lt;file&gt; /FILE</td>
<td>Same as in original recording</td>
</tr>
</tbody>
</table>

Due to the different zero time points of the various data sources, it is required to align the zero time points, before trace or timing recordings can be observed in a correlated manner. This is usually achieved by locating a common event in the different sources and selecting this event as common zero time point.

See also
- <trace>.state
- Trace
Using the TRACEPORT command group, you can configure the communication between the target trace port and the TRACE32 PowerTrace tool. Logically the TRACEPORT command group is located between the physical pins of the target platform and the TRACE32 trace input stage (preprocessor), see illustration below.

For trace port configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the TRACEPORT.state window.

See also
- TRACEPORT.EndsKiP
- TRACEPORT.LaneCount
- TRACEPORT.LaneSpeed
- TRACEPORT.MsgBitEndian
- TRACEPORT.MsgByteEndian
- TRACEPORT.MsgLongEndian
- TRACEPORT.MsgWordEndian
- TRACEPORT.PinReMap
- TRACEPORT.ReSet
- TRACEPORT.StartsKiP
- TRACEPORT.state
TRACEPORT.EndsKiP Define number of bytes skipped at the end of frame
For serial trace ports (AURORA) only

<table>
<thead>
<tr>
<th>Format:</th>
<th>TRACEPORT.EndsKiP [&lt;option&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;:</td>
<td>AUTO</td>
</tr>
</tbody>
</table>

Allows to cut off data bytes at the end of each data packet or data frame. Depending on the target configuration, the last bytes of a frame contain CRC information, which is not used by TRACE32. With the command TRACEPORT.EndsKiP it is possible to remove the unused bytes.

| AUTO | TRACE32 defines the number of bytes to be cut. |
| 0 | Don’t cut any bytes. |
| 2 | Cut 2 bytes at the end of each frame. |
| 8 | Cut 8 bytes at the end of each frame. |

See also
- TRACEPORT
- TRACEPORT.state

©1989-2020 Lauterbach GmbH
TRACEPORT.LaneCount

Select port size of the trace port

For serial trace ports (AURORA/PCIe) only

<table>
<thead>
<tr>
<th>Format:</th>
<th>TRACEPORT.LaneCount &lt;size&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;size&gt;:</td>
<td>AUTO</td>
</tr>
</tbody>
</table>

Specifies the number of used lanes for the trace port. The number must match the target configuration, else the trace link between the target and the TRACE32 hardware cannot be established.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>TRACE32 defines the lane count.</td>
</tr>
<tr>
<td>1Lane, 2Lane, 3Lane, 4Lane, 5Lane, 6Lane, 7Lane, 8Lane</td>
<td>Number of used lanes. In case of PCIe the lane setup will be done automatically.</td>
</tr>
</tbody>
</table>

See also
- TRACEPORT
- TRACEPORT.state
- TRACEPORT.LaneCount()
Remember that not all TRACE32 PowerTrace tools support all data rates. Contact icrstp-support@lauterbach.com if a lane speed is not supported.

<table>
<thead>
<tr>
<th></th>
<th>Data rate in megabits per second.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>TRACE32 defines the value.</td>
</tr>
<tr>
<td>625Mbps, ...</td>
<td></td>
</tr>
<tr>
<td>GEN1, ...</td>
<td>Limits the data rate of the PCIe link to 2500Mbps (GEN1), 5000Mbps (GEN2) or 8000Mbps (GEN3).</td>
</tr>
</tbody>
</table>

Example:
```
TPIU.PortClock 3125Mbps
TPIU.PortClock 3125M       ; M is the short form of Mbps
```

See also
■ TRACEPORT
■ TRACEPORT.state

TRACEPORT.MsgBltEndian

Change bit-order within each byte

For serial trace ports (AURORA) only

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>TRACE32 defines the value.</td>
</tr>
<tr>
<td>LittleEndian</td>
<td>Bit order is normal ([31-24],[23-16],[15-8],[7-0]).</td>
</tr>
<tr>
<td>BigEndian</td>
<td>Bit order is reversed ([24-31],[16-23],[8-15],[0-7]).</td>
</tr>
</tbody>
</table>

See also
■ TRACEPORT
■ TRACEPORT.state
TRACEPORT.MsgBYteEndian  Change byte-order within each word
For serial trace ports (AURORA) only

Format:  
```
TRACEPORT.MsgBYteEndian [<option>]
```

<option>:  AUTO | LittleEndian | BigEndian

Allows you to change the byte order of the payload data if the byte order used by the target differs from the default bit order. This might be necessary in case of bus connection errors on the target side between the Aurora logic and the trace source.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>TRACE32 defines the value.</td>
</tr>
<tr>
<td>LittleEndian</td>
<td>Byte order is normal ([31-24],[23-16],[15-8],[7-0]).</td>
</tr>
<tr>
<td>BigEndian</td>
<td>Byte order is reversed ([23-16],[31-24],[7-0],[15-8]).</td>
</tr>
</tbody>
</table>

See also
- TRACEPORT
- TRACEPORT.state

TRACEPORT.MsgLOngEndian  Change dword-order within each qword
For serial trace ports (AURORA) only

Format:  
```
TRACEPORT.MsgLOngEndian [<option>]
```

<option>:  AUTO | LittleEndian | BigEndian

Allows you to change the byte order of the payload data if the byte order used by the target differs from the default bit order. This might be necessary in case of bus connection errors on the target side between the Aurora logic and the trace source.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>TRACE32 defines the value.</td>
</tr>
<tr>
<td>LittleEndian</td>
<td>Double-word order is normal ([63-32],[31-0]).</td>
</tr>
<tr>
<td>BigEndian</td>
<td>Double-word order is reversed ([31-0],[63-32]).</td>
</tr>
</tbody>
</table>

See also
- TRACEPORT
- TRACEPORT.state
**TRACEPORT.MsgWOrdEndian**  
Change word-order within each dword  

For serial trace ports (AURORA) only

<table>
<thead>
<tr>
<th>Format: TRACEPORT.MsgWOrdEndian [&lt;option&gt;]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&lt;option&gt;:</th>
<th>AUTO</th>
<th>LittleEndian</th>
<th>BigEndian</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>TRACE32 defines the value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LittleEndian</td>
<td>Word order is normal ([31-16],[15-0]).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BigEndian</td>
<td>Word order is reversed ([15-0],[31-16]).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allows you to change the byte order of the payload data if the byte order used by the target differs from the default bit order. This might be necessary in case of bus connection errors on the target side between the Aurora logic and the trace source.

See also  
- TRACEPORT
- TRACEPORT.state
TRACEPORT.PinReMap
Adapt the lane order of the trace port

For serial trace ports (AURORA) only

Adapts the lane order of the trace port to the lane order of your target. You need the TRACEPORT.PinReMap command only in rare cases where the lane orders of trace port and target actually differ from each other.

| AUTO | TRACE32 defines the values. |
| RESET | Sets all values to AUTO again. |
| <source_lane> | Number of the target lane which needs to be remapped. |
| <destination_lane> | Number of the TRACE32 tool lane which will get the new <source_lane>. Number <n> is TRACE32 tool dependent; e.g. for PowerTrace Serial <n> can be 5 or 7 depending on the used tool connector. |

Example:

TRACEPORT.state /PinReMap ;optionally, open the TRACEPORT.state window
TRACEPORT.LaneCount 6Lane ;the number of used lanes for the trace port
TRACEPORT.PinReMap 4. 5. ;map source lane 4. to destination lane 5.
TRACEPORT.PinReMap 5. 4. ;map source lane 5. to destination lane 4.

See also
- TRACEPORT
- TRACEPORT.state

©1989-2020 Lauterbach GmbH
TRACEPORT.RefCLocK

Set up reference clock for trace port

For serial trace ports (AURORA) only

<table>
<thead>
<tr>
<th>Format:</th>
<th>TRACEPORT.RefCLocK [&lt;option&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;:</td>
<td>AUTO</td>
</tr>
</tbody>
</table>

Defines the reference clock frequency the serial trace hardware outputs to the target. The availability of parameters and the default values depend on the architecture:

- PowerPC: not configurable
- TriCore: not configurable
- RH850: not configurable
- ARM: configurable

<table>
<thead>
<tr>
<th>AUTO (default)</th>
<th>TRACE32 defines the value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>TRACE32 does not send any reference clock to the target.</td>
</tr>
<tr>
<td>OSC</td>
<td>An asynchronous oscillator will be enabled. Its frequency is architecture dependent.</td>
</tr>
<tr>
<td>1/&lt;x&gt;</td>
<td>A synchronous clock source will be enabled. Its dividers generate a reference clock as a fraction of the bit clock (lane speed), e.g. 100MHz at 5Gbps with divider 1/50. Once a divider is selected, the reference clock will automatically change with the lane speed.</td>
</tr>
</tbody>
</table>

See also

- TRACEPORT
- TRACEPORT.state

TRACEPORT.RESet

Reset trace port configuration

| Format: | TRACEPORT.RESet |

Resets the trace port configuration to its default values (AUTO).

See also

- TRACEPORT
- TRACEPORT.state
TRACEPORT.StartsKiP  Define number of bytes skipped at the start of frame

Format:  TRACEPORT.StartsKiP [<option>]

<option>:  AUTO | 0 | 1

Allows to cut off leading bytes of each data packet or data frame. Only a few targets requires this due to protocol irregularities.

<table>
<thead>
<tr>
<th>option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>TRACE32 defines the value.</td>
</tr>
<tr>
<td>0</td>
<td>No data byte will be cut off.</td>
</tr>
<tr>
<td>1</td>
<td>The first data byte of each data frame will be cut off.</td>
</tr>
</tbody>
</table>

See also
- TRACEPORT
- TRACEPORT.state

©1989-2020 Lauterbach GmbH
TRACEPORT.state

Display trace port configuration window

Format: TRACEPORT.state [<gui_option>]

<gui_option>:
- ADVanced
- PinReMap

Displays the TRACEPORT.state window, where you can configure the communication between the target trace port and the TRACE32 PowerTrace tool.

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

B Click advanced and pin mapping to display more configuration options in the window.

ADVanced | Extends the list of options in the configuration section.
---|---
PinReMap | Displays the PinReMap section. For an example, see TRACEPORT.PinReMap.

See also
- TRACEPORT
- TRACEPORT.LaneCount
- TRACEPORT.MsgBitEndian
- TRACEPORT.MsgLOngEndian
- TRACEPORT.PinReMap
- TRACEPORT.RESet

- TRACEPORT.EndsKIP
- TRACEPORT.LaneSpeed
- TRACEPORT.MsgBYteEndian
- TRACEPORT.MsgWOrdEndian
- TRACEPORT.RefCLock
- TRACEPORT.StartsKiP

©1989-2020 Lauterbach GmbH
A pulse can be created at the STROBE probe (ECU32) on accessing a certain address. The pulse width is approx. 30 ns (active-low pulse). The pulse appears at the end of the cycle and is synchronous to the CPU cycles signal. This function is often used to trigger a scope while a software loop is running.

**Breakpoint at address 1002H**

<table>
<thead>
<tr>
<th>AS-</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>1000H</td>
</tr>
<tr>
<td>STROBE.Pin3</td>
<td></td>
</tr>
</tbody>
</table>

**Pin assignment of the STROBE probe (ECU32)**

<table>
<thead>
<tr>
<th>LED</th>
<th>Pin 15 13 11 9 7 5 3 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pin</th>
<th>16 14 12 10 8 6 4 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin 1</td>
<td>Line 0</td>
</tr>
<tr>
<td>Pin 3</td>
<td>Line 1</td>
</tr>
<tr>
<td>Pin 5</td>
<td>Line 2</td>
</tr>
<tr>
<td>Pin 7</td>
<td>Line 3</td>
</tr>
<tr>
<td>Pin 9</td>
<td>Line 4</td>
</tr>
<tr>
<td>Pin 11</td>
<td>Line 5</td>
</tr>
<tr>
<td>Pin 13</td>
<td>Line 6</td>
</tr>
<tr>
<td>Pin 15</td>
<td>Line 7</td>
</tr>
<tr>
<td>Pin 2,4,6,8,10,12,14,16</td>
<td>Ground</td>
</tr>
</tbody>
</table>

On ECC8 emulation controller no **TriggerAddress** command will be supported. Instead the breakpoint CHARLY is routed to the STROBE probe directly.
**Breakpoint CHARLY at address 1002H**

**AS-**

<table>
<thead>
<tr>
<th>Address</th>
<th>1000H</th>
<th>1002H</th>
<th>1004H</th>
</tr>
</thead>
</table>

**STROBE.Pin9**

---

**Pin assignment of the STROBE probe (ECC8)**

<table>
<thead>
<tr>
<th>LED</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Pin</th>
<th>15</th>
<th>13</th>
<th>11</th>
<th>9</th>
<th>7</th>
<th>5</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Pin</th>
<th>16</th>
<th>14</th>
<th>12</th>
<th>10</th>
<th>8</th>
<th>6</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
</table>

- Pin 1  Line 0  OUT.C
- Pin 3  Line 1  OUT.D
- Pin 5  Line 2  RUN- (Foreground)
- Pin 7  Line 3  TRIGGER
- Pin 9  Line 4  CharlyBreak
- Pin 11 Line 5  RUNCYCLE-
- Pin 13 Line 6  PULSe2
- Pin 15 Line 7  PULSe
- Pin 2,4,6,8,10,12,14,16  Ground

©1989-2020 Lauterbach GmbH
**Format:** \texttt{TrAddress [<break>]} \\

**<break>:** Program \\
Hll \\
Spot \\
Read \\
Write \\
Alpha \\
Beta \\
Charly \\

**Examples:**

\begin{verbatim}
  ta ; current state is displayed
  ta alpha ; alpha breakpoint is selected for generating an address
             ; signal
\end{verbatim}
See also

- TRANSlation.AutoEnable
- TRANSlation.CacheFlush
- TRANSlation.COMMON
- TRANSlation.CreateID
- TRANSlation.Delete
- TRANSlation.List
- TRANSlation.NoProtect
- TRANSlation.ON
- TRANSlation.Protect
- TRANSlation.SCANall
- TRANSlation.SHADED
- TRANSlation.TableWalk
- TRANSlation.TRANSPARENT
- MMU.FORMAT
  - TRANS.ENABLE()
  - TRANS.INTERMEDIATEEX()
  - TRANS.LINEAREX()
  - TRANS.PHYSICAL()
  - TRANS.TABLEWALK()
- TRANSlation.AutoScan
- TRANSlation.CLEANUP
- TRANSlation.Create
- TRANSlation.CreateTab
- TRANSlation.DeleteID
- TRANSlation.ListID
- TRANSlation.OFF
- TRANSlation.PAGER
- TRANSlation.RESet
- TRANSlation.ScanID
- TRANSlation.state
- TRANSlation.TlbAutoScan
- MMU
  - SYSTEM.Option MMUSPACES
  - TRANS.INTERMEDIATE()
  - TRANS.LINEAR()
  - TRANS.LOGICAL()
  - TRANS.PHYSICAL()!

▲ ‘TRANS Functions (Debugger Address Translation)’ in ‘General Function Reference’
▲ ‘Release Information’ in ‘Release History’

Overview TRANSlation

NOTE: Formerly, the MMU command group was used for address translation inside the debugger. With the wide-spread adoption of hardware MMUs, it was necessary to rename this command group to TRANSlation to avoid confusion with hardware MMUs.

What is the difference between the command groups...?

<table>
<thead>
<tr>
<th>TRANSlation</th>
<th>MMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configures and controls the TRACE32 internal debugger address translation. This feature is used to mimic the translations within the real hardware MMU so that the debugger can access code and data of any OS process at any time.</td>
<td>Lets you access and view the real hardware MMU.</td>
</tr>
</tbody>
</table>
The **TRANSlation** commands are used for the following purposes:

- To debug an OS that runs several processes at the same logical addresses (e.g. Linux, PikeOS, etc.).
- To allow a transparent display of hardware translation tables and OS-based translation tables.
- To provide the user with unrestricted access to the target memory using either logical or physical addresses.

**MMU Tables**

To apply the MMU commands properly, it is important to differentiate between the following MMU table types:

1. **The hardware MMU table**

   The hardware MMU usually consists of registers and/or dedicated memory areas and is held in the CPU. It holds the translation tables that are used by the CPU to translate the logical addresses used by the CPU into the physical addresses required for memory accesses.

   In OSs like Linux, PikeOS, etc. each process has its own address space. Usually all processes start at the logical address 0x0. The result is that, while a process is running, the process has only access to its own address space and to the address space of the kernel.

   The hardware MMU is programmed by the scheduler for this view. If, for example, process 2 is running, the hardware MMU provides only translation tables for process 2 and the kernel.

   ![Logical addresses vs. Physical addresses diagram](image)

   - If the OS uses demand paging, the hardware MMU table is extended at each page fault.
   - At each process switch the hardware MMU is reprogrammed so that the logical address space of the current process can be translated to the physical address area.

2. **The software/OS MMU table**

   If an OS like Linux, PikeOS etc. is used, the OS maintains the translation tables for all processes, because the OS is responsible for the reprogramming of the hardware MMU on a process switch.

   The hardware MMU is usually only a subset of the OS MMU tables.
3. The debugger MMU table

If an OS that runs several processes at the same logical addresses (e.g. Linux) is used, the hardware MMU in the CPU only holds translation tables that allow the debugger memory accesses to the code/data of the kernel and the currently running process.

The debugger can access code/data from a not currently running process only with the help of the OS MMU tables. Based on the information held in the translation tables of the OS MMU, the debugger can translate any logical address to a physical address and that way perform a memory access without changing the hardware MMU. If demand paging is used, the OS MMU table contains the translation from the logical to the physical address only if the page was loaded before.

Reading the OS MMU tables on every memory access in quite time consuming. Therefore the debugger can scan the OS MMU tables once and re-use the scan for all following accesses.

The OS MMU table is scanned into the so-called debugger MMU. The debugger MMU provides also the flexibility to add user-defined entries.

| Please be aware that as soon as the debugger MMU is active, all memory accesses performed by the debugger use only the information of the debugger MMU. Please be aware that the OS MMU tables have to be scanned again if the OS has added or removed entries from these tables while running. |
**TRANSlation.AutoEnable**

Auto-enable debugger MMU translation

**Format:**

```
TRANSlation.AutoEnable
```

Auto-enable the debugger address translation if the CPU's hardware MMU is enabled. When the hardware MMU is on, the debugger also performs translations. When the hardware MMU is off, the debugger performs no translation and treats all logical addresses as physical addresses. The state of the hardware MMU is read from the CPU-specific MMU-enable bit in a system control register. This command is only available for certain CPUs.

**See also**

- TRANSlation
- TRANSlation.List
- TRANSlation.OFF
- TRANSlation.ON

---

**TRANSlation.AutoSCAN**

Autoscan feature for debugger MMU

**Format:**

```
TRANSlation.AutoSCAN ON | OFF
```

If the operating system adds or removes entries from its page table while running those changes are not performed within the debugger MMU. Trying to access those newly created logical addresses with the debugger may cause an error. If `TRANSlation.AutoSCAN` is set to ON the translation tables hold by the operating system are automatically scanned into the debugger MMU, if the debugger fails to access a logical address.

`TRANSlation.AutoSCAN` scans only pages that are already present. Depending on the JTAG speed of the processor and on the number of processes in the system scanning the translation tables can take some time. In this cases autoscanning may be more disturbing than helping.

**See also**

- TRANSlation
- TRANSlation.List
**TRANSlation.CacheFlush**

Flush TRACE32 address translation cache

| Format: | TRANSlation.CacheFlush [ALL] |

Successful MMU address translations are cached internally by TRACE32. This speeds up recurring accesses to a logical address in debug mode - mostly when the OS Awareness is enabled. Caching is most beneficial for translations done via an MMU table walk as this generates many memory accesses while parsing the OS page table.

**TRANSlation.CacheFlush** flushes the TRACE32 internal address translation caches, so a new readout of the OS page table is enforced for the next memory access. This can be useful when modifying page table content or debugging MMU table walks.

**ALL**
Additionally invalidates the complete register set cached by the debugger, including all cached MMU registers. Upon the next MMU page table walk, the registers will be re-read from the target.

See also
- TRANSlation
- TRANSlation.List
- TRANSlation.TableWalk

**TRANSlation.CLEANUP**

Clean up MMU table

| Format: | TRANSlation.CLEANUP MMU.CLEANUP (deprecated) |

Removes multiple translations for one physical address, directly joining translations and double translations.

See also
- TRANSlation
- TRANSlation.List
TRANSlation.COMMON

Common address ranges for kernel and tasks

**Format:**

```
TRANSlation.COMMON <logical_range> [[<logical_range>]]
```

MMU.COMMON (deprecated)

Defines one or more mappings of logical address ranges that are shared by the kernel and the tasks.

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

This allows to apply the kernel address translation to modules or libraries that are called by a user process in the context of the currently running task.

**A** Space ID of the kernel = 0x0000

**B** Space IDs of the tasks ≠ 0x0000

TRACE32 assigns space IDs if SYStem.Option MMUSPACES is set to ON

**C** Common logical address ranges are flagged as “COMMON” in the TRANSlation.List window, which displays the mappings between logical and physical address ranges.

**NOTE:**

Executing the TRANSlation.COMMON command again discards all previously existing common address ranges.

Use TRANSlation.COMMON.ADD to add further common address ranges without discarding existing common address ranges.

<table>
<thead>
<tr>
<th><strong>&lt;logical_range&gt;</strong></th>
<th>You can specify up to 10 common ranges in one line. For an example with two common address ranges, see below. Overlapping common address ranges are merged automatically.</th>
</tr>
</thead>
</table>

©1989-2020 Lauterbach GmbH
;Enable the space IDs to display them in the TRANSlation.List window
SYStem.Option MMUSPACES ON

TRANSlation.List ;Open the Translation.List window

;Create some translation entries for a particular debug session
TRANSlation.Create 0x0:0x00000000++0xffffffff 0x10000
TRANSlation.Create 0x123:0x00000000++0xffffffff 0x101000
TRANSlation.Create 0x042:0x00000000++0xffffffff 0x102000

;Define two common logical address ranges:
; <logical_range_1>         <logical_range_2>
TRANSlation.COMMON 0x80004020--0x800041ff   0x80004700--0x800049ff

;Alternatively, you can define the common ranges as follows:
;TRANSlation.COMMON 0x80004020--0x800041ff ;define the first range
;TRANSlation.COMMON.ADD 0x80004700--0x800049ff ;add the second range

See also
- TRANSlation.COMMON.ADD
- TRANSlation
- MMU.FORMAT
- TRANSlation.COMMON.CLEAR
- TRANSlation.List
- SYStem.Option MMUSPACES

▲ 'Release Information' in 'Release History'
**TRANSlation.COMMON.ADD**

Add another common address range

<table>
<thead>
<tr>
<th>Format:</th>
<th>TRANSlation.COMMON.ADD &lt;logical_range&gt;</th>
</tr>
</thead>
</table>

Adds another mapping for a common logical address range that is shared by the kernel and the tasks.

**NOTE:** Use `TRANSlation.COMMON.ADD` to add further common address ranges without discarding existing common address ranges. Executing `TRANSlation.COMMON` again discards all previously existing common address ranges.

**Example:**

```plaintext
; Define the first common logical address range
TRANSlation.COMMON 0x80000200--0x800007ff

; Add two additional ranges
TRANSlation.COMMON.ADD 0x80004020--0x800041ff
TRANSlation.COMMON.ADD 0x80004700--0x800049ff
```

**See also**

| TRANSlation.COMMON | TRANSlation.COMMON.CLEAR |

**TRANSlation.COMMON.CLEAR**

Clear all common logical address ranges

<table>
<thead>
<tr>
<th>Format:</th>
<th>TRANSlation.COMMON.CLEAR</th>
</tr>
</thead>
</table>

Clears only those logical address ranges that are flagged as “COMMON” in the `TRANSlation.List` window.

**See also**

| TRANSlation.COMMON | TRANSlation.COMMON.ADD |

©1989-2020 Lauterbach GmbH
The defined translation can either be function code specific or generic for all function codes (except I/O). The physical address or range is not allowed on probes with fixed MMU translation (e.g. 80186,Z180).

**More**

The **More** option suppresses the generation of the MMU tables. This speeds up the entry of large translation tables with PRACTICE scripts (*.cmm). The last translation command should not have a **More** option, otherwise the translations are not accessible.

**Logical** and **Physical**

The **Logical** and **Physical** options create translations that work only in one direction. This allows to create multiple logical addresses that map to the same physical address and still having a well-defined logical address for the reverse translation.

**Example:** Translation for 68030 TRANSLation

```
TRANSlation.Create 0x1000--0x1fff a:0x20000--0x20fff /More
TRANSlation.Create sd:0x2000--0x2fff asd:0x0--0x0fff /More
TRANSlation.Create ud:0x2000--0x2fff aud:0x1000--0x1fff /More
TRANSlation.Create sp:0x2000--0x2fff asp:0x2000--0x2fff /More
TRANSlation.Create up:0x2000--0x2fff aup:0x3000--0x3fff
```

**See also**

- TRANSlation
- TRANSlation.List
- TRANSlation.TlbAutoScan
- MMU.FORMAT
- MMU.CREATE
- MMU.SCAN
- SYStem.Option MMUSPACES

▲ ‘ARM Specific Implementations’ in ‘ARM Debugger’
▲ ‘ARM Specific Implementations’ in ‘ARMv8-A/-R Debugger’
**TRANSlation.CreateID**

Add entry to MMU space ID table

Format:

```
TRANSlation.CreateID <space_id>:0x0 <base_address>
```

- `<space_id>`: Space ID to be added.
- `<base_address>`: Physical base address of task page table associated with `<space_id>`.

**See also**

- TRANSlation
- TRANSlation.DeleteID
- TRANSlation.List
- TRANSlation.ListID

---

**TRANSlation.CreateTab**

Create multiple translations

Format:

```
TRANSlation.CreateTab <logical_range> <increment> <logical_range>

MMU.CreateTab (deprecated)
```

- `<option>`:
  - More
  - Logical
  - Physical

Same as **TRANSlation.Create**, but creates multiple translations with one command. The first range defines the logical range for the created translations. The increment parameter is the offset added to the logical address to generate the next address. The other parameters are interpreted like the **TRANSlation.Create** command.

**Example:**

```
; Translation for COMMON area from 0x08000--0x0fffff
TRANSlation.ct 0x0--0x0fffff 0x10000 0x08000--0x0ffff 0x08000--0x0fffff

; Translation for 16 BANKS
TRANSlation.ct 0x0--0x0fffff 0x10000 0x0--0x7fff
```

**See also**

- TRANSlation
- TRANSlation.List

©1989-2020 Lauterbach GmbH

General Commands Reference Guide T 491
**TRANSlation.Delete**

Delete translation

The defined translation is removed from the list; see **TRANSlation.List**. Use the command **TRANSlation.RESet** to clear the whole list.

**Example:**

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSlation.Delete 0x1000--0x1fff</td>
</tr>
</tbody>
</table>

**See also**

- **TRANSlation**
- **TRANSlation.List**

**TRANSlation.DeleteID**

Remove entry from MMU space ID table

Format: **TRANSlation.DeleteID** <space_id>:0x0

- **<space_id>**
  - Space ID to be removed.

**See also**

- **TRANSlation**
- **TRANSlation.CreateID**
- **TRANSlation.List**
- **TRANSlation.ListID**
- **TRANSlation.ScanID**
Displays the list of static address translations created with the commands `TRANSlation.Create` or `MMU.SCAN`.

The static MMU translation table of TRACE32 contains relations between logical address spaces and physical address spaces. This table is consulted when the debugger address translation is enabled with `TRANSlation.ON` and a logical address must be converted into a physical address. In some cases this table is also used for reverse translating a physical address into its logical counterpart.

### MMU.List (deprecated)

#### Format:

`TRANSlation.List [Logical | Physical]`  
`MMU.List`  

#### A  
Space ID of the kernel = 0x0000

#### B  
Space IDs of the tasks ≠ 0x0000  
TRACE32 assigns space IDs if `SYStem.Option MMUSPACES` is set to `ON`.

#### C  
The default logical-to-physical address translation, which is used for fast memory accesses into the kernel address range.  
The default address translation is specified with the command `MMU.FORMAT`.

#### D  
Common address ranges are created with the commands `TRANSlation.COMMON` or `TRANSlation.COMMON.ADD`.

<table>
<thead>
<tr>
<th>Logical</th>
<th>Sorts logical addresses in ascending order.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical</td>
<td>Sorts physical addresses in ascending order.</td>
</tr>
</tbody>
</table>

**See also**

- `TRANSlation.ListID`
- `TRANSlation.CacheFlush`
- `TRANSlation.CreateID`
- `TRANSlation.NoProtect`
- `TRANSlation.Protect`
- `TRANSlation.SHADOW`
- `TRANSlation.TRANSparent`
- `MMU.FORMAT()`

▲ ‘Release Information’ in ‘Release History’

©1989-2020 Lauterbach GmbH
The table is used to translate MMU root pointer contents into a memory space ID. Memory space IDs may also be obtained from the OS Awareness without the use of this table.

See also
- TRANSlation.List
- TRANSlation
- TRANSlation.CreateID
- TRANSlation.DeleteID

**TRANSlation.NoProtect**

Unprotect memory

Format: 

TRANSlation.NoProtect <logical_range>

MMU.NoProtect (deprecated)

Removes the protection for the specified logical address range. As a result, the debugger can access this range. See TRANSlation.Protect.ADD.

Example:

```plaintext
TRANSlation.Protect.ADD 0x100000--0x1fffff ;no access here
TRANSlation.Protect.ADD 0x280000--0x0fff000 ;no access here
TRANSlation.Protect.ADD 0x1000000--0x0ffffffff ;no access here

TRANSlation.ON
TRANSlation.List ;display overview of protected memory ranges

;your code

;remove this logical address range from the list of protected memory
;ranges
TRANSlation.NoProtect 0x100000--0x0ffffffff
```

See also
- TRANSlation
- TRANSlation.List
- TRANSlation.Protect
- TRANSlation.Protect.ADD
Deactivates the TRACE32 internal debugger address translation.

**Logical addresses** used by the debugger are directly sent to the target CPU without translation. Also, the protection of address ranges which have been declared as protected is disabled.

See also

- **TRANSlation.ON**
- **TRANSlation**
- **TRANSlation.AutoEnable**
- **TRANSlation.AutoEnable**

**TRANSlation.ON**

Activates the TRACE32 internal debugger address translation. For Intel® architecture debuggers, the address translation is enabled by default. For all other architectures, the default is **TRANSlation.OFF**.

With **TRANSlation.ON**, the following features are enabled:

- **Logical addresses** are translated to **physical addresses**. The address translation is based on the following translation tables:
  - The static address translation list (see **TRANSlation.List**)
  - Intel® architectures only: the segment translation for boot mode, real mode and protected mode (see **MMU.view**, **MMU.DUMP.GDT**, **MMU.DUMP.LDT** and **SYStem.Option.MEMoryMODEL**)
  - MIPS architectures only: the EVA or fixed mapping KSEG0/1 translations are done.
  - OS page tables if the TRACE32 table walk is enabled (see **TRANSlation.TableWalk** and **MMU.FORMAT**).
  - For some architectures, TLBs can be evaluated. This feature is also enabled with **TRANSlation.TableWalk** and **MMU.FORMAT**.
- **Address ranges declared as protected** are no longer accessible to the debugger (see **TRANSlation.Protect**).
For an overview of the state of the debugger address translation, see `TRANSlation.state`.

### TRANSlation.PAGER

Allow paged breakpoints for Linux

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TRANSlation.PAGER ON</strong> <code>&lt;address&gt;</code></td>
</tr>
</tbody>
</table>

The TRACE32 software and a suitable Linux patch enable a software breakpoint to be set for program code that has not yet been loaded.

Details on the Linux patch can be found in the directory `~/demo/arm/kernel/linux/etc/t32pager`

See also

- `TRANSlation.Off`
- `TRANSlation.List`

©1989-2020 Lauterbach GmbH
Using the `TRANSlation.Protect` command group, you can protect the entire logical address range or individual logical address ranges from debugger access. This can be useful if an access would otherwise cause a fatal hardware error or cause the debugger to go down.

**What is the difference between...?**

<table>
<thead>
<tr>
<th><code>TRANSlation.Protect.ON</code></th>
<th><code>TRANSlation.Protect.ADD</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Protects the <em>entire</em> logical address range from debugger access.</td>
<td></td>
</tr>
<tr>
<td>• However, you can allow debugger access to individual logical address ranges by specifying them with <code>TRANSlation.Create &lt;logical_range&gt;</code>.</td>
<td></td>
</tr>
<tr>
<td>• Protects <em>individual</em> logical address ranges from debugger access.</td>
<td></td>
</tr>
<tr>
<td>• <code>TRANSlation.Protect</code> must <strong>not</strong> be set to <strong>ON</strong>.</td>
<td></td>
</tr>
</tbody>
</table>

**See also**
- `TRANSlation.Protect.ADD`
- `TRANSlation.Protect.OFF`
- `TRANSlation.Protect.ON`
- `TRANSlation.List`
- `TRANSlation.NoProtect`
- `TRANSlation`

---

**`TRANSlation.Protect.ADD`**

**Add range to protected memory ranges**

[Example]

**Format:**

```
TRANSlation.Protect.ADD <logical_range>
```

Protects the specified logical address range from debugger access.

**NOTE:**

Use `MAP.DenyAccess` to protect physical address ranges from debugger access. Use `TRANSlation.Protect.ADD` to protect logical address ranges from debugger access.
Example:

```plaintext
;[A] allow debugger access to the logical address ranges 0x0--0x103F
;    and 0x1070--0xFFFFFFFF, i.e. almost the entire logical range, ...

;[B] ...but protect this logical address range from debugger access
TRANSlation.Protect.ADD 0x1040--0x106F
TRANSlation.ON

;display overview of protected memory range(s)
TRANSlation.List

;let's open this window for demo purposes to visualize the result
Data.dump 0x1020 /NoAscii
```

See also

- TRANSlation.Protect
- TRANSlation.NoProtect
- MAP.DenyAccess

**TRANSlation.Protect.OFF**

Switch protection of target memory off.

Format: `TRANSlation.Protect.OFF`

Re-allows debugger access to the entire logical address range. See `TRANSlation.Protect.ON`.

See also

- TRANSlation.Protect
Protect entire target memory

Format: TRANSlation.Protect.ON

Protects the entire logical address range from debugger access, provided the address translation is enabled with TRANSlation.ON.

Example:

TRANSlation.ON

; protect entire logical address range from debugger access (see red [A])
TRANSlation.Protect.ON

; but allow debugger access to this logical address range (see green [B])
TRANSlation.Create 0x1040--0x106F

; display overview of static translations
TRANSlation.List

; let's open this window for demo purposes to visualize the result
Data.dump 0x1020 /NoAscii

See also

- TRANSlation.Protect
**TRANSlation.RESet**  
Reset MMU configuration

Format:  
TRANSlation.RESet  
MMU.RESet (deprecated)

The translation table is cleared and all setups are set to the defaults.

See also  
- TRANSlation  
- TRANSlation.List

**TRANSlation.SCANall**  
Scan MMU tables

Format:  
TRANSlation.SCANall  
MMU.SCAN ALL (as an alias)  
MMU.SCANALL (deprecated)

Scans all page translation tables known to the debugger into the static translation list. That is, this command is a repeated call of the MMU.SCAN command for all known page tables of an architecture known to the debugger.

See also  
- TRANSlation  
- TRANSlation.List

**TRANSlation.ScanID**  
Scan MMU address space tables from kernel

Format:  
TRANSlation.ScanID

Scans the translation information from the kernel into the MMU space ID table. The operation is target and kernel dependent.

See also  
- TRANSlation  
- TRANSlation.CreateID  
- TRANSlation.DeleteID  
- TRANSlation.List

©1989-2020 Lauterbach GmbH
Use VM: for data access, if the address translation on the target failed.

The debugger first tries to resolve a logical address with the standard address translation, and then accesses the target to read the requested data. If the translation fails (due to missing table entries, or due to an access error), and if Transit. SHADOW is ON, the debugger uses the data within VM: at the requested address.

The debugger provides a “virtual memory” (access class VM:) that is not accessible from the CPU, but only by the debugger (stored within the host). The idea is to have a (partial) copy of the target memory in the host for unlimited access.

VM: usually is a “virtual physical memory”. The debugger does an address translation (logical -> physical), then accesses VM: with the physical address. I.e. VM: maps a physical memory.

If Transit. SHADOW and SYStem. Option MMUSPACES is ON, VM: is used as several logical addressed memory areas, separated by the space ID. No address translation is done, instead the debugger directly accesses the memory in VM: with space ID:address. I.e. VM: maps several logical memory areas. In complex OS target systems (e.g. Linux), you may load the code of several processes into VM: to have access to the code, even if the target does currently not allow memory access.

See also
- Transit
- Transit. List
Format: **TRANSlation.state**

Opens the **TRANSlation.state** window.

A The header displays an overview of all settings affecting the debugger address translation:
- Address translation: ON, OFF = TRANSLation.ON or TRANSLation.OFF
- MMU protection: ON, OFF = TRANSLation.Protect.ON or TRANSLation.Protect.OFF
- Table walk: ON, OFF = TRANSLation.TableWalk [ON | OFF]
- MMU spaces: ON, OFF = SYStem.Option.MMUSPACES [ON | OFF]
- Zone spaces: ON, OFF = SYStem.Option.ZoneSPACES [ON | OFF]
- Machine spaces: ON, OFF = SYStem.Option.MACHINESPACES [ON | OFF]
- Architecture-specific settings (here LPAE)

B The columns below the header list the settings configured with the **MMU.FORMAT** command.

---

**Description of Columns in the TRANSlation.state Window**

<table>
<thead>
<tr>
<th>Zone</th>
<th>For information about zones, refer to the glossary.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMU format</td>
<td>The MMU formats for each zone.</td>
</tr>
<tr>
<td>Default page table</td>
<td>The start addresses of the default page tables for all active zones.</td>
</tr>
</tbody>
</table>

---

**See also**
- TRANSLation
- TRANSlation.List

©1989-2020 Lauterbach GmbH
TRANSlation.TableWalk

Automatic MMU page table walk

| Format: | TRANSlation.TableWalk [ON | OFF] |

Confirms the debugger to perform an MMU page table walk (short: table walk). If enabled, the debugger will try the following steps upon a logical-to-physical address translation request:

1. Look up the logical address in the debugger’s static address translation table (see TRANSlation.List and TRANSlation.Create for details about the static address translation table).

2. If the address lookup in the static address translation table fails, walk through the software/OS MMU tables to find a valid logical-to-physical translation.

3. For Intel® architecture debuggers, the boot mode, real mode, or protected mode segment translation is done before the page table walk is performed.

4. For MIPS architectures only: the EVA or fixed mapping KSEG0/1 translations are done before the page table walk is performed.

Valid address translations found are cached by TRACE32. When debug mode is left, i.e. at a Go or Step, the cached translations are flushed because page table contents may change when the target continues execution.

ON

Configure TRACE32 to use the automatic MMU table walk. Only physical addresses are sent to the target.

**NOTE for expert users:**
For some architectures, although a valid translation is available, TRACE32 sends logical addresses in certain situations in order to ensure cache coherency.
This behavior can be controlled with the architecture-specific command SYStem.Option.MMUPhysLogMemaccess.

OFF

Configure TRACE32 to not use the automatic MMU table walk.

**NOTE:**
Page tables are dynamic structures and are frequently modified by the OS.

The MMU page table walk of the debugger dynamically parses the page tables on demand for every debugger address translation. The table walk ensures that the debugger address translations correspond to the current OS address translations.

©1989-2020 Lauterbach GmbH
If no valid translation could be found for a logical address in any available translation table, then the error handling depends on whether `TRANSlation.TableWalk` is set to **OFF** or **ON**:

<table>
<thead>
<tr>
<th>OFF</th>
<th>No error will be produced by TRACE32. The logical address will be sent to the target CPU without translation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>A “MMU translation failed” error will be produced by TRACE32. Scripts will stop upon a failing translation. This mimics the behavior of the target MMU, where a failing translation causes an exception.</td>
</tr>
</tbody>
</table>

See also
- `TRANSlation`
- `TRANSlation.CacheFlush`
- `TRANSlation.List`
- `MMU.FORMAT`
- `TRANS.TABLEWALK()`
- ‘ARM Specific Implementations’ in ‘ARM Debugger’
- ‘ARM Specific Implementations’ in ‘ARMv8-A/-R Debugger’
- ‘Release Information’ in ‘Release History’

**TRANSlation.TlbAutoScan**

Allow automatic TLB scans during table walk

Format:

```
TRANSlation.TlbAutoScan [ON | OFF] [<logical_range>] [<logical_range>]
```

Enable automatic scan of the TLBs for missing kernel address translations during MMU table walks. Ignore TLB entries with logical addresses outside the specified `<logical_range>`.

**NOTE:**
This command is not available for all architectures

Some OS specify logical base addresses for kernel or task page tables. The table walk algorithm must translate them to physical addresses before the page table can be parsed. If there is no suitable user-defined default translation (`MMU.FORMAT`) or debugger MMU table (`TRANSlation.List`) entry, `TRANSlation.TlbAutoScan` will search the target MMU TLBs for a suitable translation that has been set up and used by the OS itself. If a suitable translation is found, it is copied into the debugger MMU table. This automatism can prevent debugger memory access failures caused by incomplete MMU setup scripts.

**NOTE:**
Only TLB entries in the kernel address range must be auto-extracted from TLBs. If you specify the typical kernel address range(s) for your target’s OS in `<logical_range>`, `TRANSlation.TlbAutoScan` will ignore dynamic TLB entries used for user processes.
Place the **TRANSlation.TlbAutoScan** command into the MMU section of your PRACTICE script preparing the debugger for OS Awareness as in this example:

```
;*******************************************************************
; example MMU setup section for Linux awareness
; - "TRANSlation.TlbAutoScan ON" replaces the explicit
;   default translation in MMU.FORMAT and fixed kernel
;   address translations in TRANSlation.Create.
;*******************************************************************
PRINT "Initializing debugger MMU..."
MMU.FORMAT LINUX swapper_pg_dir
TRANSlation.COMMON 0xC0000000--0xFFFFFFFF

; this translation will be auto-extracted by TlbAutoScan from the TLB
; TRANSlation.Create 0xC0000000--0xCFFFFFFF 0x0

; enable TlbAutoScan - TLB entries in 0x80000000--0xFFFFFFFF are kernel
; addresses here and ok to be auto-extracted
TRANSlation.TlbAutoScan ON 0xC0000000--0xFFFFFFFF

TRANSlation.TableWalk ON
TRANSlation.ON
```

**See also**

- TRANSlation
- TRANSlation.Create
- TRANSlation.List
- MMU.DUMP
- MMU.FORMAT
- MMU.SCAN
A debugger access to a logical address within `<logical_range>` will not be translated to a physical address, even if a page table translation for it is defined. Instead, this access will use the logical address.

**Example**

In a banked memory system, we want the debugger to see the current memory bank (selected by the CPU’s BNK register) for memory accesses within `<logical_range>`. The following example shows a PRACTICE script for such a setup for a CPU with 16-bit logical addresses:

```plaintext
sYmbol.RESet
TRANSlation.Reset
; define fixed translation window into bank 0
TRANSlation.Create 0x1000000--0x10000ff A:0x00000--0x00fff
; define fixed translation window into bank 1
TRANSlation.Create 0x1010000--0x10100ff A:0x10000--0x10fff
; define transparent address window (no translation in this range)
TRANSlation.TRANSparent 0x0--0xffff
TRANSlation.ON
; load code into current bank, somewhere in 0x0--0xffff
Data.LOAD.UBROF example.dbg
; shift symbols to logical addresses at 1000000
sYmbol.RELOCate C:0x1000000
```

Any access within 0x0..0xffff is defined as transparent and will thus not be translated to a physical address by the debugger. Instead, such an access will be carried out with the logical address, so the CPU’s “current bank” register will decide which data is seen. That is, examining a variable pointing to a certain logical address somewhere within 0x0..0xffff with bank 1 being active, will show the data stored in bank 1.

We want to make sure that symbols belonging to code or data loaded into a certain bank are always tied to the correct bank. Addresses in 0x0..0xffff may show any bank, depending on the BNK register. So we first define fixed translation windows of 0x1000000..0x10000fff to bank 0 and 0x1010000..0x10100fff to bank 1. Note that those address windows exist only for the debugger.

Now we load code (assuming bank 0 being selected by register BNK) into memory. Finally, we shift the symbols belonging to the code into the address window belonging to bank 0, i.e. we add an offset of 0x1000000 after loading. Now we have a clear assignment between the symbols and the data in bank 0, while debugger accesses to logical addresses in 0x0..0xffff still see the data the CPU sees currently.

**See also**

- TRANSlation
- TRANSlation.List

©1989-2020 Lauterbach GmbH
The **TrBus** command group allows:

- To generate a trigger pulse that can be used to trigger an external device e.g. a Logic Analyzer.
- To connect an incoming trigger signal to TRACE32-ICD or TRACE32-FIRE.

In both cases the TRIG connector is used. The TRIG connector has the following characteristics on the different TRACE32 tools:

<table>
<thead>
<tr>
<th>TRACE32 tool</th>
<th>Output voltage</th>
<th>Input voltage</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerDebug PRO</td>
<td>4.4V</td>
<td>3.3V</td>
<td>Input: 5V tolerant, 10K pull-up/down*</td>
</tr>
<tr>
<td>uTrace for Cortex-M</td>
<td>3.3V</td>
<td>3.3V</td>
<td>Input: 5V tolerant, 10K pull-up/down*</td>
</tr>
<tr>
<td>PowerDebug Module USB 3.0</td>
<td>4.4V</td>
<td>3.3V</td>
<td>Input: 5V tolerant, 10K pull-up/down*</td>
</tr>
<tr>
<td>PowerDebug II Ethernet</td>
<td>5.0V</td>
<td>3.3V</td>
<td>Input: 5V tolerant, 10K pull-up/down*</td>
</tr>
<tr>
<td>PowerDebug Module Ethernet / PowerTrace Ethernet</td>
<td>3.3V</td>
<td>3.3V</td>
<td>Input: 5V tolerant, 10K pull-up/down*</td>
</tr>
<tr>
<td>Power Debug Module USB 2.0</td>
<td>3.3V</td>
<td>3.3V</td>
<td>Input: 5V tolerant, 10K pull-up/down*</td>
</tr>
<tr>
<td>PODBUS Ethernet Controller</td>
<td>3.3V</td>
<td>3.3V</td>
<td>Input: 5V tolerant, 10K pull-up/down*</td>
</tr>
<tr>
<td>Power Debug Module USB 1.x</td>
<td>3.3V</td>
<td>3.3V</td>
<td>Input: 5V tolerant, 10K pull-up/down*</td>
</tr>
</tbody>
</table>
* Pull-up/down selected automatically depending on low-active or high-active settings.

An **external trigger pulse** of at least 100ns can be generated when:

- The program execution is stopped.
- A trigger is generated for the trace (not available on all CPUs, depends on the implementation of the trace trigger feature).
- The sampling to the trace buffer is stopped (not available on all CPUs, depends on the implementation of the trace trigger feature).
- A breakpoint with the Action BusTrigger is used (not available on all CPUs).

An **incoming trigger signal** can be used:

- To stop the program execution.
- To generate a trigger for the trace (not available on all CPUs, depends on the implementation of the trace trigger feature).

The sources for the external trigger pulse and the targets for the incoming trigger signal are connected to the trigger bus.
The following picture shows the Trigger Bus on the PowerTrace as an example.

```
Example: Generate a trigger for the trace at a falling edge of the incoming trigger signal.

TrBus.Arm ; Switch the trigger bus ON
TrBus.Connect In ; Configure the TRIGGER connector as input
TrBus.Mode.Falling ; define that the trigger target should react
  ; on the falling edge of the incoming trigger signal
TrBus.Set ATrigger ON ; generate a trigger for the trace (trigger
  ; target) on the falling edge of the external trigger signal
  ; a trigger for the trace can stop the sampling to the trace directly or it can be
  ; delayed by the command Analyzer.TDelay
TrBus.Set Break OFF ; switch all other sources and targets to OFF
TrBus.Out Break OFF
TrBus.Out ABreak OFF
TrBus.Out ATrigger OFF
```

©1989-2020 Lauterbach GmbH
Interaction Between Independent PODBUS Devices

If several independent PODBUS devices are plugged together, they share the same trigger bus. Example configurations are:

- A POWER DEBUG II and a POWER TRACE II
- A POWER DEBUG INTERFACE / USB and a POWERPROBE
- A POWERTRACE / ETHERNET and a POWERINTERGRATOR
- Several POWER DEBUG INTERFACEs that form a multi-processor debugging environment.

The common trigger bus allows a synchronization between the PODBUS devices.
**Example:** A soon as the POWERPROBE is stopped by a trigger, the program execution should be stopped via the connected POWER DEBUG INTERFACE:

```
PowerProbe

... PP:Analyzer.TOut BUSA ON ; definition of the trigger condition ; generate a trigger for the trigger bus ; when the defined trigger event occurs

Debugger

TrBus.Arm ; switch the trigger bus ON
TrBus.Connect Out ; Configure the TRIGGER connector as output
TrBus.Set Break ON ; allow any trigger from the trigger bus to ; stop the program execution
```

The trigger bus also allows to stop the processors in a multi-processor configuration synchronously. Precondition is that the JTAG/OCDS/BDM … connector provides:

- A signal which indicates that the program execution was stopped (stop indication).
- A signal that allows to stop the program execution immediately (stop request).

After the configuration for the synchronous start and stop by the **SYnch** command is done, you can configure the stop synchronization per hardware by the **TrBus** commands.

Multicore chip sets provide normally internal (chip specific) trigger connections.
Example: Configure the stop synchronization per hardware for the TRICORE OCDS connector:

**OCDS connector 1**

<table>
<thead>
<tr>
<th>Break Set (from Bus)</th>
<th>Break Out (to Bus)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/BRK_IN (Stop request)</td>
<td>/BRK_OUT (Stop indication)</td>
</tr>
</tbody>
</table>

**OCDS connector 2**

<table>
<thead>
<tr>
<th>Break Set (from Bus)</th>
<th>Break Out (to Bus)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/BRK_IN (Stop request)</td>
<td>/BRK_OUT (Stop indication)</td>
</tr>
</tbody>
</table>

**Trigger Bus**

- **TrBus.Arm**; Switch the trigger bus ON
- **TrBus.Connect Out**; Configure the TRIGGER connector as output
- **TrBus.Set Break ON**; Connect Break to stop request
- **TrBus.Out Break ON**; Connect Break to stop indication

**Format:**

```
TrBus.Arm
```

Arms the trigger bus.

See also
- TrBus
- TrBus.state
TrBus.Connect

Configure TRIGGER as input or output

Debugger / FIRE only

Format: TrBus.Connect In | Out

The TRIGGER connector should work as:

- Input for an incoming trigger signal.
- Output for the generation of an external trigger signal.

See also

- TrBus
- TrBus.state

TrBus.Mode

Define polarity/edge for the trigger signal

Debugger / FIRE only

Format: TrBus.Mode Low | High | Falling | Rising

If TrBus.Connect Out is set a Low or High pulse is generated on TRIGGER (at least 100 ns) as soon as the defined source becomes active.

If TrBus.Connect In is set, the defined target can react on a Low/High pulse or Falling/Rising edge of the incoming trigger signal.

See also

- TrBus
- TrBus.state

TrBus.OFF

Switch trigger bus off

Debugger / FIRE only

Format: TrBus.OFF

Switches the trigger bus off.

See also

- TrBus
- TrBus.state
TrBus.Out

Define source for the external trigger pulse

```
Format: TrBus.Out Break | ABreak | ATrigger [ON | OFF]
```

Defines the source for the external trigger pulse.

- **Break**: Generate an external trigger pulse when the program execution is stopped.
- **ABreak**: Generate an external trigger pulse when the sampling to the trace buffer is stopped.
- **ATrigger**: Generate an external trigger pulse when a trigger is generated for the trace. A trigger for the trace can be used to stop the sampling to the trace buffer after a specified delay `Analyzer.TDelay`.

**See also**
- TrBus
- TrBus.state
- ‘Trigger Commands’ in ‘FIRE Emulator for C166 Cell-Based-Core’
- ‘CPU specific Trigger Commands’ in ‘MAC71xx/72xx NEXUS Debugger and Trace’
- ‘Release Information’ in ‘Release History’

TrBus.RESet

Reset setting for trigger bus

**Debugger / FIRE only**

```
Format: TrBus.RESet
```

Resets the settings for the trigger bus.

**See also**
- TrBus
- TrBus.state
TrBus.Set  
Define the target for the incoming trigger

```
Format: TrBus.Set Break | ATrigger [ON | OFF]
```

Selects the target for the incoming trigger signal.

**Break**  
Stop the program execution as soon as the external trigger signal becomes active.

**ATrigger**  
Generate a trigger for the trace as soon as the external trigger signal becomes active. A trigger for the trace can be used to stop the sampling to the trace buffer directly or after a specified delay `Analyzer.TDelay`.

**See also**
- TrBus
- TrBus.state
- ‘Trigger Commands’ in ‘FIRE Emulator for C166 Cell-Based-Core’
- ‘CPU specific Trigger Commands’ in ‘MAC71xx/72xx NEXUS Debugger and Trace’

TrBus.state  
Display settings for the trigger bus

```
Format: TrBus.state  
TrBus.view (deprecated)
```

Displays all settings for the trigger bus.

**See also**
- TrBus
- TrBus.OFF
- TrBus.Trigger
- TrBus.Arm
- TrBus.Out
- TrBus.Connect
- TrBus.RESet
- TrBus.Mode
- TrBus.Set
Stimulate a trigger on the trigger bus.

Format: `TrBus.Trigger`

Stimulates a trigger on the trigger bus.

See also
- `TrBus`
- `TrBus.state`
Overview TrEvent

The event trigger function facilitates the connection between a time counter, or an event counter, and individual trigger events. This function can be particularly useful whenever a trigger should take place after a defined number of events, or when events fail to take place. On ECC8 not event trigger system is available, but the analyzer trigger system will work very similar.
TrEvent.Delay

Define delay

Format:  
TrEvent.Delay <time> | <events>

Defines the number of occurrences or the delay time.

Examples:

; Trigger after 1000 accesses to breakpoint ALPHA

  te.s alpha ; select breakpoint ALPHA
  te.m count ; count mode
  te.d 1000. ; event counter
  te.on ; switch-on

; Trigger after 1000 edges to probe EXTERNAL

  te.s ext ; select external input
  te.m count ; count mode
  te.d 1000. ; event counter
  te.on ; switch-on

; Trigger if breakpoint ALPHA does not respond at least every 10 ms

  te.s alpha ; select breakpoint ALPHA
  te.m nottime ; mode
  te.d 10.ms ; set delay
  te.on ; switch-on

; Trigger 10 ms after breakpoint ALPHA

  te.s alpha ; select breakpoint ALPHA
  te.m thentime ; mode
  te.d 10.ms ; set delay
  te.on ; switch-on

See also

- TrEvent
- TrEvent.view
**TrEvent.Enable**

**ICE only**

If **Always** is selected, the event trigger system will be active (time counters are running) when the emulation has stopped. On default set to **Running**.

```
Format:    TrEvent.Enable Always | Running
```

See also

- **TrEvent**
- **TrEvent.view**
- 'Event Trigger System' in 'ICE User's Guide'
- 'Event Trigger System' in 'ICE User's Guide'

**TrEvent.Init**

**Initialization**

```
Format:    TrEvent.Init
```

Event or time counters are initialized.

See also

- **TrEvent**
- **TrEvent.view**
- 'Event Trigger System' in 'ICE User's Guide'

**TrEvent.MinInit**

**ICE only**

**Initialization**

```
Format:    TrEvent.MinInit
```

The minimum value of the event display is reset to the maximum value.

See also

- **TrEvent**
- **TrEvent.view**
TrEvent.Mode

Select operation mode

ICE only

Format:  

\[ \text{TrEvent.Mode } \text{<mode> [</delay>] } \]

<mode>:

<table>
<thead>
<tr>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThenCycle</td>
</tr>
<tr>
<td>ThenTime</td>
</tr>
<tr>
<td>NotCycle</td>
</tr>
<tr>
<td>NotTime</td>
</tr>
<tr>
<td>AllCycle</td>
</tr>
<tr>
<td>AllTime</td>
</tr>
</tbody>
</table>

Defines the operation mode of the event trigger system.

Count

Event is counted, trigger is executed at event counter set to 0.

Init to 3  Count down  Trigger at zero

;-----------------------------------------------------------------
; Trigger after 1000. rising edges on input probe T0

c.s T0              ; select input probe by counter
te.s signal+         ; route counter signal to event trigger
te.m count           ; use counter function
te.d 1000.           ; define delay
te.on                ; arm

;-----------------------------------------------------------------
; Trigger after 1000. accesses to breakpoint alpha

te.s alpha           ; route counter signal to event trigger
te.m count           ; use counter function
te.d 1000.           ; define delay
te.on                ; arm
ThenCycle

Event is recorded, trigger is executed after a given number of cycles.

<table>
<thead>
<tr>
<th>Event</th>
<th>CPU cycles</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3 2 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Init to 3 Event Count down Trigger at zero</th>
</tr>
</thead>
</table>

;-----------------------------------------------------------------
; Trigger 1000 cycles after breakpoint ALPHA is reached
;-----------------------------------------------------------------

```
te.s alpha          ; select breakpoint
te.m thencycle      ; use delay trigger function
te.d 1000.           ; define delay value
t.e.on               ; arm
```

ThenTime

Event is recorded, trigger is executed after the delay time.

<table>
<thead>
<tr>
<th>Event</th>
<th>Time pulse</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>9 9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Init time Event Count down Trigger at zero</th>
</tr>
</thead>
</table>

;-----------------------------------------------------------------
; Trigger 1000 cycles after falling edges on input probe T0
;-----------------------------------------------------------------

```
c.s T0           ; select input probe by counter
t.e.s signal-    ; route counter signal to event trigger
te.m thentime    ; use delay trigger function
te.d 1.ms         ; define delay value
t.e.on           ; arm
```
**NotCycle**

If no event occurs during a specific number of cycles, the trigger signal will be set.

<table>
<thead>
<tr>
<th>Event</th>
<th>CPU cycles</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 2 1 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

**Trigger**

Init to 3 Event Count Event Count down Trigger at zero

---

**NotTime**

If no event occurs within a specific amount of time, the trigger signal will be activated.

<table>
<thead>
<tr>
<th>Event</th>
<th>Time pulse</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9 8 9 8 7 6 5 4 3 2 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

**Trigger**

Init time Event Count Event Count down Trigger at zero

---

;-----------------------------------------------------------------
; Trigger if the distance between two CPU cycles increases 100 us,
; e.g. by DMA cycles or CPU reset

t.e.m nottim ; set mode
t.e.s always ; select cycle signal for retrigger
t.e.d 100.us ; define time-out value
t.e.on ; arm event trigger system

;-----------------------------------------------------------------
; Trigger if the distance between two timer interrupts increases
; 1.5 ms, e.g. if the interrupt will be disabled for long time

t.e.m nottime ; set mode
t.e.s alpha ; select breakpoint
b.s timerinterrupt /a ; set breakpoint to interrupt routine
t.e.d 1.5 ms ; define time-out value
t.e.on ; arm event trigger system
AllCycle

Trigger is always activated after a certain number of CPU cycles.

<table>
<thead>
<tr>
<th>CPU cycles</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger</td>
<td>Init to 3</td>
<td>Count down</td>
<td>Trigger at zero</td>
<td></td>
</tr>
</tbody>
</table>

AllTime

Trigger is always executed after a fixed time.

<table>
<thead>
<tr>
<th>Time pulse</th>
<th>9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger</td>
<td>Init time</td>
</tr>
</tbody>
</table>

;---------------------------------------------------------------
; Run for a fixed time

te.m alltime

te.delay 1000.s

te.on

go

See also
- TrEvent
- TrEvent.view

▲ 'Event Trigger System' in 'ICE User's Guide'
The event trigger system does not trigger. The input source selection can still be used for the frequency counter or other purposes (Event field in the frequency counter sources).

```plaintext
; Use event trigger unit to measure the interrupt rate

; switch off
; select alpha breakpoint
; set breakpoint to interrupt routine
; select event signal by counter
; set counter mode to frequency

... print "Interrupt rate 
count.value()

... 

; Switch cycle signal to BNC connector

; disarm event trigger
; select signal

; Switch breakpoint signal to BNC connector

; disarm event trigger
; select breakpoint
```

See also

- TrEvent
- TrEvent.view

▲ 'Event Trigger System' in 'ICE User's Guide'
### TrEvent.ON

**ICE only**

<table>
<thead>
<tr>
<th>Format:</th>
<th>TrEvent.ON</th>
</tr>
</thead>
</table>

The event trigger system is activated and will trigger if the defined event occurs.

**See also**
- TrEvent
- TrEvent.view
- 'Event Trigger System' in 'ICE User's Guide'

### TrEvent.RESet

**ICE only**

<table>
<thead>
<tr>
<th>Format:</th>
<th>TrEvent.RESet</th>
</tr>
</thead>
</table>

Reset to original condition, event trigger is disabled.

**See also**
- TrEvent
- TrEvent.view
- 'Event Trigger System' in 'ICE User's Guide'
Defines the event which is selected as source for the event trigger system. This selector can also be used to pass a line to the frequency counter (source field Event) or synchronize other instruments. The output to BNC is on backside.

**Program, Hll, Spot, Read, Write, Alpha, Beta, Charly**

Breakpoints

**DataRead, DataWrite**

Read/Write breakpoints and data fetch.

**ExtData**

External trigger input, sampled at time of CPU data acceptance.

**ExtSynch**

External trigger input, sampled on external cycle.

**ExtComp**

External trigger input, asynchronous recording.

**eXception**

Exception trigger (see eXception.Trigger command).

**TrInput**

Reserved.

**Glitch**

Glitch detector.

**TimeOut**

Timeout bus access (see SYStem.TimeOut command).

**AnalyzerA**

Analyzer trigger outputs.

**EXTernal**

External trigger input signal.

**SIGnal+, SIGnal-**

Universal counter signal.

**ALways**

Cycle signal (occurs once per CPU cycle).
TrEvent.view

State display

Displays all settings and the current status of the event trigger system.

![Diagram of event settings and current status]

The initial and current counter values are displayed in the top line. The value on the right (min) results from the smallest displayed value. This display value does not necessarily correspond to the minimum value because it is only updated when a new value is displayed on the monitor (software, not hardware comparator).

See also
- TrEvent
- TrEvent.view
- 'Event Trigger System' in 'ICE User's Guide'

Format:

```
TrEvent.view
```

```
E68::w.te
```

<table>
<thead>
<tr>
<th>Event</th>
<th>Set</th>
<th>Actual</th>
<th>(min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>564</td>
<td>564</td>
<td></td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>Select</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtData</td>
<td>SIGnal+</td>
</tr>
<tr>
<td>ExtSynch</td>
<td>SIGnal-</td>
</tr>
<tr>
<td>ExtComp</td>
<td>ALways</td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>Mode</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td></td>
</tr>
<tr>
<td>ON</td>
<td></td>
</tr>
<tr>
<td>Init</td>
<td></td>
</tr>
<tr>
<td>RESet</td>
<td></td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>Enable</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allways</td>
<td></td>
</tr>
<tr>
<td>Running</td>
<td></td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>Program</th>
<th>Spot</th>
<th>Read</th>
<th>Write</th>
<th>Alpha</th>
<th>Beta</th>
<th>Charly</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtData</td>
<td>ExtSynch</td>
<td>ExtComp</td>
<td>eXception</td>
<td>TrInput</td>
<td>Glitch</td>
<td>TimeOut</td>
</tr>
<tr>
<td>AnalyzerA</td>
<td>AnalyzerB</td>
<td>RBW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

See also
- TrEvent.Delay
- TrEvent.Mode
- TrEvent.Select
- TrEvent.Init
- TrEvent.OFF
- TrEvent.ON

- 'Event Trigger System' in 'ICE User's Guide'
See also

- TrIn.Clock
- TrIn.RESet
- TrIn.Data
- TrIn.state
- TrIn.Mask
- TrIn.Transient
- TrIn.Normal

**Overview TrIn**
The `TriggerIn` function can be used in conjunction with EXTERNAL probe. It allows to combine external asynchronous or synchronous events and to synchronize these with the internal trigger logic. The trigger logic consists of a programmable comparator, combined with a D-flip-flop with delayed reset for generating a short pulse. Flip-flop data signals and clock signals can be switched to any probe input, internal signals or comparator output. The output of the flip-flop (named EXT), as well as the output of the comparator (named ExtComp) can be used as a trigger event. These signals can also be routed directly to the counter and the event trigger unit for complex triggering. Synchronized to the CPU cycle the state analyzer can trigger on this events. In addition, a "transient" mode is available and can be triggered by status changes to individual input signals.

An EXTERNAL probe is not available on ECC8.

### EXTERNAL-Probe Layout

<table>
<thead>
<tr>
<th>LED</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin</td>
<td>15</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Pin</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pin</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pin</td>
<td>16</td>
<td>14</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

- Pin 1  Line E0
- Pin 3  Line E1
- Pin 5  Line E2
- Pin 7  Line E3
- Pin 9  Line E4
- Pin 11 Line E5
- Pin 13 Line E6
- Pin 15 Line E7
- Pin 2,4,6,8,10,12,14,16  GND
Examples

;-------------------------------------------
; trigger on rising edge of E6 input line

ti.res
ti.data true ; unqualified data
ti.clock E6 + ; select clock
t.s extsynch on ; activate trigger input

;-------------------------------------------
; trigger on high level of E6 input line

ti.res
ti.data E6 ; select data bit
ti.clock cyclemid ; select internal clock
t.s extsynch on ; activate trigger input

;-------------------------------------------
; trigger if E1 is low and E0 is high at
; the end of the cycle

ti.res
ti.mask 0x0xxxxxx01 ; define mask for E0 and E1
ti.data extcomp ; select comparator for data input
ti.clock cycleend ; select internal clock
t.s extsynch on ; activate trigger

;-------------------------------------------
; trigger if E1 is low and E0 is high at
; the end of the cycle

ti.res
ti.mask 0x0xxxxxx01 ; set trigger mask
t.s extdata on ; route comparator to trigger synchronously

;-------------------------------------------
; trigger if E1 and E0 are high for a short time

ti.res
ti.mask 0x0xxxxxx11 ; set comparator mask
t.s extcomp on ; activate asynchronous trigger

;-------------------------------------------
; trigger if E0 and E1 are low at the falling edge of E6

ti.res
ti.mask 0x0xxxxxxx00 ; define mask for data
ti.data extcomp ; select comparator
ti.clock E6 - ; select clock line and polarity
t.s extsynch on ; activate standard trigger

;-------------------------------------------
; trigger on every transient on input E0 and E1
; set mask for transient detection
; set mode
; set trigger synch. to CPU cycles

; trigger if no clock edge arrives within 1.ms on E6

; no data qualifier
; set clock signal to E6
; select EXT for event trigger
; set mode for event trigger
; set delay
; activate trigger

; count the clock edges on E6 while E0 and E1 are both low

; set mask for data qualifier
; select comparator
; select clock
; route signal to counter
; select counter mode
; read-out counter value
; count.value()
Selects the clock signal used by the flip-flop. For more information see chapter Function.

<table>
<thead>
<tr>
<th><strong>&lt;clock&gt;</strong>:</th>
<th>SIGnal</th>
<th>Universal counter signal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CycleMid</td>
<td>SIGnal</td>
<td>Sampling in mid cycle.</td>
</tr>
<tr>
<td></td>
<td>DS-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sample</td>
<td></td>
</tr>
<tr>
<td>CycleEnd</td>
<td>SIGnal</td>
<td>Sampling at the end of the cycle.</td>
</tr>
<tr>
<td></td>
<td>DS-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sample</td>
<td></td>
</tr>
<tr>
<td>ExtComp</td>
<td>SIGnal</td>
<td>External trigger input via comparator.</td>
</tr>
<tr>
<td>Clock</td>
<td>SIGnal</td>
<td>CPU clock.</td>
</tr>
<tr>
<td>E6</td>
<td>SIGnal</td>
<td>External trigger input, Bit 6.</td>
</tr>
<tr>
<td>E7</td>
<td>SIGnal</td>
<td>External trigger input, Bit 7.</td>
</tr>
<tr>
<td>E67</td>
<td>SIGnal</td>
<td>External trigger input, Bit 6 and Bit 7, OR-operation (AND for active-low signals).</td>
</tr>
</tbody>
</table>

See also
- Trln
- Trln.state
- ‘External Trigger Input’ in ‘ICE User's Guide’
**TrIn.Data**

**Format:**  
TrIn.Data <data> [<polarity>]

- **<data>:**  
  - SIGnal
  - True
  - ExtComp
  - E4
  - E5
  - E6
  - E7

- **<polarity>:**  
  - +  
  - -

Selects the data signal used by the flip-flop. For more information see chapter Overview TrIn.

- **SIGnal**  
  Signal frequency counter.

- **True**  
  Active-high signal. It is used, whenever a trigger event is generated on each clock edge (unqualified trigger).

- **ExtComp**  
  EXTERNAL trigger input probe via comparator.

- **E4, E5, E6, E7**  
  EXTERNAL input probe, Bit 4, Bit 5, Bit 6 or Bit 7.

**See also**

- TrIn
- TrIn.state
- 'External Trigger Input' in 'ICE User's Guide'
TrIn.Mask

Define bits

Format:  TrIn.Mask <mask>

The comparator mask can be defined by the characters "0", "1" or "X" (don't care).

Examples:

```
ti.mask 0x0xxxxxxxxl ; true if bit 0 is high
ti.mask 0x0xxxxxxx01 ; true if bit 0 is high and bit 1 is low
ti.mask 0x011111111000000 ; true if all input lines or low
```

See also

- TrIn
- TrIn.state
- 'External Trigger Input' in 'ICE User's Guide'

TrIn.Normal

Level operation

ICE only

Format:  TrIn.Normal

The trigger system searches for the defined events. Transient trigger is switched off.

See also

- TrIn
- TrIn.state
**Trln.RESet**

**Reset command**

ICE only

Format: `Trln.RESet`

Resets the Trln command to the default settings.

See also
- Trln
- Trln.state
- ▲ ‘External Trigger Input’ in ‘ICE User’s Guide’

**Trln.state**

**State display**

ICE only

Format: `Trln.state`

Displays all setups of the Trln command.

```
 Elementary::w.ti

<table>
<thead>
<tr>
<th>state</th>
<th>Clock</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIGNAL</td>
<td>SIGNAL</td>
</tr>
<tr>
<td></td>
<td>CycleMid</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>CycleEnd</td>
<td>ExtComp</td>
</tr>
<tr>
<td></td>
<td>ExtComp</td>
<td>Clock</td>
</tr>
<tr>
<td></td>
<td>E6</td>
<td>E7</td>
</tr>
<tr>
<td></td>
<td>E6</td>
<td>E7</td>
</tr>
<tr>
<td>external</td>
<td>E6</td>
<td>E7</td>
</tr>
<tr>
<td>Normal</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Transient</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RESSet</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The current level of the probe line is displayed in the state field.

See also
- Trln
- Trln.Clock
- Trln.Data
- Trln.Normal
- Trln.RESet
- Trln.Transient
- Trln.Mask
- ▲ TRIN.VALUE()
In case of a transient trigger, the trigger event is generated whenever the level of at least one selected signal line between two sampling points changes. The selection of active inputs results from the comparator mask definition. All inputs labeled "0" or "1" are valid. Polarity has no consequences.

Example:

```
; Trigger event when a change in polarity occurs at line 0 or 2 of the 
; external probe

ti.t                      ; select transient mode
 ti.m 0x1X1                ; define mask
 t.s ed on                 ; switch on
```

See also

- TrIn
- TrIn.state

▲ 'External Trigger Input' in 'ICE User's Guide'
TrMain

The trigger system is used for collecting asynchronous events, converting them into a trigger signal, and passing this trigger signal on to the analyzer and emulation controller. The trigger system can exhibit the following conditions: OFF, ARMED, TRIGGERED or BREAKED. If the trigger system is in the ARMED condition and a trigger event occurs, the TRIGGERED condition will be the result. After the trigger delay sequence is completed the BREAK state will be the result.

Reaching the break state the emulation system or the analyzer will be stopped. The trigger source and trigger address (not ECC8) will be displayed.

There are three trigger modes:

- **Emulator Trigger**: The emulation system is stopped. The real-time emulation will be stopped at the next HLL or ASM line. If the analyzer slave mode is selected, it will be stopped too.

- **Analyzer Trigger**: The state analyzer is stopped, the real-time emulation is not affected.

- **Memory Trigger**: In memory trigger mode the state of emulation memory is locked (write protection). This function is useful only if emulation memory is 'shadowed' to the target memory.

**NOTE:** The trigger system will be blocked, if the PERF function is activated.

Overview TrMain

'TRIGGER Functions (ICE only)’ in ‘General Function Reference’

©1989-2020 Lauterbach GmbH
Trigger Functions

Condition: OFF
Trigger disabled

Condition: ARMED
Trigger disabled

Condition: TRIGGERED
Trigger event has occurred

Condition: BREAKED
Trigger executed

Trigger.OFF
Trigger.ARM
Trigger.Trigger
Trigger.Event
Trigger.Break
Trigger delay finished
Trigger.Sequence
### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIGGER STATE()</td>
<td>State of the trigger system</td>
</tr>
<tr>
<td>0 = OFF</td>
<td></td>
</tr>
<tr>
<td>1 = ARMED</td>
<td></td>
</tr>
<tr>
<td>2 = TRIGGERED</td>
<td></td>
</tr>
<tr>
<td>3 = BREAKED</td>
<td></td>
</tr>
<tr>
<td>TRIGGER ADDRESS()</td>
<td>Address of the trigger event</td>
</tr>
<tr>
<td>TRIGGER COUNT ALPHA()</td>
<td>Value of the trigger counter</td>
</tr>
<tr>
<td>TRIGGER COUNT BETA()</td>
<td></td>
</tr>
<tr>
<td>TRIGGER COUNT CHARLY()</td>
<td></td>
</tr>
<tr>
<td>TRIGGER DELAY TIME()</td>
<td>Value of the trigger delay counter</td>
</tr>
<tr>
<td>TRIGGER DELAY CYCLE()</td>
<td></td>
</tr>
<tr>
<td>TRIGGER DELAY TRACE()</td>
<td></td>
</tr>
<tr>
<td>TRIGGER SOURCE()</td>
<td>Source of last trigger event as hex code (not ECC8).</td>
</tr>
<tr>
<td>TRIGGER CYCLE()</td>
<td>Cycle type of the trigger cycle, bit 0 is read/write (not ECC8).</td>
</tr>
<tr>
<td>TRIGGER BYTES()</td>
<td>Upper four bits contain the byte enable signals on a 32-bit bus (not ECC8).</td>
</tr>
</tbody>
</table>
TrMain.ALways

**Format:**

```
TrMain.ALways [ON | OFF]
```

Ensures that a trigger event is always generated at each cycle during real-time emulation. This function is normally used together with the **TrMain.Delay** option to run for a fixed time.

```
; Program breaks immediately after 100 µs

tm.m emulator ; emulator mode
tm.al on ; trigger at first cycle
tm.d time 100.us ; delay time is 100.us
g ; start emulation

; Analyzer break after 1000 sampled CPU cycles

tm.res ; reset trigger system
tm.m analyzer ; break analyzer only
tm.always on ; trigger immediately
tm.delay trace 1000. ; delay set to 1000 traced cycles
tm.a ; arm trigger system
```

See also
- **TrMain**
- **TrMain.state**
- ‘Trigger System’ in ‘ICE User’s Guide’

---

TrMain.Arm

Release and activate trigger system

ICE only

**Format:**

```
TrMain.Arm
```

After being armed, the trigger system waits for a trigger event. If the option **AutoArm** is set, the trigger system would be activated (ARMED) automatically.

```
tm.a ; Initialize and activate trigger system
```

See also
- **TrMain.AutoInit**
- **TrMain.AutoStart**
- **TrMain**
- **TrMain.state**
- ‘Trigger System’ in ‘ICE User’s Guide’
The trigger logic can be used independently of emulation commands and emulator state. In most cases, however, both trigger and emulation system must work together. Using the \texttt{TrMain.AutoInit} command, the trigger logic will be initialized whenever real-time emulation is started.

\begin{verbatim}
  tm.ai on ; Activate AUTOINIT mode
\end{verbatim}

\textbf{See also}

- \texttt{TrMain.Arm}
- \texttt{TrMain}
- \texttt{TrMain.state}

\footnotesize
\textbf{ICE only}

\section*{TrMain.AutoStart}

\textbf{Automatic trigger initialization}

The trigger logic will be restarted only, if the trigger system is in \textbf{Break} state and emulation is started. No action is done, if the trigger system remains in \textbf{ARMED} or \textbf{TRIGGERED} state.

\begin{verbatim}
  tm.ai on ; activate AUTOSTART mode
  s
  s
  ... ; no trigger
  g ; trigger event is reached
  ... ; trigger is ARMED, the real-time
  g ; emulation is started
\end{verbatim}

\textbf{See also}

- \texttt{TrMain.Arm}
- \texttt{TrMain}
- \texttt{TrMain.state}

\footnotesize
\textbf{ICE only}

\footnotesize
\textbf{ICE only}

\footnotesize
\textbf{ICE only}

\footnotesize
\textbf{ICE only}

\footnotesize
\textbf{ICE only}
TrMain.Break

ICE only

Format: \texttt{TrMain.Break}

Forces the trigger system to enter the \texttt{Break} state. The command will work only, if the CPU generates cycles and the real-time emulation is running.

\textbf{See also}

- TrMain
- TrMain.state

▲ 'Trigger System' in 'ICE User's Guide'

---

TrMain.Count

ICE only

Format: \texttt{TrMain.Count <counter> [<delay>] [ON | OFF]}

\begin{itemize}
  \item \texttt{<counter>}: Alpha | Beta | Charly
  \item \texttt{<delay>}: 0. …
\end{itemize}

The three trigger counters (one in case of ECC8) can be used to count the number of accesses to a certain address. They are directly coupled with the breakpoint memory bits. If a counter is set, it will generate a trigger signal running through zero (not for ECC8).

\begin{verbatim}
E::b.s flags /alpha ; set breakpoint
E::tm.c alpha 100000. ; trigger after 100000 accesses

E::b.s flags /a ; set breakpoint
E::tm.c a 1000. ; set delay count
E::tm.c a off ; switch off trigger function
\end{verbatim}

\textbf{See also}

- TrMain
- TrMain.state

▲ 'Trigger System' in 'ICE User's Guide'
### TrMain.Delay

Set trigger delay

**Format:**

```
TrMain.Delay <counter> [<delay> | ON | OFF]
```

- **<counter>:** TRace | TIme | Cycle
- **<delay>:** 0. ...

Three different delay modes, occurring between trigger event and break state, can be defined:

- **TRace**
  - The number of traced records in trace memory.
- **TIme**
  - The trigger delay can be set from 100 ns to 300 days.
- **Cycle**
  - The trigger delay after CPU cycles.

Trigger delays can be used together. The first delay function which runs to zero will force the trigger system to BREAK state.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tm.d trace 1000.</td>
<td>Delay of 1000 trace cycles</td>
</tr>
<tr>
<td>tm.d time 100.us</td>
<td>Delay of 100 µs</td>
</tr>
<tr>
<td>tm.d cycle 100.</td>
<td>Delay of 100 cycles</td>
</tr>
</tbody>
</table>

### See also

- TrMain
- TrMain.state
- 'Trigger System' in 'ICE User's Guide'
**TrMain.Init**  
ICE only

**Initialize trigger system**

Format:  

```
TrMain.Init
```

This function is similar to that of **TrMain.Arm**. If the trigger system was OFF, it would not automatically be switched-on.

**See also**

- TrMain
- TrMain.state
- 'Trigger System' in 'ICE User's Guide'

---

**TrMain.Mode**  
ICE only

**Select mode**

Format:  

```
TrMain.Mode <mode> [ON | OFF]
```

*<mode>*:

- **Emulator**
  - Real-time emulation is stopped on the next HLL or ASM instruction. If the analyzer is in SLAVE mode, it will be stopped too.
- **Analyzer**
  - Analyzer triggering. Sampling is stopped, the analyzer will be ready for read-out. Real-time emulation is not terminated.
- **Memory**
  - Memory triggering. Emulation memory and flag memory are locked. Access to target system memory is not affected.

**See also**

- TrMain
- TrMain.state
- 'Trigger System' in 'ICE User's Guide'
TrMain.OFF  
Switch off trigger system

ICE only

Format:  TrMain.OFF

The trigger system will not break or trigger, regardless of the other settings.

NOTE:  Program and HLL-breakpoints are not affected, because they are not part of the trigger system.

See also

■ TrMain  ■ TrMain.state

▲ 'Trigger System' in 'ICE User’s Guide'

TrMain.Out  
Output trigger pulse

Debugger only

Format:  TrMain.Out

Outputs a high pulse of 200ns.

An analyzer trigger program of the TRACE32-ICE system controls the reaction to a trigger pulse of the BDM Debugger.

```
trigger.a if busa ; asserts trigger A of the TRACE32-ICE system,
                   ; when the BDM Debugger outputs a trigger pulse
```

See also

■ TrMain  ■ TrMain.state

TrMain.RESet  
Reset trigger system

Format:  TrMain.RESet

Resets trigger system to its original condition.
(E) **ReadData**, **WriteData** and **AnalyzerA** are selected as sources, **AutoStart** is activated and emulator triggering is selected.

See also

- **TrMain**
- **TrMain.state**

▲ 'Trigger System' in 'ICE User's Guide'

---

**TrMain.Set**

**Select trigger sources**

Format:

```
TrMain.Set <source> [ON | OFF]
```

**<source>:**

- Program
- Hll
- Spot
- Read
- Write
- Alpha
- Beta
- Charly
- DataRead
- DataWrite
- ExtData
- ExtSynch
- ExtComp
- eXception
- TrInput
- Glitch
- TimeOut
- AnalyzerA
- RBW
- BUSA

Selects the possible trigger sources. The selected sources are ored together, i.e. the trigger will occur if one source becomes true.

**Program, Hll, Spot, Read, Write, Alpha, Beta, Charly (E)**

This function activates breakpoints (address markers) for triggering (default OFF). The trigger system makes no difference if the bus cycle is data-fetch, op-fetch, or pre-fetch only.

<table>
<thead>
<tr>
<th><strong><code>b.s</code></strong></th>
<th><strong>flags /a</strong></th>
<th>; set single breakpoint to variable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tm.s a on</strong></td>
<td>; activate for trigger (read/write/opfetch)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong><code>b.s</code></strong></th>
<th><strong>v.range(flags) /a</strong></th>
<th>; set single breakpoint to data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tm.s a on</strong></td>
<td>; activate trigger function (read/write/; opfetch)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong><code>b.s</code></strong></th>
<th><strong>stack-0x100</strong></th>
<th>; protect against stack overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tm.s a on</strong></td>
<td>; activate trigger function</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong><code>b.s</code></strong></th>
<th><strong>0x10000--0xffffffff</strong></th>
<th>; protect range outside of working area</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tm.s a on</strong></td>
<td>; activate trigger function</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong><code>b.s</code></strong></th>
<th><strong>x:0x0--0xffffffff</strong></th>
<th>; protect selective (one bus cycle type only)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tm.s a on</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**DataRead, DataWrite (E)**
Data read (write) to the Read (Write) breakpoints generate a trigger signal. Opfetch or other CPU cycles are ignored. This function is activated on default, because it will be used as the standard breakpoint function for variable access. To set up memory protection automatically, use the **Break.SetSec** command.

```
b.s flags /w ; set write breakpoint to variable
b.s flags /r ; set read breakpoint to variable
b.s flags /r /w ; set read and write breakpoint to variable
b.s flags ; set read/write/program breakpoint to variable
b.s v.range(initdata) /w ; protect against write to initialized data
 tm.s wd on ; area
tm.s p_begin--p_end /w ; protect against writing to program area
tm.s p_begin--p_end /r /w ; protect against data access to program area
tm.s rd on ; activate breakpoint function
tm.s wd on ; for read and write
b.s ss ; set breakpoints on data sections
g ; run real-time
```

**ExtData** (E)

External trigger input, sampled synchronous together with the CPU data bus sampling. The trigger message will qualify exactly the cycle of the trigger event. The trigger level is defined within the **TrIn.Mask** command.

```
AS- Sample Trigger

; Trigger if both pin 0 and 1 of EXTERNAL probe are high

  ti.mask 0x00xxxxx11 ; set trigger mask
tm.s extdata on ; activate

```

©1989-2020 Lauterbach GmbH
ExtSynch (E)

External trigger input, sampled on external clock cycle and synchronized to the CPU cycle signal. For more information see TrIn commands.

```
; Trigger if pin 0 is high in the rising edge of pin 7 of
; the EXTERNAL probe

ti.mask 0x0xxxxxxxx1 ; select pin 0 only
ti.data extcomp + ; select comparator for data
ti.clock e7 + ; select input 7 for clock
tm.s extsynch on
```
**ExtComp** (E)

External trigger input, asynchronous triggering. See **TrIn** command.

```plaintext
; Trigger if pin 0 and 1 of the external probe are high at the same time
ti.mask 0x0xxxxxx11 ; select pin 0 and 1 only
tm.s extcomp on ; activate asynchronous trigger
```

**eXception** (E)

Exception trigger (see **eXception.Trigger** command).

```plaintext
; Trigger asynchronous to RESET signal
x.trigger off
x.trigger reset on
t.s x on
```

**TrInput** (E)

Reserved for future use.

**Glitch** (E)

The glitch detector output is used as trigger signal.

```plaintext
; Trigger to glitch on external probe pin 0
c.select e0 ; select probe input to glitch detector
tm.s glitch on ; activate glitch trigger
g wait trigger.state()>=2 ; start real-time emulation
print "Glitch found when accessing address "
trigger.address()
```

©1989-2020 Lauterbach GmbH
TimeOut (E)

Time-out on bus access (see SYStem.TimeOut command).

```plaintext
; Trigger to bus time-out
sys.to 10.us ; set time-out value
tm.s timeout on ; activate time-out trigger
g wait trigger.state()>=2 ; start real-time emulation
print "No READY signal generated on address ", trigger.address()
```

AnalyzerA (E)

Analyzer trigger output.

RWB (E)

Read-before-write triggering. This function is used to detect accesses to uninitialized variables or memory areas. To preset the flags in order to the linker information use the FLAG.SetSec command.

```plaintext
; Read-before-write trigger
flag.setsec ; set write flag in program and initialized data areas
tm.s rbw on ; activate read-before-write trigger
g wait trigger.state()>=2 ; start real-time emulation
print "Uninitialized variable on address ", trigger.address()
```

See also

- TrMain
- TrMain.state
- 'Trigger System' in 'ICE User's Guide'
- 'Release Information' in 'Release History'
**TrMain.state**

**Trigger state display**

Format: `TrMain.state`

See also

- `TrMain`
- `TrMain.AutoStart`
- `TrMain.Init`
- `TrMain.RESet`
- `TRIGGER.ADDRESS()`
- `TRIGGER.SOURCE()`

▲ 'Trigger System' in 'ICE User's Guide'

**TrMain.Trigger**

**Trigger**

Format: `TrMain.Trigger`

Execute trigger manually or by script.

See also

- `TrMain`
- `TrMain.state`

▲ 'Trigger System' in 'ICE User's Guide'
The **TrOnchip** command group provides low-level access to the on-chip debug register.

### TrOnchip

#### TrOnchip.CONVert

**Adjust range breakpoint in onchip registers**

**Format:**

```
TrOnchip.CONVert
```

---

**See also**

- TrOnchip
- TrOnchip.state

- 'TrOnchip Commands' in 'XC2000/XC16x/C166CBC Debugger'
- 'TrOnchip Commands in 'DSP56K Debugger'
- 'CPU specific TrOnchip Commands' in 'C6000 Debugger'
- 'ARM Specific TrOnchip Commands' in 'ARMv8-A/R Debugger'
- 'CPU specific TrOnchip Commands' in 'AVR32 Debugger and NEXUS Trace'
- 'CPU specific TrOnchip Commands' in 'ARMv8-A/R Debugger'
- 'Beyond Specific TrOnchip Commands' in 'Beyond Debugger and Trace'
- 'TrOnchip Commands' in 'C5000 Debugger'
- 'TrOnchip Commands' in 'C6000 Debugger'
- 'TrOnchip Commands' in 'C7000 Debugger'
- 'TrOnchip Commands in ‘CEVA-X Debugger’
- 'ARM specific TrOnchip Commands' in 'Cortex-M Debugger'
- 'CPU specific TrOnchip Commands in ‘eTPU Debugger and Trace’
- 'General TrOnchip Commands’ in ‘GTM Debugger and Trace’

---

**See also**

- 'CPU specific TrOnchip Commands’ in ‘CPU32/ColdFire Debugger and Trace’
- 'ARM Specific TrOnchip Commands’ in ‘ARM Debugger’
- 'ARM specific TrOnchip Commands’ in ‘ARMv8-A/R Debugger’
- ‘Command Reference: TrOnchip’ in ‘Gorivva MPC5xxx/SPC5xx Debugger and NEXUS Trace’
- ‘CPU specific TrOnchip Commands’ in ‘PQIII Debugger’
- ‘CPU specific TrOnchip Commands’ in ‘GorIQ Debugger and NEXUS Trace’
- ‘CPU specific TrOnchip Commands’ in ‘RH850 Debugger and Trace’
- ‘Nexus specific TrOnchip Commands’ in ‘RH850 Debugger and Trace’
- ‘TrOnchip’ in ‘StarCore Debugger and Trace’
- ‘CPU specific TrOnchip Commands’ in ‘TriCore Debugger and Trace’
- ‘CPU specific TrOnchip Commands - Onchip Triggers’ in ‘Intel® x86/x64 Debugger’
- ‘Specific TrOnchip Commands’ in ‘FIRE Emulator for C166 Cell-Based-Core’
- ‘On-chip Trigger System’ in ‘FIRE Emulator for C166 Family’
- ‘Release Information’ in ‘Release History’

---

©1989-2020 Lauterbach GmbH
**TrOnchip.RESet**

Reset settings to defaults

Format: `TrOnchip.RESet`

Set on-chip trigger system to initial state.

---

See also

- `TrOnchip`  
- `TrOnchip.state`

- 'TrOnchip Commands' in 'H8S/23x9 Debugger'
- 'TrOnchip Commands' in 'MCS08 Debugger'
- 'TrOnchip Commands' in 'MCS12 Debugger'
- 'TrOnchip' in 'M32R Debugger and Trace'
- 'TrOnchip Commands' in 'M8051EW Debugger'
- 'Trigger On-chip Commands' in 'M-Core Debugger'
- 'TrOnchip Commands' in 'MicroBlaze Debugger and Trace'
- 'On-chip Breakpoints' in 'MIPS Debugger and Trace'
- 'CPU specific TrOnchip Commands' in 'MMDSP Debugger'
- 'Command Reference: TrOnchip' in 'Gorivva MPC5xxx/SPC5xx Debugger and NEXUS Trace'
- 'MSP430 Specific TrOnchip Commands' in 'MSP430 Debugger'
- 'TrOnchip Commands' in 'NIOS II Debugger and Trace'
- 'CPU specific TriggerOnchip Commands' in 'PCP Debugger Reference'
- 'CPU specific TrOnchip Commands' in 'PPC400/PPC440 Debugger and Trace'
- 'CPU specific TrOnchip Commands' in 'PPC600 Family Debugger'
- 'CPU specific TrOnchip Commands' in 'MPC5xx/Bxx Debugger and Trace'
- 'CPU specific TrOnchip Commands' in 'PQIII Debugger'
- 'CPU specific TrOnchip Commands' in 'GorIQ Debugger and NEXUS Trace'
- 'TrOnchip Commands' in 'R8051XC Debugger'
- 'CPU specific TrOnchip Commands' in 'RH850 Debugger and Trace'
- 'TrOnchip Commands' in 'RX Debugger'
- 'CPU specific TrOnchip Commands' in 'SH2, SH3 and SH4 Debugger'
- 'TrOnchip' in 'StarCore Debugger and Trace'
- 'CPU specific TrOnchip Commands' in 'TriCore Debugger and Trace'
- 'TrOnchip Commands' in 'V850 Debugger and Trace'
- 'TrOnchip Commands' in 'XC800 Debugger'
- 'CPU specific TrOnchip Commands' in 'ZSP Debugger'
- 'Specific TrOnchip Commands' in 'FIRE Emulator for C166 Cell-Based-Core'
- 'CPU specific TrOnchip Commands' in 'FIRE Emulator for HC12/MCS12'
- 'Specific TrOnchip Commands' in 'FIRE Emulator for SH2'
- 'CPU specific TrOnchip Commands' in 'ARM and XSCALE Monitor'
- 'TrOnchip' in 'C166 Monitor'
- 'TrOnchip' in 'TriCore Monitor'
- 'TrOnchip Commands' in 'x186 Monitor'
- 'ARM specific TrOnchip Commands' in 'MAC71xx/72xx NEXUS Debugger and Trace'
- 'CPU specific TrOnchip Commands' in 'MMDSP NEXUS Debugger and Trace'
- 'Release Information' in 'Release History'
- 'On-chip Breakpoints/Actionpoints' in 'Simulator for ARC'

©1989-2020 Lauterbach GmbH
TrOnchip.Set

Break on event

See also

- TrOnchip
- TrOnchip.state
- ‘ARM Specific TrOnchip Commands’ in ‘ARM Debugger’
- ‘ARM specific TrOnchip Commands’ in ‘ARMv8-A/-R Debugger’

©1989-2020 Lauterbach GmbH
^ `ARM specific TrOnchip Commands` in 'Cortex-M Debugger'
^ `CPU specific TrOnchip Commands` in 'eTPU Debugger and Trace'
^ `Command Reference: TrOnchip` in 'Corinva MPC5xx/SPC5xx Debugger and NEXUS Trace'
^ `CPU specific TrOnchip Commands` in 'PPC400/PPC440 Debugger and Trace'
^ `CPU specific TrOnchip Commands` in 'MPC5xx/8xx Debugger and Trace'
^ `CPU specific TrOnchip Commands` in 'PQIIl Debugger'
^ `CPU specific TrOnchip Commands` in 'QorIQ Debugger and NEXUS Trace'
^ `Specific TrOnchip Commands` in 'FIRE Emulator for SH2'
^ `ARM specific TrOnchip Commands` in 'MAC71xx/72xx NEXUS Debugger and Trace'

**TrOnchip.state**  
Display onchip trigger window

```
Format:       TrOnchip.state
```

Displays a window with the state of the on-chip trigger setting.

**See also**

- `TrOnchip`  
- `TrOnchip.CONVert`  
- `TrOnchip.RESet`  
- `TrOnchip.Set`

^ `TrOnchip Commands` in 'XC2000/XC16x/C166CBC Debugger'
^ `CPU specific TrOnchip Commands` in 'CPU32/ColdFire Debugger and Trace'
^ `AndesCore Specific TrOnchip Commands` in 'Andes Debugger'
^ `APEX Specific TrOnchip Commands` in 'APEX Debugger'
^ `On-chip Breakpoints/Actionpoints` in 'ARC Debugger'
^ `ARM Specific TrOnchip Commands` in 'ARM Debugger'
^ `ARM specific TrOnchip Commands` in 'ARMv8-A/-R Debugger'
^ `CPU specific TrOnchip Commands` in 'AVR32 Debugger and NEXUS Trace'
^ `CPU specific TrOnchip Commands` in 'AVR8 Debugger'
^ `Beyond Specific TrOnchip Commands` in 'Beyond Debugger and Trace'
^ `TrOnchip Commands` in 'C2000 Debugger'
^ `TrOnchip Commands` in 'C5000 Debugger'
^ `TrOnchip Commands` in 'C6000 Debugger'
^ `TrOnchip Commands` in 'C7000 Debugger'
^ `TrOnchip Commands` in 'CEVA-X Debugger'
^ `ARM specific TrOnchip Commands` in 'Cortex-M Debugger'
^ `CPU specific TrOnchip Commands` in 'eTPU Debugger and Trace'
^ `TrOnchip Commands` in 'H8S/23x9 Debugger'
^ `TrOnchip Commands` in 'MCS12 Debugger'
^ `TrOnchip Commands` in 'Hexagon Debugger'
^ `IPU Specific TrOnchip Commands` in 'IPU Debugger'
^ `TrOnchip` in 'M32R Debugger and Trace'
^ `TrOnchip Commands` in 'M8051EW Debugger'
^ `Trigger On-chip Commands` in 'M-Core Debugger'
^ `Meta specific Implementations` in 'Meta Debugger'
^ `Mico32 specific TrOnchip Commands` in 'Mico32 Debugger'
^ `TrOnchip Commands` in 'MicroBlaze Debugger and Trace'
^ `On-chip Breakpoints` in 'MIPS Debugger and Trace'
^ `CPU specific TrOnchip Commands` in 'MMDSP Debugger'
^ `MSP430 Specific TrOnchip Commands` in 'MSP430 Debugger'
^ `TrOnchip Commands` in 'NIOS II Debugger and Trace'
^ `TrOnchip Commands` in 'CEVA-Oak/Teak/TeakLite Debugger'
^ `CPU specific TrOnchip Commands` in 'PPC400/PPC440 Debugger and Trace'
^ `CPU specific TrOnchip Commands` in 'PPC600 Family Debugger'
^ `CPU specific TrOnchip Commands` in 'MPC5xx/8xx Debugger and Trace'
^ `TrOnchip Commands` in 'R8051XC Debugger'

©1989-2020 Lauterbach GmbH
‘CPU specific TrOnchip Commands’ in ‘RH850 Debugger and Trace’
‘TrOnchip Commands’ in ‘RX Debugger’
‘CPU specific TrOnchip Commands’ in ‘SH2, SH3 and SH4 Debugger’
‘TrOnchip’ in ‘StarCore Debugger and Trace’
‘CPU specific TrOnchip Commands’ in ‘STRED Debugger and Trace’
‘CPU specific TrOnchip Commands’ in ‘TriCore Debugger and Trace’
‘TrOnchip Commands’ in ‘V850 Debugger and Trace’
‘TrOnchip Commands’ in ‘XC800 Debugger’
‘CPU specific TrOnchip Commands’ in ‘XTENSA Debugger’
‘CPU specific TrOnchip Commands’ in ‘ZSP Debugger’
‘CPU specific TrOnchip Commands’ in ‘FIRE Emulator for HC12/MCS12’
‘Specific TrOnchip Commands’ in ‘FIRE Emulator for SH2’
‘CPU specific TrOnchip Commands’ in ‘ARM and XSCALE Monitor’
‘TrOnchip’ in ‘C166 Monitor’
‘TrOnchip Commands’ in ‘SH2 Monitor’
‘TrOnchip Commands’ in ‘x186 Monitor’
‘ARM specific TrOnchip Commands’ in ‘MAC71xx/72xx NEXUS Debugger and Trace’
‘CPU specific TrOnchip Commands’ in ‘MMDSP NEXUS Debugger and Trace’
‘Release Information’ in ‘Release History’
‘On-chip Breakpoints/Actionpoints’ in ‘Simulator for ARC’
**TrPOD**

**TrPOD**  
Trigger probe

---

### TrPOD.Clock

**Defines data mask**

Format:  

```
TrPOD.Clock [mask]
```

The clock mask is defined. Every input line can be high or low or don’t care.

---

### TrPOD.ClockPOL

**Defines data polarity**

Format:  

```
TrPOD.ClockPOL [polarity]
```

<polarity>:  
+  |  -

The clock polarity can be set to true or false.

---

©1989-2020 Lauterbach GmbH

General Commands Reference Guide T 557
TrPOD.Data

Defines data mask

Format: \texttt{TrPOD.Data \[<mask>\]}

The data mask is defined. Every input line can be high or low or don’t care.

See also

- \texttt{TrPOD}
- \texttt{TrPOD.state}

\textbf{TrPOD.DataPOL}

Defines data polarity

Format: \texttt{TrPOD.DataPOL \[<polarity>\]}

\textit{<polarity>}:
\begin{itemize}
\item \texttt{+}
\item \texttt{-}
\end{itemize}

The data polarity can be set to true or false.

See also

- \texttt{TrPOD}
- \texttt{TrPOD.state}
TrPOD.Mode

Defines data polarity

<table>
<thead>
<tr>
<th>Format:</th>
<th>TrPOD.Mode [&lt;mode&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;mode&gt;:</td>
<td>DATA</td>
</tr>
<tr>
<td></td>
<td>CLOCK</td>
</tr>
<tr>
<td></td>
<td>SYNC</td>
</tr>
<tr>
<td></td>
<td>LONGER</td>
</tr>
<tr>
<td></td>
<td>SHORTER</td>
</tr>
<tr>
<td></td>
<td>GLITCH</td>
</tr>
<tr>
<td></td>
<td>GLITCH+</td>
</tr>
<tr>
<td></td>
<td>GLITCH-</td>
</tr>
</tbody>
</table>

The state display shows all the settings of the trigger probe and the level of the input pins.

- **DATA**: Asynchronous trigger on inputs with data comparator
- **CLOCK**: Asynchronous trigger on inputs with clock comparator
- **SYNC**: Synchronous trigger
- **LONGER**: Pulse width trigger when pulse exceeds time
- **SHORTER**: Pulse width trigger when pulse width below time limit
- **GLITCH**: Glitch trigger on both edges
- **GLITCH+**: Glitch trigger on positive glitch
- **GLITCH-**: Glitch trigger on negative glitch

See also

- TrPOD
- TrPOD.state
TrPOD.OFF

Switch off

The trigger probe is disabled.

See also
- TrPOD
- TrPOD.state

TrPOD.ON

Switch on

The trigger probe is enabled.

See also
- TrPOD
- TrPOD.state

TrPOD.RESet

Reset command

The trigger probe is initialized to the default setup condition

See also
- TrPOD
- TrPOD.state
Using this command the operating mode of the analyzer may be selected. During operation this command displays the current state of the analyzer.

<table>
<thead>
<tr>
<th>B::tp</th>
<th>Mode</th>
<th>Data</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OFF</td>
<td>0xxxxxxx!</td>
<td>0xxxxxxx!</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>0xxxxxxx!</td>
<td>0xxxxxxx!</td>
</tr>
<tr>
<td></td>
<td>RESet</td>
<td>0xxxxxxx!</td>
<td>0xxxxxxx!</td>
</tr>
</tbody>
</table>

state
- Displays the current signal levels on the input lines.
- OFF: Indicates that the trigger is deactivated
- ON: Indicates that the trigger probe is activated

See also
- TrPOD
- TrPOD.Clock
- TrPOD.ClockPOL
- TrPOD.Data
- TrPOD.DataPOL
- TrPOD.MODE
- TrPOD.OFF
- TrPOD.RESet
- TrPOD.Time

▲ 'Release Information' in 'Release History'
TrPOD.Time

Defines the time for the pulse width trigger

Format:

TrPOD.Time [<time>]

The time limit for the pulse width detection can be set between 20 ns and 6 ms.

See also

- TrPOD
- TrPOD.state