

General Commands Reference Guide S

General Commands Reference Guide S

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents	
General Commands	
General Commands Reference Guide S	1
History	14
SELFTEST	15
SELFTEST	15
Execute selftest operation	15
SETUP	16
SETUP	16
Setup commands	16
SETUP.ALIST	17
Default analyzer display	17
SETUP.ALIST.RESet	17
Reset analyzer display	17
SETUP.ALIST.set	17
Default analyzer display	17
SETUP.BreakPointTableWalk	17
Set up MMU translation for breakpoints	17
SETUP.BreakTransfer	18
Breakpoint synchronization	18
SETUP.COLORCORE	18
Enable coloring for core-specific info in SMP systems	18
SETUP.DIS	19
Disassembler configuration	19
SETUP.DUMP	20
Defaults for hex-dumps	20
SETUP.EMUPATH	21
Emulation softkeys configuration	21
SETUP.GoOnPaused	21
Route go to paused core	21
SETUP.IMASKASM	22
Mask interrupts during assembler step	22
SETUP.IMASKHLL	22
Mask interrupts during HLL step	22
SETUP.LISTCLICK	23
Double-click source line symbol to run this action	23
SETUP.PROCESS	24
Processing percentage in statistics window	24
SETUP.SIMULINK	24
Deprecated command	24
SETUP.StepAllCores	25
Force single stepping on all cores	25
SETUP.StepAtBreakPoint	26
Single step to skip breakpoint	26
SETUP.StepAutoAsm	26
HLL steps stops at assembler code	26
SETUP.StepBeforeGo	27
Single step before go	27
SETUP.StepByStep	27
Single step HLL lines	27
SETUP.StepNoBreak	27
Stepping HLL lines with disabled breakpoints	27
SETUP.StepOnPaused	28
Route step to selected core	28
SETUP.StepTrace	28
Show stepping trail in list window	28
SETUP.StepWithinBreakpoints	28
Multi-core step on SMP systems	28
SETUP.StepWithinTask	29
Task selective stepping	29
SETUP.sYmbol	29
Length of symbols	29

SETUP.TIMEOUT	Define emulation monitor time-out	30
SETUP.Var	Defaults for the Var commands	31
SETUP.VarCall	Define call dummy routine	35
SETUP.VarPtr	Limit pointer access	36
SETUP.VerifyBreakSet	Additional verification for software breakpoints	36
SIM		37
SIM	TRACE32 Instruction Set Simulators	37
SIM.AREA	Selects area for simulation output	37
SIM.CACHE	Cache/MMU simulation and more	38
SIM.CACHE.Allocation	Define the cache allocation technique	39
SIM.CACHE.Mode	Define memory coherency strategy	40
SIM.CACHE.MPURegions	Specify MPU regions	40
SIM.CACHE.OFF	Disable cache and MMU simulation	41
SIM.CACHE.ON	Enable cache and MMU simulation	41
SIM.CACHE.Replacement	Define the replacement strategy	41
SIM.CACHE.SETS	Define the number of cache/TLB sets	43
SIM.CACHE.state	Display cache and MMU settings	44
SIM.CACHE.Tags	Define address mode for cache lines	45
SIM.CACHE.TRACE	Select simulator trace method	45
SIM.CACHE.View	Analysis of memory accesses for cache simulation	46
SIM.CACHE.ViewTLB	Analysis of TLB accesses for MMU simulation	46
SIM.CACHE.WAYS	Define number of cache ways	47
SIM.CACHE.Width	Define width of cache line	47
SIM.command	Issue command to simulation model	48
SIM.INTerrupt	Trigger interrupt	48
SIM.List	List loaded simulator models	49
SIM.LOAD	Load simulator module	49
SIM.RESet	Reset TRACE32 Instruction Set Simulator	49
SIM.UNLOAD	Unload simulator module	50
SLTrace		51
SLTrace	Trace sink for SYStem.LOG events	51
SLTrace.state	Display configuration window	52
SNOOPer		53
SNOOPer	Sample-based trace	53
SNOOPer-specific Trace Commands		54
SNOOPer.<specific_cmds>	Overview of SNOOPer-specific commands	54
SNOOPer.CORE	Select cores for PC snooping	54
SNOOPer.ERRORSTOP	Set behavior on sampling errors	55
SNOOPer.Mode	Set operation mode of SNOOPer trace	56
SNOOPer.PC	Enable PC snooping	60
SNOOPer.Rate	Select sampling rate	61
SNOOPer.SELect	Define address for monitoring	61

SNOOPer.SIZE	Define trace buffer size	63
SNOOPer.TDelay	Define trigger delay	63
SNOOPer.TOut	Define the trigger destination	64
SNOOPer.TValue	Define data value for trigger	65
Generic SNOOPer Trace Commands		66
SNOOPer.ACCESS	Define access path to program code for trace decoding	66
SNOOPer.Arm	Arm the trace	66
SNOOPer.AutoArm	Arm automatically	66
SNOOPer.AutoInit	Automatic initialization	66
SNOOPer.BookMark	Set a bookmark in trace listing	66
SNOOPer.BookMarkToggle	Toggles a single trace bookmark	66
SNOOPer.Chart	Display trace contents graphically	67
SNOOPer.Chart.DistriB	Distribution display graphically	67
SNOOPer.Chart.sYmbol	Symbol analysis	67
SNOOPer.Chart.VarState	Variable activity chart	67
SNOOPer.ComPare	Compare trace contents	67
SNOOPer.DISable	Disable the trace	67
SNOOPer.DRAW	Plot trace data against time	67
SNOOPer.DRAW.channel	Plot no-data values against time	68
SNOOPer.DRAW.Var	Plot variable values against time	68
SNOOPer.EXPORT	Export trace data for processing in other applications	68
SNOOPer.FILE	Load a file into the file trace buffer	68
SNOOPer.Find	Find specified entry in trace	68
SNOOPer.FindAll	Find all specified entries in trace	68
SNOOPer.FindChange	Search for changes in trace flow	68
SNOOPer.Get	Display input level	69
SNOOPer.GOTO	Move cursor to specified trace record	69
SNOOPer.Init	Initialize trace	69
SNOOPer.List	List trace contents	69
SNOOPer.ListVar	List variable recorded to trace	69
SNOOPer.LOAD	Load trace file for offline processing	69
SNOOPer.OFF	Switch off	69
SNOOPer.PROfileChart	Profile charts	69
SNOOPer.PROfileChart.COUNTER	Display a profile chart	70
SNOOPer.PROfileSTATistic	Statistical analysis in a table versus time	70
SNOOPer.PROTOcol	Protocol analysis	70
SNOOPer.PROTOcol.Chart	Graphic display for user-defined protocol	70
SNOOPer.PROTOcol.Draw	Graphic display for user-defined protocol	70
SNOOPer.PROTOcol.EXPORT	Export trace buffer for user-defined protocol	70
SNOOPer.PROTOcol.Find	Find in trace buffer for user-defined protocol	70
SNOOPer.PROTOcol.list	Display trace buffer for user-defined protocol	71
SNOOPer.PROTOcol.PROfileChart	Profile chart for user-defined protocol	71
SNOOPer.PROTOcol.PROfileSTATistic	Profile chart for user-defined protocol	71

SNOOPer.PROTOcol.STATistic	Display statistics for user-defined protocol	71
SNOOPer.REF	Set reference point for time measurement	71
SNOOPer.RESet	Reset command	71
SNOOPer.SAVE	Save trace for postprocessing in TRACE32	71
SNOOPer.SelfArm	Automatic restart of trace recording	72
SNOOPer.SnapShot	Restart trace capturing once	72
SNOOPer.state	Display trace configuration window	72
SNOOPer.STATistic	Statistic analysis	72
SNOOPer.STATistic.DistriB	Distribution analysis	72
SNOOPer.Timing	Waveform of trace buffer	72
SNOOPer.TRACK	Set tracking record	72
SNOOPer.View	Display single record	73
SNOOPer.ZERO	Align timestamps of trace and timing analyzers	73
SPE		74
SPE	Signal Processing eXtension (SPE)	74
SPE.Init	Initialize SPE registers	74
SPE.Set	Modify SPE registers	74
SPE.view	Display SPE register window	75
SSE		76
SSE	SSE registers (Streaming SIMD Extension)	76
SSE.Init	Initialize SSE registers	76
SSE.Set	Modify SSE registers	76
SSE.view	Display SSE registers	77
StatCol		78
StatCol	Statistics collector	78
Step		79
Step	Single-step	79
Step.Asm	Assembler single-step	79
Step.Back	Step backwards	79
Step.BackChange	Step back until expression changes	80
Step.BackOver	Step back over call	80
Step.BackTill	Step back until expression true	80
Step.Change	Step until expression changes	81
Step.Diverge	Step to next unreached line	82
Step.Hll	Step in HLL-mode	84
Step.Mix	Step in mixed-mode	84
Step.Over	Step over call	85
Step.single	Single-step	85
Step.Till	Step until expression true	86
STM		87
STM	System trace configuration	87

STOre		88
STOre	Store settings as PRACTICE script	88
SVE		92
SVE	Access the scalable vector extension SVE	92
SVE.Init	Initialize SVE registers	92
SVE.RESet	Reset SVE settings	92
SVE.Set	Modify SVE registers	92
SVE.view	Display SVE registers	93
sYmbol		94
sYmbol	Debug symbols	94
Overview sYmbol		94
sYmbol.AddInfo	Provide additional symbolic information	96
sYmbol.AddInfo.Address	Add symbol information to fixed address	98
sYmbol.AddInfo.Delete	Delete information	99
sYmbol.AddInfo.LINK	Define information for 'sYmbol.AddInfo' commands	100
sYmbol.AddInfo.List	List additional information	101
sYmbol.AddInfo.LOADASAP2	Load scaling information from ASAP2 file	101
sYmbol.AddInfo.Member	Add information to member of struct	102
sYmbol.AddInfo.RESet	Remove all additional information	104
sYmbol.AddInfo.Type	Add information to a data type	104
sYmbol.AddInfo.Var	Add information to a variable	105
sYmbol.AutoLOAD	Automated loading of symbols	106
sYmbol.AutoLOAD.CHECK	Update autoloader table	107
sYmbol.AutoLOAD.CHECKCoMmanD	Configure dynamic autoloader	108
sYmbol.AutoLOAD.CHECKDLL	Configure automatic DLL file loader	109
sYmbol.AutoLOAD.CHECKEPOC	Dynamic autoloader for Symbian	110
sYmbol.AutoLOAD.CHECKLINUX	Configure autoloader for Linux debugging	110
sYmbol.AutoLOAD.CHECKQNX	Configure autoloader for QNX debugging	111
sYmbol.AutoLOAD.CHECKUEFI	Configure autoloader for UEFI debugging	111
sYmbol.AutoLOAD.CHECKWIN	Configure autoloader	112
sYmbol.AutoLOAD.CHECKWINCE	Configure autoloader	112
sYmbol.AutoLOAD.CLEAR	Remove symbol information	113
sYmbol.AutoLOAD.config	Configure symbol autoloader	113
sYmbol.AutoLOAD.Create	Create entry for autoloader table	114
sYmbol.AutoLOAD.Delete	Delete autoloader entries	114
sYmbol.AutoLOAD.List	List autoloader table	115
sYmbol.AutoLOAD.LOADEPOC	Definition for static autoloader for Symbian	116
sYmbol.AutoLOAD.RESet	Reset autoloader	117
sYmbol.AutoLOAD.SET	Mark symbol information manually as loaded	117
sYmbol.AutoLOAD.TOUCH	Initiate automatic loading by command	118
sYmbol.Browse	Browse symbols	119
sYmbol.Browse.Class	Browse classes	119
sYmbol.Browse.Enum	Browse enumeration types	119

sYmbol.Browse.Function	Browse functions	120
sYmbol.Browse.Module	Browse modules	121
sYmbol.Browse.MVar	Browse module variables	122
sYmbol.Browse.name	Browse symbols (flat)	122
sYmbol.Browse.SFunction	Browse functions	123
sYmbol.Browse.SModule	Browse modules	125
sYmbol.Browse.SOURCE	Browse source	126
sYmbol.Browse.Struct	Browse containers for different variable types	127
sYmbol.Browse.sYmbol	Browse symbols	128
sYmbol.Browse.Type	Browse HLL types	129
sYmbol.Browse.TypeDef	Browse type definitions	130
sYmbol.Browse.Union	Browse unions	130
sYmbol.Browse.Var	Browse variables	131
sYmbol.CASE	Set symbol search mode	132
sYmbol.CHECK	Check database	132
sYmbol.Class	View class hierarchy	133
sYmbol.CLEANUP	Workarounds for redundant symbol information	134
sYmbol.CLEANUP.DOUBLES	Make ambiguous symbols unique	135
sYmbol.ColorCode	Enable color coding	135
sYmbol.ColorDef	Specify keyword colors	136
sYmbol.CREATE	Create and modify user-defined symbols	136
sYmbol.CREATE.ATTRibute	Create user-defined attribute	137
sYmbol.CREATE.Done	Finish symbol creation	137
sYmbol.CREATE.Function	Create user-defined function	138
sYmbol.CREATE.Label	Create user-defined symbol	139
sYmbol.CREATE.LocalVar	Create user-defined local variable	139
sYmbol.CREATE.MACRO	Create user-defined macro	140
sYmbol.CREATE.Module	Create user-defined module	140
sYmbol.CREATE.RESet	Erase all user-defined symbols	141
sYmbol.CREATE.Var	Create user-defined variable	141
sYmbol.CUTLINE	Limit size of text blocks	142
sYmbol.Delete	Delete symbols of one program	142
sYmbol.DeleteMACRO	Delete macro information	143
sYmbol.DeletePATtern	Delete labels from symbol database using wildcards	143
sYmbol.DEMangle	C++ demangler	143
sYmbol.DEOBFUSCATE	Deobfuscate global and static symbol	144
sYmbol.DONE	Finish load of symbols	144
sYmbol.ECA	ECA file management	145
sYmbol.ECA.BINary	Static preprocessing for code coverage	146
sYmbol.ECA.BINary.CollapseAll	Collapse all trees	146
sYmbol.ECA.BINary.ControlFlowMode.INSTR	Consider instrumentation	146
sYmbol.ECA.BINary.ControlFlowMode.Trace	Consider trace events	147
sYmbol.ECA.BINary.EditDecision	Modify start address of decision	147

sYmbol.ECA.BINary.ExpandAll	Expand all trees	148
sYmbol.ECA.BINary.EXPORT.AdJoinGAPS	Split up observability gaps	149
sYmbol.ECA.BINary.EXPORT.Decisions	Export decision details as CSV	149
sYmbol.ECA.BINary.EXPORT.GAPS	Export observability gaps to JSON	149
sYmbol.ECA.BINary.FilterMapped	Filter display by the mapping state	150
sYmbol.ECA.BINary.FilterType	Filter display by decision type	151
sYmbol.ECA.BINary.PROCESS	Static preprocessing for code coverage	151
sYmbol.ECA.BINary.SetCONDitionOffset	Set condition offset	153
sYmbol.ECA.BINary.SetDecisionState	Disable/Enable decision evaluation	154
sYmbol.ECA.BINary.view	Result of static preprocessing for code coverage	155
sYmbol.ECA.Delete	Delete loaded ECA data	157
sYmbol.ECA.Init	Clear gathered ECA data	157
sYmbol.ECA.List	List ECA file overview	157
sYmbol.ECA.LOAD	Load a single ECA file	160
sYmbol.ECA.LOADALL	Load all ECA files	161
sYmbol.FILTER.ADD.SOURCE	Add source files to filter	162
sYmbol.FILTER.ADD.sYmbol	Add symbols to filter	162
sYmbol.FILTER.Delete	Delete filter	163
sYmbol.ForEach	Symbol wildcard command	164
sYmbol.INFO	Display detailed information about debug symbol	165
sYmbol.LANGUAGE	Select language	168
sYmbol.List	Display list of all symbols	169
sYmbol.List.ATTRibute	Display memory attributes	169
sYmbol.List.BUILTIN	List built-in data types	169
sYmbol.List.ColorDef	List the keyword color definitions	170
sYmbol.List.Enum	List of enumeration constants	171
sYmbol.List.FRAME	Display frames	172
sYmbol.List.Function	Display functions	173
sYmbol.List.IMPORT	List imported symbols	173
sYmbol.List.InlineBlock	List inlined code blocks	174
sYmbol.List.InlineFunction	List inlined functions	174
sYmbol.List.LINE	Display source lines	175
sYmbol.List.Local	Display local symbols	176
sYmbol.List.MACRO	List all C macros	176
sYmbol.List.MAP	Display memory load map	177
sYmbol.List.Module	Display modules	177
sYmbol.List.PATCH	Display STF-symbol information	178
sYmbol.List.Program	Display programs	178
sYmbol.List.REFERENCE	Display reference information	179
sYmbol.List.SECTION	Display physical sections	180
sYmbol.List.SOURCE	Display source file names	181
sYmbol.List.SourceFunction	Display source to function relations	183
sYmbol.List.SOURCETREE	Display source files hierarchy	184

sYmbol.List.STACK	Display virtual stack	184
sYmbol.List.Static	Display static symbols	185
sYmbol.List.TREE	Display symbols in tree form	185
sYmbol.List.Type	Display data types	186
sYmbol.LSTLOAD	Load assembler source file	187
sYmbol.LSTLOAD.GHILLS	Load GHILLS assembler source file	187
sYmbol.LSTLOAD.HPASM	Load HP assembler source file	187
sYmbol.LSTLOAD.IAR	Load IAR assembler source file	189
sYmbol.LSTLOAD.INT68K	Load Intermetrics assembler source file	190
sYmbol.LSTLOAD.INTEL	Load INTEL assembler source file	190
sYmbol.LSTLOAD.INTEL2	Load INTEL assembler source file	191
sYmbol.LSTLOAD.KEIL	Load Keil assembler source file	191
sYmbol.LSTLOAD.MicroWare	Load MICROWARE assembler source file	191
sYmbol.LSTLOAD.MRI68K	Load MICROTEC assembler source file	193
sYmbol.LSTLOAD.OAK	Load OAK assembler source file	193
sYmbol.MARKER	Fine-tune the nested function run-time analysis	194
sYmbol.MARKER.Create KBEGIN/KEND Marker	Marker for nesting function run-time analysis	195
sYmbol.MARKER.Delete	Delete a marker	199
sYmbol.MARKER.List	Displays the marker list	199
sYmbol.MARKER.RESet	Erase all markers	199
sYmbol.MARKER.TOUCH	Marker post-processing	200
sYmbol.MATCH	Symbol search mode	200
sYmbol.MEMory	Display memory usage	201
sYmbol.Modify	Modify symbols	202
sYmbol.Modify.Access	Modify access of symbols	202
sYmbol.Modify.ADDress	Modify address of symbols	203
sYmbol.Modify.AddressToRange	Modify address of symbols	203
sYmbol.Modify.AlienFunction	Disable frame info for a function	204
sYmbol.Modify.ATTRibute	Modify memory attribute	204
sYmbol.Modify.CutFunction	Reduce function address information	204
sYmbol.Modify.NAME	Rename symbol	205
sYmbol.Modify.NAMES	Rename symbols	205
sYmbol.Modify.RangeToAddress	Modify address of symbols	206
sYmbol.Modify.RangeToFunction	Modify address range into function	206
sYmbol.Modify.SOURCE	Define source file	206
sYmbol.Modify.SplitFunction	Split function	207
sYmbol.Modify.StaticCOPY	Create static copy of local stack variables	207
sYmbol.Modify.StaticToStack	Change static variables	208
sYmbol.Modify.TYPE	Modify type of symbols	208
sYmbol.name	Display symbols	209
sYmbol.NAMESPACES	Search symbol in C++ namespace	211
sYmbol.NEW	Create new symbol	212

sYmbol.NEW.Attribute	Create user-defined memory attribute	212
sYmbol.NEW.Function	Create user-defined function	214
sYmbol.NEW.Label	Create user-defined symbol	215
sYmbol.NEW.LocalVar	Create user-defined local variable	216
sYmbol.NEW.MACRO	Create user-defined macro	216
sYmbol.NEW.Module	Create user-defined module	216
sYmbol.NEW.Var	Create user-defined variable	217
sYmbol.OVERLAY	Code overlay	218
sYmbol.OVERLAY.AutoID	Automatically determine overlay IDs	218
sYmbol.OVERLAY.Create	Declare code overlay section	220
sYmbol.OVERLAY.DETECT	Detect the current overlay status	224
sYmbol.OVERLAY.FRIEND	Declare a friend overlay segment	224
sYmbol.OVERLAY.List	Show declared code overlay sections	226
sYmbol.OVERLAY.RESet	Reset overlay declarations	226
sYmbol.PATCH	STF-symbol information	227
sYmbol.PATCH.DISable	Disable instrumentation code	227
sYmbol.PATCH.ENABLE	Enable instrumentation code	227
sYmbol.PATCH.List	Display STF-symbol information	228
sYmbol.POINTER	Define special register	230
sYmbol.POSTFIX	Set symbol postfix	230
sYmbol.PREFIX	Set symbol prefix	230
sYmbol.RELOCate	Relocate symbols	231
sYmbol.RELOCate.Auto	Control automatic relocation	231
sYmbol.RELOCate.Base	Define base address	232
sYmbol.RELOCate.List	List relocation info	232
sYmbol.RELOCate.Magic	Define program magic number	232
sYmbol.RELOCate.Passive	Define passive base address	233
sYmbol.RELOCate.shift	Relocate symbols	233
sYmbol.RESet	Clear symbol table	234
sYmbol.SourceBeautify	Beautify HLL lines on loading	235
sYmbol.SourceCONVert	Conversion for Japanese font	236
sYmbol.SourceLOAD	Initiate the loading of an HLL source file	237
sYmbol.SourcePATH	Source search path	238
sYmbol.SourcePATH.Delete	Delete path from search list	238
sYmbol.SourcePATH.DOWN	Make directory last in search order	239
sYmbol.SourcePATH.List	List source search paths	239
sYmbol.SourcePATH.RESet	Reset search path configuration	241
sYmbol.SourcePATH.Set	Define search path	242
sYmbol.SourcePATH.SetBaseDir	Define directory as base for relative paths	243
sYmbol.SourcePATH.SetCache	Internal use only	244
sYmbol.SourcePATH.SetCachedDir	Cache direct search path directory	244
sYmbol.SourcePATH.SetCachedDirCache	Internal use only	245
sYmbol.SourcePATH.SetCachedDirIgnoreCache	Cache direct search path	245

sYmbol.SourcePATH.SetDir	Define directory as direct search path	246
sYmbol.SourcePATH.SetDynamicDir	Adjust search order at hit	247
sYmbol.SourcePATH.SetMasterDir	Store cached files only relative	248
sYmbol.SourcePATH.SetRecurseDir	Define recursive direct search path	249
sYmbol.SourcePATH.SetRecurseDirCache	Internal use only	249
sYmbol.SourcePATH.SetRecurseDirIgnoreCase	Recursive search path	250
sYmbol.SourcePATH.Translate	Replace part of the source path	250
sYmbol.SourcePATH.TranslateSUBpath	Replace sub-path	252
sYmbol.SourcePATH.UP	Move path up in the search order	252
sYmbol.SourcePATH.Verbose	Display search details in message AREA	253
sYmbol.SourceRELOAD	Reload source files	254
sYmbol.STATE	Display statistic	254
sYmbol.STRIP	Set max. symbol length	255
sYmbol.TYPEINFO	Display information about a specific data type	255
sYmbol.View	Show symbol info	256
SYnch		257
SYnch	Synchronization mechanisms between different TRACE32 systems	257
Overview SYnch		257
SYnch.Connect	Connect to other TRACE32 PowerView instances	258
SYnch.MasterBreak	Invite other TRACE32 to stop synchronously	260
SYnch.MasterGo	Invite other TRACE32 to start synchronously	261
SYnch.MasterStep	Invite other TRACE32 to Asm step synchronously	261
SYnch.MasterSystemMode	Invite other TRACE32 to follow mode change	262
SYnch.OFF	Disable connection mechanism	262
SYnch.ON	Enable connection mechanism	262
SYnch.RESet	Reset SYnch mechanism	263
SYnch.SlaveBreak	Synchronize with stop in connected TRACE32	263
SYnch.SlaveGo	Synchronize with start in connected TRACE32	264
SYnch.SlaveStep	Synchronize with asm step in connected TRACE32	264
SYnch.SlaveSystemMode	Synch. with mode changes in other TRACE32	265
SYnch.state	Display current SYnch settings	265
SYnch.XTrack	Establish time synchronization to another TRACE32 instance	266
SYStem		268
SYStem	System configuration	268
SYStem.BdmClock	Select BDM clock	268
SYStem.BREAKTIMEOUT	Define the used timeout for break	269
SYStem.CADICommand	Send a command to target	270
SYStem.CADIconfig	CADI-specific setups	271
SYStem.CADIconfig.ExecSwOnly	Filter on executing software capability	271
SYStem.CADIconfig.RemoteServer	Define connection to CADI server	271
SYStem.CADIconfig.SpecRegDefine	Define special register set	273
SYStem.CADIconfig.SpecRegsOnly	Use only special defined register set	273
SYStem.CADIconfig.Traceconfig	Define network settings to CADI trace	274

SYStem.CADIconfig.TraceCore	Define core for CADI trace	274
SYStem.CONFIG	Configure debugger according to target topology	275
SYStem.CONFIG.CORE	Assign core to TRACE32 instance	276
SYStem.CONFIG.CoreNumber	Set up number of hardware threads	282
SYStem.CONFIG.DEBUGPORT	Specify debugport	283
SYStem.CONFIG.DEBUGTIMESCALE	Extend debug driver timeouts	286
SYStem.CONFIG.ELA	Configure Embedded Logic Analyzer (ELA)	287
SYStem.CONFIG.ListCORE	Display the cores of a virtual target	287
SYStem.CONFIG.ListSIMulation	Display the simulations of a virtual target	288
SYStem.CONFIG.MULTITAP	Select type of JTAG multi-TAP network	289
SYStem.CONFIG.MULTITAP.JtagSEQUence	JTAG seq. on SYStem.Up	290
SYStem.CONFIG.state	Display target configuration	292
SYStem.CONFIG.TRACEPORT	Declare trace source and trace port type	293
SYStem.CONFIG.TRANSACTORPIPENAME	Set up pipe name	294
SYStem.CONFIG.USB	USB configuration	294
SYStem.CONFIG.XCP	XCP specific settings	294
SYStem.CPU	Select CPU	296
SYStem.CpuAccess	Run-time memory access (intrusive)	297
SYStem.CpuBreak	Master control to deny stopping the target (long stop)	298
SYStem.CpuSpot	Master control to deny spotting the target (short stop)	299
SYStem.DCI	DCI configuration	299
SYStem.DETECT	Detect target system resources	300
The System Detection Wizard		302
Daisy-Chain Detection via the TRACE32 AREA Window		304
SYStem.DLLCommand	Custom DLL connection to target	304
SYStem.InfineonDAS	Configure the InfineonDAS debug port	305
SYStem.IRISconfig	IRIS-specific setups	306
SYStem.IRISconfig.RemoteServer	Define connection to IRIS server	306
SYStem.JtagClock	Define JTAG frequency	307
SYStem.LOG	Log read and write accesses to the target	308
SYStem.LOG.CLEAR	Clear the 'SYStem.LOG.List' window	309
SYStem.LOG.CLOSE	Close the system log file	310
SYStem.LOG.Init	Clear the 'SYStem.LOG.List' window	310
SYStem.LOG.List	Log the accesses made by TRACE32	311
SYStem.LOG.Mode	Set logging mode	312
SYStem.LOG.OFF	Pause logging	313
SYStem.LOG.ON	Resume logging	313
SYStem.LOG.OPEN	Open a system log file	314
SYStem.LOG.RESet	Reset configuration of system log to defaults	314
SYStem.LOG.Set	Select the TRACE32 accesses to be logged	315
SYStem.LOG.SIZE	Define number of lines in the 'SYStem.LOG.List' window	316
SYStem.LOG.state	Open configuration window of system log	317
SYStem.LOG.StopOnError	Stop logging on error	318

SYStem.MCDCommand	Send command to MCD server	318
SYStem.MCDconfig	Send configuration to MCD server	319
SYStem.MemAccess	Select run-time memory access method	320
SYStem.Mode	Select mode	321
SYStem.Option	Special setup	321
SYStem.Option.IMASKASM	Disable interrupts while single stepping	322
SYStem.Option.IMASKHLL	Disable interrupts while HLL single stepping	322
SYStem.Option.MACHINESPACES	Address extension for guest OSes	322
SYStem.Option.MMUSPACES	Separate address spaces by space IDs	323
SYStem.Option.ZoneSPACES	Enable symbol management for zones	324
SYStem.PAUSE	Pause the execution of operations	325
SYStem.POLLING	Polling mode of CPU	326
SYStem.PORT	Configure external communication interface	327
SYStem.RESet	Reset configuration	328
SYStem.RESetOut	Reset peripherals	328
SYStem.RESetTarget	Release target reset	328
SYStem.state	Display SYStem.state window	329
SYStem.TARGET	Set target IP name or address	330
SYStem.VirtualTiming	Modify timing constraints	331
SYStem.VirtualTiming.HardwareTimeout	Disable/enable hardware timeout	332
SYStem.VirtualTiming.HardwareTimeoutScale	Multiply hardware timeout	332
SYStem.VirtualTiming.InternalClock	Base for artificial time calculation	333
SYStem.VirtualTiming.MaxPause	Limit pause	334
SYStem.VirtualTiming.MaxTimeout	Override time-outs	334
SYStem.VirtualTiming.OperationPause	Insert a pause after each operation	335
SYStem.VirtualTiming.PauseinTargetTime	Set up pause time-base	335
SYStem.VirtualTiming.PauseScale	Multiply pause with a factor	336
SYStem.VirtualTiming.PollingPause	Advance emulation time when polling	336
SYStem.VirtualTiming.TimeinTargetTime	Set up general time-base	337
SYStem.VirtualTiming.TimeScale	Multiply time-base with a factor	338
SystemTrace		339
SystemTrace	MIPI STP and CoreSight ITM	339
SystemTrace.state	Open system-trace configuration window	341

History

- 26-Mar-2024 New command [sYmbol.ECA.BINary.EXPORT.AdJoinGAPS](#).
- 19-Feb-2024 Description for [sYmbol.AutoLOAD.List](#) updated.
- 24-Oct-2023 Description for [SYStem.CONFIG.DEBUGPORT](#) updated.
- 16-Oct-2023 Description for [sYmbol.ECA.BINary.PROCESS](#) updated.
- 16-Oct-2023 New command [sYmbol.ECA.BINary.ControlFlowMode.INSTR](#),
[sYmbol.ECA.BINary.ControlFlowMode.Trace](#) and [sYmbol.ECA.BINary.EXPORT.GAPS](#).
- 23-Aug-2023 New command [sYmbol.List.Enum](#).
- 19-May-2023 Updated the window [sYmbol.ECA.List](#) by introducing a new checkbox **LENient** and
replacing the column **Time** with **signature**.
- 26-Apr-2023 Renamed [sYmbol.ECA.BINary.EXPORT](#) to [sYmbol.ECA.BINary.EXPORT.Decisions](#).
- 12-Aug-2022 New option **/WinTOP** for the command [STOre](#).
- 28-Jul-2022 Updated description of [sYmbol.MATCH Best](#).
- 13-Apr-2022 New command [sYmbol.DEOBFUSCATE](#).
- 24-Jan-2022 New command [sYmbol.ECA.BINary.PROCESS](#).
- Dec-2021 Added description for the command [sYmbol.ECA.Init](#).
- Dec-2021 Added description for the command [sYmbol.CLEANUP.AlignmentPaddings](#).

SELFTEST

SELFTEST

Execute selftest operation

Format:	SELFTEST
---------	-----------------

Executes the **SELFTEST** operation. Error results are shown in the selected **AREA** window.

Using the **SETUP** command group, many parameters of the debugger or window system can be changed.

For additional **SETUP** commands, refer to the **SETUP** commands in “[PowerView Command Reference](#)” (ide_ref.pdf).

See also

- | | |
|--|---|
| ■ SETUP.ALIST | ■ SETUP.BreakPointTableWalk |
| ■ SETUP.BreakTransfer | ■ SETUP.COLORCORE |
| ■ SETUP.DIS | ■ SETUP.DUMP |
| ■ SETUP.EMUPATH | ■ SETUP.GoOnPaused |
| ■ SETUP.IMASKASM | ■ SETUP.IMASKHLL |
| ■ SETUP.LISTCLICK | ■ SETUP.PROCESS |
| ■ SETUP.SIMULINK | ■ SETUP.StepAllCores |
| ■ SETUP.StepAtBreakPoint | ■ SETUP.StepAutoAsm |
| ■ SETUP.StepBeforeGo | ■ SETUP.StepByStep |
| ■ SETUP.StepNoBreak | ■ SETUP.StepOnPaused |
| ■ SETUP.StepTrace | ■ SETUP.StepWithinBreakpoints |
| ■ SETUP.StepWithinTask | ■ SETUP.sYmbol |
| ■ SETUP.TIMEOUT | ■ SETUP.Var |
| ■ SETUP.VarCall | ■ SETUP.VarPtr |
| ■ SETUP.VerifyBreakSet | |
| ▲ ‘ SETUP ’ in ‘ PowerView Command Reference ’ | |

The **SETUP.ALIST** commands allow to set up the default display of the [Analyzer.List](#) command.

See also

■ [SETUP](#)

SETUP.ALIST.RESet

Reset analyzer display

Format: **SETUP.ALIST.RESet**

Resets analyzer display to the default settings.

SETUP.ALIST.set

Default analyzer display

Format: **SETUP.ALIST.set** *<items>* ... [/BT | /FT]

The syntax of the command is the same as the channel selection for the command [Analyzer.List](#). The option defines if the Flowtrace or the Bustrace analyzer should be used as default by **all** analyzer display commands.

Example:

```
SETUP.ALIST Address CPU Time.REF           ; display external trace, cpu and
                                           time
```

SETUP.BreakPointTableWalk

Set up MMU translation for breakpoints

Format: **SETUP.BreakPointTableWalk** [ON | OFF]

When set to ON, this command enables MMU translation for breakpoint tag delete.

See also

■ [SETUP](#)

Format: **SETUP.BreakTransfer** [ON | OFF] (deprecated)

This command is deprecated because the TRACE32 TCF integration provides a synchronization between TRACE32 PowerView and Eclipse. For example, setting a breakpoint or executing a single step at the TRACE32 side will be reported to Eclipse and vice versa.

For more information, see [“TRACE32 as TCF Agent”](#) (app_tcf_setup.pdf).

See also

- [SETUP](#)

SETUP.COLORCORE

Enable coloring for core-specific info in SMP systems

Format: **SETUP.COLORCORE** [ON | OFF]

- | | |
|------------------------|--|
| ON
(default) | Core-specific information is displayed against a colored window background (SMP debugging and tracing only). |
| OFF | Coloring of core-specific information is disabled. |

See also

- [SETUP](#)
- ▲ [‘PowerView - Screen Display’](#) in [‘PowerView User’s Guide’](#)

Format:	SETUP.DIS [<fields> [<bar>]] [<constants>]
<fields>:	[<code>] [<label>] [<mnemonic>] [<comment>]
<bar>:	[<head>] [<bottom>]
<constants>:	[Hex Decimal] [Signed Unsigned] [Absolut sYmbol]

The command sets **default values** for configuring the disassembler output of **newly created windows** (e.g. **Data.List**). The command does **not affect existing windows** containing disassembler output.

Among other things the size of columns and the format of for constants (signed, unsigned, ...) can be configured.

The first four parameters for this command configure the size of the columns in disassembler output:

<code>	Number of displayed code bytes. Set to zero is possible.
<label>	Size of the label field.
<mnemonic>	Size of the mnemonic field.
<comment>	Size of the comment field.

The next two arguments limit the movement of the PC bar within the window:

<head>	Size of reserved area on the top of the window (in percent).
<bottom>	Size of reserved area in bottom of window.

With these arguments, you can configure the display of constants and symbols:

Hex	In the mnemonic field the constants are displayed in hex.
Decimal	In the mnemonic field the constants are displayed in decimal.
Signed	The constants are displayed as signed numbers.
Unsigned	The constants are displayed as unsigned numbers.

Absolut

In the mnemonic field the constants are displayed absolute, with the comment field they are displayed symbolically.

sYmbol

The constants are displayed symbolically within the mnemonic field.

There might be more architecture specific keywords. E.g.: For the 68K family there is an additional parameter <traparg> to specify the number of bytes to be used as argument after trap commands (for OS-9). For PowerPC you can use **Simple** (default) or **Generic** to chose between simple or generic mnemonics. Please see the related [Processor Architecture Manual](#) for architecture specific keywords.

Example:

```
SETUP.DIS Unsigned           ; display constants as unsigned values
SETUP.DIS sYmbol            ; Switch to symbolic display of constants
```

See also

■ [SETUP](#)

■ [List](#)

■ [List.auto](#)

SETUP.DUMP

Defaults for hex-dumps

Format: **SETUP.DUMP** [/<option> ...]

<option>

For a description of the options, see [Data.dump](#).

Example:

```
SETUP.DUMP /Byte           ; display width is now byte by default
```

See also

■ [SETUP](#)

■ [Data.dump](#)

■ [Var.DUMP](#)

Format: **SETUP.EMUPATH** "<command>" ...

The most left softkey selects emulation softkeys. These softkeys will be defined by this command.

```
SETUP.EMUPATH "s." "g.n" "r." "fpu." "d.s 0x0ff000 0x0" "r.res"
```

gives the following softkeys:



See also

- [SETUP](#)

SETUP.GoOnPaused

Route go to paused core

Virtual targets only: MCD

Format: **SETUP.GoOnPaused** [ON | OFF]

Default: OFF.

- ON

Go command will be sent to a core even if it is in “paused” state.
- OFF

If selected core is “paused” state, Go command will be sent to the next core which is in “stopped” state.

See also

- [SETUP](#)

Format:

SETUP.IMASKASM [ON | OFF]

If enabled, the interrupt enable bit of the microcontroller will be disabled during single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt enable bit is restored to the value before the step. This command is not implemented within all emulation probes.

NOTE:

On some processors this modification is also seen by the user program. So this option can affect the flow of the target program. Accesses to the interrupt-enable bit can see the wrong values. Operations to modify the interrupt enable bit may not work as expected.

See also

- [SETUP](#)
- [Step.single](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format:

SETUP.IMASKHLL [ON | OFF]

If enabled, the interrupt enable bit of the microcontroller will be disabled during HLL single-step operation. The interrupt routine is not executed during single-step operations. After single step the interrupt enable bit is restored to the value before the step. This command is not implemented within all emulation probes.

NOTE:

By changing the register through target software, this option can affect the flow of the target program. Accesses to the interrupt-enable bit will see the wrong values. Operations to modify the interrupt enable bit will not work as expected. When the HLL line enables the interrupts (e.g. in an RTOS function call) then pending interrupts will be executed.

See also

- [SETUP](#)
- [Step.single](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format: **SETUP.LISTCLICK "<command>"**

Executes the defined *<command>* when you double-click a variable or function in HLL-source (with the left mouse key). The name of the variable/function is appended to the command. The characters '?' or '*' can be used to mark the position of the variable/function name in the command. Lines with '*' will be executed without further input query.

Example 1: Default action when you double-click a function in a TRACE32 window, e.g. in the [List.Mix](#) window:

```
; display a listing for the double-clicked function in a new List  
; window that is superimposed on the previous List window  
SETUP.LISTCLICK "WinOverlay.List `*`"
```

Example 2: Default action when you double-click a variable in a TRACE32 window, e.g. in the [List.Mix](#) window:

```
SETUP.LISTCLICK "Var ?"           ; lets you modify the variable in  
                                   the  
                                   ; TRACE32 command line
```

Example 3: A user-defined action:

```
SETUP.LISTCLICK "Var.View %Multiline *" ; open a temporary window with  
                                           variable
```

See also

[■ SETUP](#)[■ List](#)[■ List.auto](#)[■ WinOverlay](#)

Format:

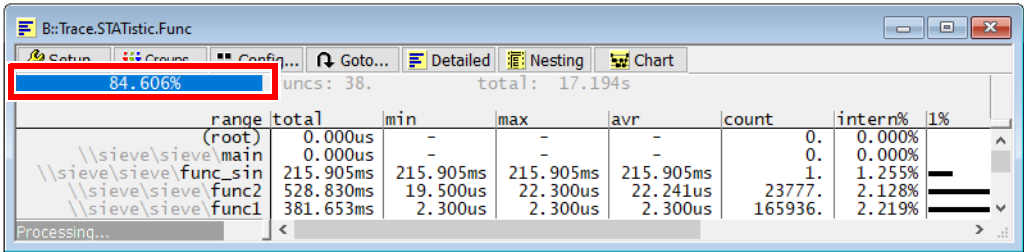
SETUP.PROCESS [ON | OFF]

Default: OFF.

- OFF

While trace is processing, the string “PROCESING” is displayed on the top left corner of `<trace>.STATistic` windows.
- ON

While trace is processing, the processing percentage is displayed on the top left corner of `<trace>.STATistic` windows.



See also

- [SETUP](#)

SETUP.SIMULINK

Deprecated command

Format:

SETUP.SIMULINK ON [*<option>*] | OFF (deprecated)

<option>:

<release> | **IntegrationDir** *<init_dir>* | **ToModelAlways** | *<debug>*

<release>:

R2010A | R2010B | R2011A | R2011B | R2012A | R2012B | R2010A | R2013A | R2013B | R2014A | R2014B

<debug>:

Verbose | NoExchange

This command is no longer needed in the new integration. For information about the new TRACE32 Simulink integration, refer to [“Integration for Simulink”](#) (int_simulink.pdf).

See also

- [SETUP](#)

Format:	SETUP.StepAllCores [ON OFF]
---------	--------------------------------------

Default: OFF.

Forces assembler single stepping on all cores of an SMP system.

If you debug a multicore system in SMP configuration a single step on HLL code affects all cores while single stepping on ASM code does affect only the active core.
By switching **SETUP.StepAllCores** to **ON** also single steps on assembler level will affect all cores.

Lauterbach recommends to keep **SETUP.StepAllCores OFF**

Support of this feature depends on your CPU.
Setting **SETUP.StepAllCores** to **ON** might have no effect.
The setting is supported for MPC5xxx PowerPCs. It is not yet supported for ARM or TriCore.
Please contact Lauterbach, if you need this feature for your target architecture.

By just typing the command and appending a blank, you can view the current setting in the [TRACE32 message line](#).
By executing the command without arguments, **SETUP.StepAllCores** toggles the current setting.

See also

- [SETUP](#)

Format:

SETUP.StepAtBreakPoint [ON | OFF | DEFault]
SETUP.StepBreak [ON | OFF] (deprecated)

When interrupts are pending and the emulation is started on a breakpoint, it is possible that the target executes the interrupt routine and returns to the same breakpoint location after. The debugging 'hangs' on the breakpoints. To avoid this, this option will first execute a single step when the program would start on a breakpoint. On some processors with internal interrupt sources, the [SETUP.IMASKASM](#) option must also be turned **ON**. This option is usually the default for ICD.

DEFault	Selects the architecture's default behavior.
OFF	Performs an ASM single step before continuing program execution after a Go .
ON	Immediately continues program execution after a Go .

See also

- [SETUP](#)

Format:

SETUP.AutoAsm [ON | OFF]

When a single step is performed in HLL debug mode and the target address of the step is code without HLL information (e.g. a module compiled without HLL debug symbols), the debugger will per default continue single stepping in the background until the next HLL line is reached (i.e. step from HLL line to HLL line). If the setting is turned **ON**, the debugger will stop at the address without debug symbols. Use this setting to debug modules without HLL debug information or compiler generated code sections.

See also

- [SETUP](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format: **SETUP.StepBeforeGo** [ON | OFF]

Perform an ASM single step before each **Go**. In contrast to the **SETUP.StepAtBreakPoint** command, this option steps always regardless the emulation is started on a breakpoint or not.

- OFF**
- Performs an ASM single step before continuing program execution after a **Go**.
- ON**
- Immediately continues program execution after a **Go**.

See also

■ [SETUP](#)

SETUP.StepByStep

Single step HLL lines

Format: **SETUP.StepByStep** [ON | OFF]

Single steps HLL when executing an HLL step. On some processors with internal interrupt sources, the **SETUP.IMASKASM** option must also be turned on to avoid stepping through the interrupt program.

See also

■ [SETUP](#)

SETUP.StepNoBreak

Stepping HLL lines with disabled breakpoints

Format: **SETUP.StepNoBreak** [ON | OFF]

- OFF**
- User-defined breakpoints are active while single stepping in HLL.
- ON**
- User-defined breakpoints are not active while single stepping in HLL.

See also

■ [SETUP](#)

Virtual targets only: MCD

Format:

SETUP.StepOnPaused [ON | OFF]

Default: OFF.

- ON

The MCD step command is also sent to a core in “paused” state.
- OFF

The MCD step command is sent to the next core which is in “stopped” state and not in “paused stated.

See also

■ [SETUP](#)

SETUP.StepTrace

Show stepping trail in list window

Format:

SETUP.StepTrace [ON | OFF]

If this option is enabled, list windows will show stepping trails.

See also

■ [SETUP](#)

SETUP.StepWithinBreakpoints

Multi-core step on SMP systems

Format:

SETUP.StepWithinBreakpoints [ON | OFF]

Enables/disables multi-core step on SMP systems.

See also

■ [SETUP](#)

Format:

SETUP.StepWithinTask [ON | OFF]

When enabled all HLL stepping and temporary breakpoints will be task selective (on the currently active task). This allows to step and debug shared code without stopping in another task.

See also

■ [SETUP](#)

SETUP.sYmbol

Length of symbols

Format:

SETUP.sYmbol <path_len> <name_len> <type_len> [ON | OFF]

Configures the width of the columns for the symbol display commands. The **SETUP.sYmbol** command only affects the display of symbols, not the number of significant characters during symbol entry.

<path_len>	Sets the default display width for columns which hold a complete symbol path, including program and module names. An example is the width of the path\symbol column in the sYmbol.List window.
<name_len>	Sets the default display width for single symbol names. An example is the width of the symbol column in the sYmbol.Browse window.
<type_len>	Sets the default display width for columns holding information on the symbol type. An example is the width of the type column in the sYmbol.Browse window.
ON, OFF	Displays or hides the program name in symbol paths. An example is the path in the path\symbol column of the sYmbol.List window.

See also

■ [SETUP](#)

■ [List](#)

■ [List.auto](#)

■ [sYmbol.Browse](#)

Format:

SETUP.TIMEOUT <factor>

Values larger than 1 stretch the time-out delay within the emulation monitor. This value determines, how long a window waits for becoming inactive. Short values will result in a fast screen update, but may result in flickering windows when a spot point or the OS Awareness is active. Large values will cause a slower update on the screen when real-time emulation is running.

See also

- [SETUP](#)
- [Data.dump](#)
- [Data.Test](#)
- [List](#)
- [List.auto](#)

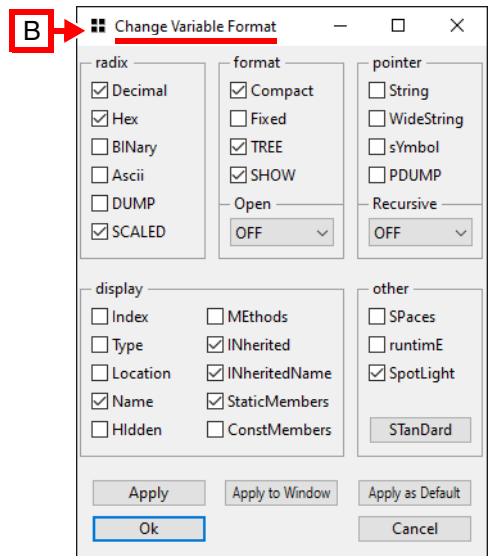
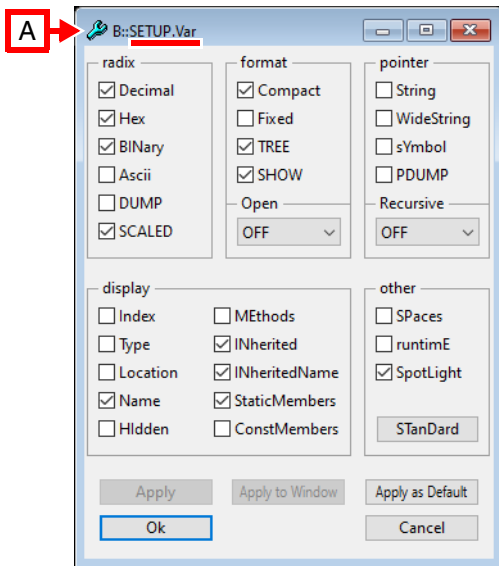
Format:	SETUP.Var [%<format> ...]
---------	----------------------------------

Defines the default formatting of variables in **Var** windows.

- Without <format> parameters, the **SETUP.Var** command opens the **SETUP.Var** dialog window. (Same as choosing **Var** menu > **Format** from the TRACE32 menu bar.)
- With one or more parameters: The settings can be changed via the TRACE32 command line or a PRACTICE script (*.cmm) while the dialog window remains closed.

Your settings are used during a TRACE32 session, or until you change the settings again during the same session.

<format>	For a description of the <formats>, see section “ Display Formats ” of the Var command group.
----------	---



A Global format settings.

Use the command **SETUP.Var**, or choose **Var** menu > **Format** to open the **SETUP.Var** dialog window.

In this dialog window, you can make format settings that apply to all **Var.*** windows you open afterwards. Your changes have no effect on **Var.*** windows that are already open.

For a description of the buttons in the **SETUP.Var** dialog window, see [table \[A\] below](#).

B Local format settings.

Right-click any variable in a **Var.*** window, and then select **Format** from the popup menu to open the **Change Variable Format** dialog window.

Use this dialog window to format just a particular variable or all variables in a particular **Var.*** window.

For a description of the buttons in the **Change Variable Format** dialog window, see [table \[B\] below](#).

[A] - Description of Buttons in the SETUP.Var Window

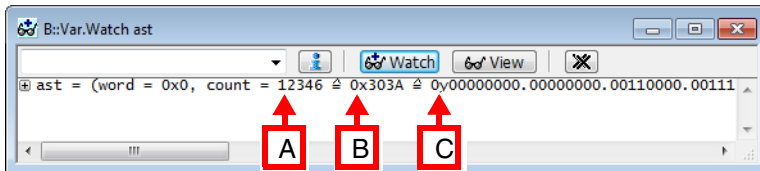
STanDard	Selects only the check boxes that belong to the built-in standard settings. All other check boxes are cleared.
Apply as Default	Applies your settings without closing the dialog window.
OK	Applies your settings and closes the dialog window.

STanDard	Selects only the check boxes that belong to the built-in standard settings. All other check boxes are cleared.
Apply	Applies the settings only to a particular variable you have selected in a particular Var.* window. The formatting of the other variables remains unchanged.
Apply to Window	Applies the settings to all variables displayed in a particular Var.* window. To apply new format settings to a particular Var.* window only: 1. Right-click any variable in the desired Var.* window, and then select Format from the popup menu. The Change Variable Format dialog window opens. 2. Make your new settings. 3. Click Apply to Window. The new settings are applied as local format settings to this particular Var.* window only. NOTE: All the other Var.* windows continue to use the global format settings as configured in the SETUP.Var dialog window.
Apply as Default	The local format settings of a particular Var.* window become the new global format settings. You can view the new configuration in the SETUP.Var dialog window.
OK	Applies your settings and closes the dialog window.

Examples

Example 1: Three `<format>` settings are switched on while the **SETUP.Var** dialog window remains closed. Then the **Var.Watch** window is opened, displaying the variable `ast` in the new format:

```
SETUP.Var %Decimal.on %Hex.on %BINary.on      ; see above screenshot [A]  
  
Var.Watch flags ast
```



A Decimal.on **B** Hex.on **C** BINary.on

Example 2: The built-in `<format>` standard is restored.

```
SETUP.Var %STanDard                          ; restores the built-in  
                                              ; standard
```

See also

■ [SETUP](#)

■ [Var](#)

▲ ['Var' in 'General Commands Reference Guide V'](#)

▲ ['Format Variable' in 'Training Source Level Debugging'](#)

Format: **SETUP.VarCall** [<address>]

If a function is called from the **Var** commands, a dummy routine is placed in memory to catch the processor after the called function has terminated. Under normal circumstances this code is never reached, as the HLL debugger breaks, when the end of the function is reached. If the command **Var.Call** is used, a **Go** command may start the function without any breakpoints set to the return point. In this cases, the processor will loop endless in the 'dummy' routine. Processors with linear addressing usually require no fixed address, the routine is kept on the stack. Processors with special addressing, like 8051 cannot keep a function on the stack. For this processors the command **SETUP.VarCall** can define a free location in code memory to hold the endless loop of the dummy function. The required space is usually two bytes.

Example:

```
SETUP.VarCall P:0x00fff0      ; place the dummy routine at P:0x00fff0
Var.NEWLOCAL \x              ; create virtual variable for result
Var.Call \x=func5(4,8,17)    ; call a function of the target program
```

See also

■ [SETUP](#)

Format:

SETUP.VarPtr [<address_range>]

Defines the address ranges for valid memory pointers. This range is checked whenever an automatic access to the contents of a pointer is made. Pointer referenced by an HLL expression are not checked against this range.

```
SETUP.VarPtr 0x0--0x0ffff
Var vpchar = 0x123456                ; set pointer to character to
                                     ; 123456

Var *vpchar                          ; manual access to pointer, not
                                     ; checked

displays: *vpchar = 0

Var %Recursive vpchar                ; automatic pointer access is
displays: vpchar = 0x123456 ->        ; checked
INVALID

SETUP.VarPtr 0x0--0x0ffffff          ; enlarge the address space for
                                     ; pointers

Var %Recursive vpchar                ; automatic pointer access is
displays: vpchar = 0x123456 -> 0x0    ; checked

SETUP.VarPtr 0x0--0x1ffff|0x800000--0x80ffff
```

See also

- [SETUP](#)

SETUP.VerifyBreakSet

Additional verification for software breakpoints

Format:

SETUP.VerifyBreakSet [ON | OFF]

Default: OFF

Setting **SETUP.VerifyBreakSet** to **ON** forces the debugger to perform an additional verification whenever a software breakpoint become active or inactive.

See also

- [SETUP](#)

The **SIM** command group covers the following features for the TRACE32 Instruction Set Simulators:

- Cache/MMU/MPU simulation: configuration, enabling and basic analysis
Cache simulation is currently only fully implemented for the ARM architecture. It can be implemented for other architectures on request.
Please be aware that enabling the cache/MMU simulation slows down the simulator performance.
- Trace generation: configuration
- Peripheral Simulation Models: load and unload
For more information on the PSM refer to “[API for TRACE32 Instruction Set Simulator](#)” (simulator_api.pdf) and “[Library for Peripheral Simulation](#)” (simulator_api_lib.pdf).

See also

- | | | | |
|----------------------------|-----------------------------|-------------------------------|---------------------------------|
| ■ SIM.AREA | ■ SIM.CACHE | ■ SIM.command | ■ SIM.INTerrupt |
| ■ SIM.List | ■ SIM.LOAD | ■ SIM.RESet | ■ SIM.UNLOAD |

SIM.AREA

Selects area for simulation output

Format: **SIM.AREA** <name>

Specify output **AREA** for API function `SIMUL_Printf(simulProcessor processor, const char *format, ...)`.

Example:

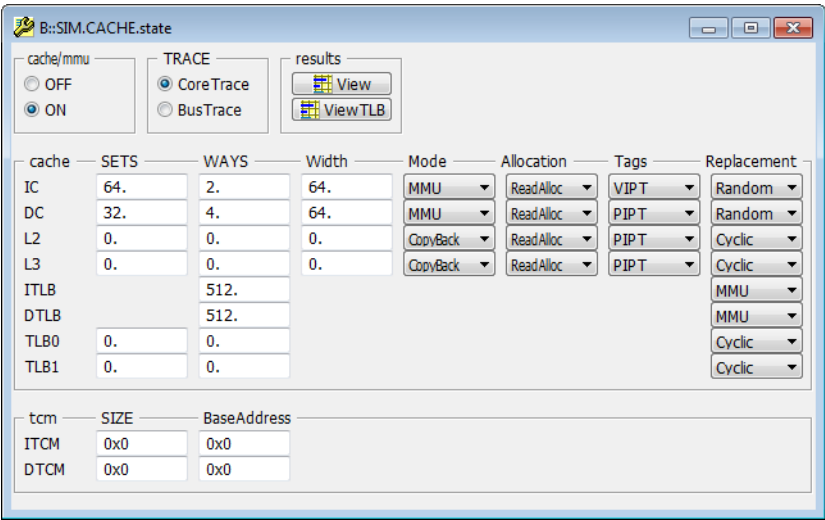
```
AREA.Create SimulOut           ; create a new AREA
AREA.view SimulOut             ; display created AREA
SIM.AREA SimulOut              ; assign AREA to SIMUL_Printf
                               ; function
```

See also

- | | |
|-----------------------|-------------------------------|
| ■ SIM | ■ SIM.command |
|-----------------------|-------------------------------|

Command group for cache/MMU simulation, simulation of tightly-coupled memory, simulator trace generation and more.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the [SIM.CACHE.state](#) window.



See also

- [■ SIM.CACHE.Allocation](#)
[■ SIM.CACHE.ON](#)
[■ SIM.CACHE.Tags](#)
[■ SIM.CACHE.WAYS](#)
- [■ SIM.CACHE.Mode](#)
[■ SIM.CACHE.Replacement](#)
[■ SIM.CACHE.TRACE](#)
[■ SIM.CACHE.Width](#)
- [■ SIM.CACHE.MPURegions](#)
[■ SIM.CACHE.SETS](#)
[■ SIM.CACHE.View](#)
[■ SIM](#)
- [■ SIM.CACHE.OFF](#)
[■ SIM.CACHE.state](#)
[■ SIM.CACHE.ViewTLB](#)
[■ SIM.command](#)
- [▲ 'Release Information' in 'Legacy Release History'](#)

Format:	SIM.CACHE.Allocation <i><cache_type></i> ReadAlloc WriteAlloc
<i><cache_type></i> :	DC L2 L3 ...

Describes how the CPU deals with a cache miss on a data store/write access.

In the **SIM.CACHE.state** window, the **Allocation** field shows the cache properties of the selected CPU. If these properties do not fit, they should be changed before a **SYStem.Up**.

ReadAlloc	The data from a memory address is only loaded to the cache on read/load accesses.
WriteAlloc	The data from a memory address is loaded to the cache on a store/write access and the new data is written in the cache line. If it is also stored/written to memory depends on the cache mode (write-through or copy-back).

The allocation technique is taken from the MMU if **SIM.CACHE.Mode** is set to MMU.

```
SIM.CACHE.Allocation IC ReadAlloc      ; the instruction cache is a
                                       ; read allocate cache
```

See also

- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

Format:	SIM.CACHE.Mode ITCM DTCM <i><mode></i>
<i><mode></i> :	CopyBack WriteThrough MMU

Defines the strategy used for the memory coherency. It is recommended to perform this setup before **SYStem.Up**.

CopyBack	Copy back strategy guarantees memory coherency. When a cache hit occurred for a data store/write, the cache contents is updated and the corresponding cache line is marked as dirty. The data value is copied back to memory when the contents of the cache line is evicted.
WriteThrough	Write Through strategy guarantees memory coherency. When a cache hit occurs for a data store/write, the cache contents is updated and the data is also stored/written to memory.
MMU	The strategy for memory coherency is taken from the MMU.

See also

■ [SIM.CACHE](#)

■ [SIM.CACHE.state](#)

SIM.CACHE.MPURegions

Specify MPU regions

Format:	SIM.CACHE.MPURegions <i><region></i>
---------	---

Defines the number of MPU regions implemented on your Cortex-R4 core.

See also

■ [SIM.CACHE](#)

■ [SIM.CACHE.state](#)

Format:	SIM.CACHE.OFF
---------	----------------------

Disables cache and MMU simulation.

See also

- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

Format:	SIM.CACHE.ON
---------	---------------------

Enables cache and MMU simulation.

See also

- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

Format:	SIM.CACHE.Replacement <i><cache> <replace></i>
<i><cache></i> :	ITLB DTLB TLB0 TLB1
<i><replace></i> :	NONE Random FreeRandom LRU MMU

Defines the replacement strategy for each cache.

In the **SIM.CACHE.state** window, the **Replacement** field shows the cache properties of the selected CPU. If these properties do not fit, they should be changed before a **SYStem.Up**.

Cyclic	Cyclic (round-robin) replacement strategy is used. One round robin counter for each cache set.
Random	Random replacement strategy is used.
LRU	Last recently used replacement strategy is used.
MMU	The replacement strategy is defined by the CPU. Please use SIM.CACHE.Replacement MMU if your CPU uses a not listed replacement strategy.

See also

- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

Format:	SIM.CACHE.SETS <i><cache></i> <i><number></i>
<i><cache></i> :	TLB0 TLB1

Defines the number of cache/TLB sets.

In the **SIM.CACHE.state** window, the **SETS** field shows the cache properties of the selected CPU. If these properties do not fit, they should be changed before a **SYStem.Up**.

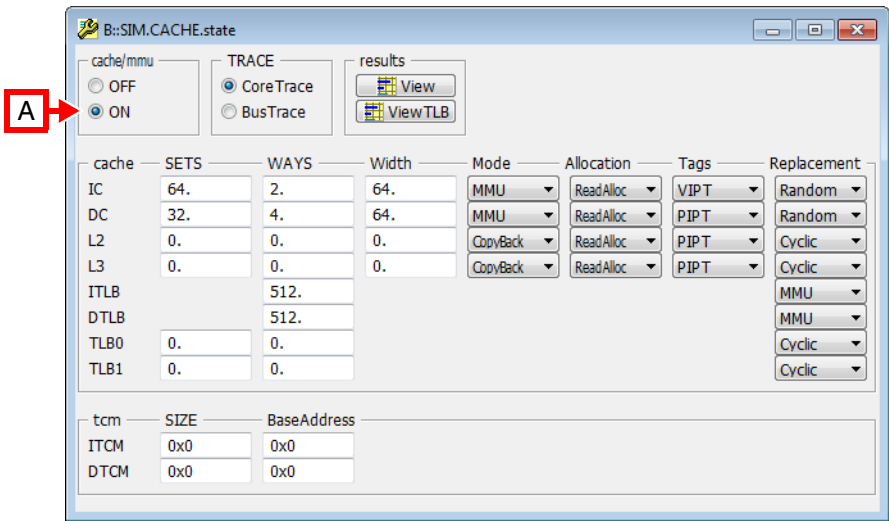
See also

- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

Format:

SIM.CACHE.state

Displays the simulator settings for cache and MMU.



A For descriptions of the commands in the **SIM.CACHE.state** window, please refer to the **SIM.CACHE.*** commands in this chapter.

Example: For information about **ON**, see [SIM.CACHE.ON](#).

See also

- [SIM.CACHE](#)
[SIM.CACHE.OFF](#)
[SIM.CACHE.Tags](#)
[SIM.CACHE.WAYS](#)
- [SIM.CACHE.Allocation](#)
[SIM.CACHE.ON](#)
[SIM.CACHE.TRACE](#)
[SIM.CACHE.Width](#)
- [SIM.CACHE.Mode](#)
[SIM.CACHE.Replacement](#)
[SIM.CACHE.View](#)
- [SIM.CACHE.MPURegions](#)
[SIM.CACHE.SETS](#)
[SIM.CACHE.ViewTLB](#)

Format:	SIM.CACHE.Tags <cache> <tag>
<tag>:	VIVT PIPT VIPT AVIVT

Defines the address mode for cache lines. The address mode for the cache lines is taken from the MMU if **SIM.CACHE.Mode** is set to MMU. It is recommended to perform this setup before **SYStem.Up**.

VIVT	Virtual Index, Virtual Tag The logical address is used as tag for a cache line.
PIPT	Physical Index, Physical Tag The physical address is used as tag for a cache line.
VIPT	Virtual Index, Physical Tag
AVIVT	Address Space ID + Virtual Index, Virtual Tag

See also

- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

SIM.CACHE.TRACE

Select simulator trace method

Format:	SIM.CACHE.TRACE BusTrace CoreTrace
---------	---

BusTrace	Trace information is generated for all bus transfers. E.g. if the cache is simulated trace information is generated for the burst cycles that filled the cache lines.
CoreTrace (default)	Trace information is generated for all executed instructions and performed load/store operations. Cache accesses are included.

See also

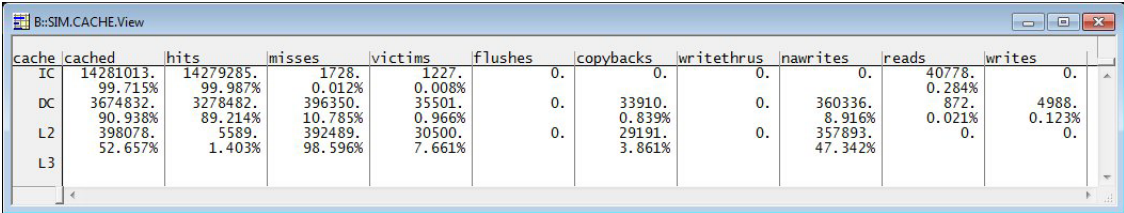
- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

Format:

SIM.CACHE.View

Displays an analysis of the simulated memory accesses if cache simulation is used. Analysis results can be displayed while program execution is running.

For detailed information on the interpretation of the results, refer to the [CTS.CACHE.View](#) command.



cache	cached	hits	misses	victims	flushes	copybacks	writethrus	nawrites	reads	writes
IC	14281013.	14279285.	1728.	1227.	0.	0.	0.	0.	40778.	0.
	99.715%	99.987%	0.012%	0.008%					0.284%	
DC	3674832.	3278482.	396350.	35501.	0.	33910.	0.	360336.	872.	4988.
	90.938%	89.214%	10.785%	0.966%		0.839%		8.916%	0.021%	0.123%
L2	398078.	5589.	392489.	30500.	0.	29191.	0.	357893.	0.	0.
	52.657%	1.403%	98.596%	7.661%		3.861%		47.342%		
L3										

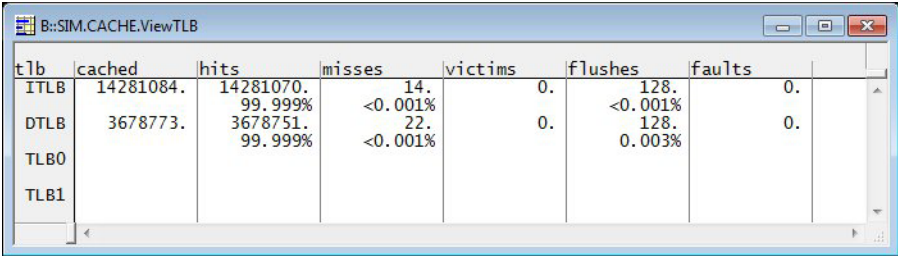
See also

- SIM.CACHE
- SIM.CACHE.state

Format:

SIM.CACHE.ViewTLB

Displays an analysis of the simulated TLB accesses if MMU simulation is used. Analysis results can be displayed while program execution is running.



tlb	cached	hits	misses	victims	flushes	faults
ITLB	14281084.	14281070.	14.	0.	128.	0.
		99.999%	<0.001%		<0.001%	
DTLB	3678773.	3678751.	22.	0.	128.	0.
		99.999%	<0.001%		0.003%	
TLB0						
TLB1						

See also

- SIM.CACHE
- SIM.CACHE.state

Format:	SIM.CACHE.WAYS <cache> <ways>
<cache>:	IC DC L2 L3 ITLB DTLB TLB0 TLB1

Defines the number of cache ways (blocks) for each cache.

In the [SIM.CACHE.state](#) window, the **WAYS** field shows the cache properties of the selected CPU. If these properties do not fit, they should be changed before a **SYStem.Up**.

Example:

```
SIM.CACHE.WAYS IC 4. ; The instruction CACHE has 4 blocks
```

See also

- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

Format:	SIM.CACHE.Width IC DC L2 L3 <width>
---------	---

Defines the width of a single cache line in bytes.

In the [SIM.CACHE.state](#) window, the **Width** field shows the cache properties of the selected CPU. If these properties do not fit, they should be changed before a **SYStem.Up**.

Example:

```
SIM.CACHE.Width IC 32. ; A cache line for the instruction cache  
; is 32. byte
```

See also

- [SIM.CACHE](#)
- [SIM.CACHE.state](#)

Format: **SIM.command** <cmd> [<string>] [<address>] [<time>] [<value>]

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

See also

- [SIM](#)

■ [SIM.AREA](#)

■ [SIM.CACHE](#)

■ [SIM.INTerrupt](#)
- [SIM.List](#)

■ [SIM.LOAD](#)

■ [SIM.RESet](#)

■ [SIM.UNLOAD](#)

SIM.INTerrupt

Trigger interrupt

Format: **SIM.INTerrupt** <level> <vector>

Triggers the specified interrupt.

Not all arguments are supported or required by all architectures.

Example for MPC55xx:

```
SIM.INTerrupt , 0x20           ; no priority required that is
                                ; why "," is used

                                ; interrupt vector 0x0 is triggered
```

Example for TriCore:

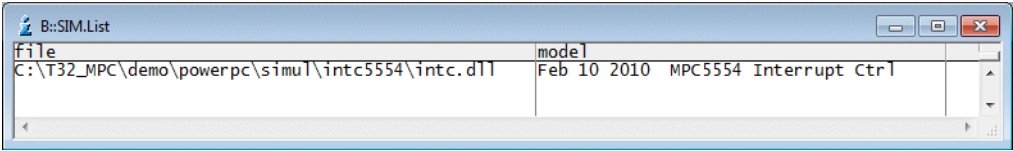
```
SIM.INTerrupt 15.              ; the interrupt is triggered by its
                                ; corresponding level

                                ; <vector> is not supported,
                                ; instead the vector is calculated
                                ; from the BIV register value
```

See also

- [SIM](#)
- [SIM.command](#)

Format:SIM.List



See also

- [SIM](#)
- [SIM.command](#)

SIM.LOAD

Load simulator module

Format:SIM.LOAD <file> [<parameter> ...]

Loads simulator DLL. The parameters are specific for the loaded DLL.

Example:

```
SIM.RESet ; reset simulator
SIM.LOAD demoport.dll 20000 0 ; loads DLL with your parameters
```

See also

- [SIM](#)
- [SIM.command](#)
- ['Release Information' in 'Legacy Release History'](#)

SIM.RESet

Reset TRACE32 Instruction Set Simulator

Format:SIM.RESet

Unloads all loaded DLL and resets all time base.

See also

- [SIM](#)
- [SIM.command](#)

Format:

SIM.UNLOAD [*<file>*]

Unloads a simulator DLL.

Example:

```
SIM.UNLOAD demoport.dll           ; unload specified DLL
SIM.UNLOAD                        ; unload all DLLs
```

See also

- [SIM](#)
- [SIM.command](#)

Format:

SLTrace.<trace_windows>

<trace_windows>:

List | Chart.Distrib | ProfileChart.DistriB | STATistic.DistriB

The **SLTrace** command group allows to trace and analyze the **SYStem.LOG** events, i.e. the read and write accesses TRACE32 performs to the target hardware.

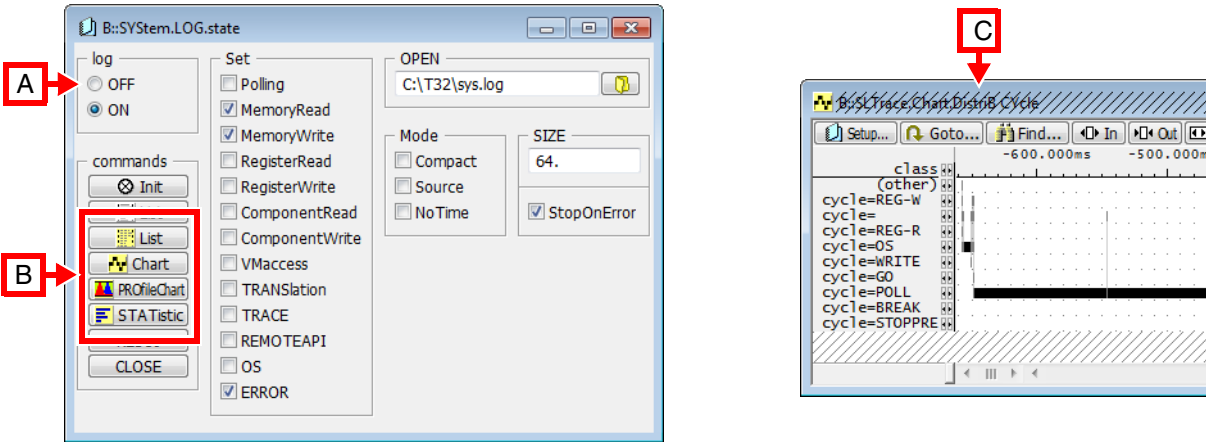
This is useful for analyzing critical timing of accesses done by the debugger. It may help to improve the speed of remote API calls.

<trace_windows>

You can view the system-log trace in chart, profile chart, trace listing or trace statistic windows. For your convenience, the <trace_windows> are directly accessible from the **SYStem.LOG.state** window, as shown below [B].

For descriptions of the subcommands, please refer to the general <trace> command descriptions in “**General Commands Reference Guide T**” (general_ref_t.pdf).

Example: For a description of **SLTrace.List** refer to <trace>.List



- A Set the **SYStem.LOG** to **OFF** so that the recorded system-log trace can be displayed in a <trace_window> or an existing <trace_window> can be refreshed with the latest the system-log trace.
- B To open a <trace_window>, click the button you want in the **SYStem.LOG.state** window [B].
- C Diagonal lines in a <trace_window> indicate that a system-log trace is being recorded and that the window has not yet been updated, i.e. **SYStem.LOG** is still **ON**.

See also

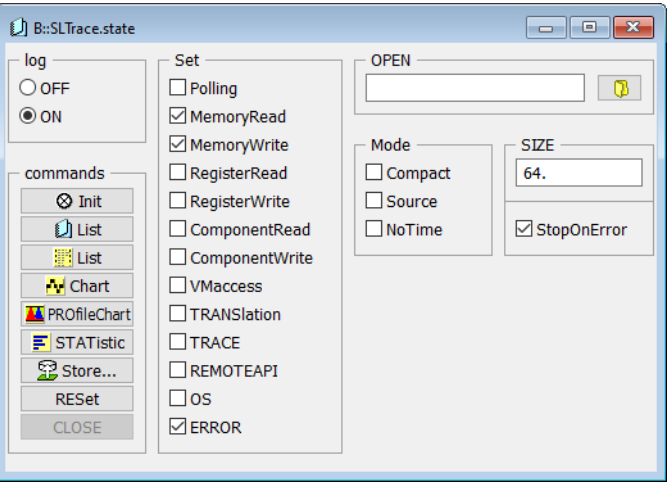
- [SYStem.LOG](#)
- [SYStem.LOG.state](#)

SLTrace.state

Display configuration window

Format: SLTrace.state

Displays the **SLTrace.state** window, where you can configure the **SLTrace**.



The **SNOOPer** trace is one of the TRACE32 trace methods which allows to gain runtime information with just a debugger. In order to get the runtime information the debugger periodically reads out information such as memory/variable contents, the program counter, or other system information while the program execution is running.

Ideally, the debugger can read this information non-intrusively. The readout period is in the microsecond range in this case. If this is not possible, the program execution has to be stopped periodically to read the desired information. The readout periods then tend to be in the millisecond range.

To achieve high SNOOPer frequencies, the sampling is performed by the software running on the TRACE32 debug hardware where the collected samples are times-stamped and stored to a temporary buffer. The buffer contents is read by TRACE32 PowerView after the recording stopped or is streamed to the host during recording if the temporary buffer within the debug hardware is smaller than the trace buffer size requested by the user.

If a TRACE32 software-only tool is used, the readout periods can be larger depending on the communication link in use. Since the sampling software runs on the host computer, it is more likely that the SNOOPer is suspended by other programs running there.

The “[Application Note for the SNOOPer Trace](#)” (app_snooper.pdf) introduces standard use cases and contains important information about the technical conditions of the SNOOPer trace.

The trace features of the **SNOOPer** can be configured and controlled with the command group **SNOOPer**.

The chapter “[SNOOPer-specific Trace Commands](#)”, page 54 describes the SNOOPer-specific configuration commands. While the chapter “[Generic SNOOPer Trace Commands](#)”, page 66 lists the SNOOPer trace analysis and display commands, which are generic for all TRACE32 trace methods.

See also

■ [Trace.METHOD](#)

- ▲ 'Introduction' in 'Application Note for the SNOOPer Trace'
- ▲ 'Generic SNOOPer Trace Commands' in 'General Commands Reference Guide S'
- ▲ 'SNOOPer-specific Trace Commands' in 'General Commands Reference Guide S'
- ▲ 'Release Information' in 'Legacy Release History'

SNOOPer.<specific_cmds>

Overview of SNOOPer-specific commands

See also

- [■ SNOOPer.SELect](#)[■ SNOOPer.SIZE](#)[■ SNOOPer.CORE](#)[■ SNOOPer.Mode](#)
- [■ SNOOPer.PC](#)[■ SNOOPer.Rate](#)[■ SNOOPer.TDelay](#)[■ SNOOPer.TOut](#)
- [■ SNOOPer.TValue](#)
- ▲ 'SNOOPer' in 'General Commands Reference Guide S'

SNOOPer.CORE

Select cores for PC snooping

Format:

SNOOPer.CORE [<number>...]

Selects all or specified cores for PC snooping.

<number>	If no argument is specified, then the command selects all cores.
----------	--

Example: In this script, the cores 0. and 3. of an SMP system are selected for PC snooping with the command **SNOOPer.CORE**. The result is then displayed in the [SNOOPer.List](#) window.

Prerequisite: The cores to be snooped have been assigned to the TRACE32 PowerView GUI with the command [CORE.ASSIGN](#).

```
SNOOPer.state           ;optional step: open the SNOOPer.state window

SNOOPer.Mode PC         ;set the operation mode of the SNOOPer trace
                        ;to PC snooping

SNOOPer.CORE 0. 3.      ;select cores for PC snooping

Go                       ;start SNOOPer trace recording
WAIT 2.s
Break                    ;stop recording

SNOOPer.List            ;display the result
```

record	run	address	cycle	data	symbol	ti.back
-000000001	0	R:000022F0	snoop		\\arm1a\\a_li_aif\\sieve+0x48	218.488ms
-000000001	3	R:0000107C	snoop			218.493ms
-000000001	0	R:000022D8	snoop		\\arm1a\\a_li_aif\\sieve+0x30	218.408ms
-000000001	3	R:0000107C	snoop			218.426ms
-000000001	0	R:0000230C	snoop		\\arm1a\\a_li_aif\\sieve+0x64	218.437ms
-0000	3	R:00000960	snoop			218.462ms
-0000	0	R:000022E0	snoop		\\arm1a\\a_li_aif\\sieve+0x38	218.336ms
-0000	3	R:00001084	snoop		\\arm1a\\Global_main+0x34	218.343ms
-000000000	0	R:00002304	snoop		\\arm1a\\a_li_aif\\sieve+0x5C	218.418ms
-000000000	3	R:00001084	snoop		\\arm1a\\Global_main+0x34	218.394ms
-000000000	0	R:000022C8	snoop		\\arm1a\\a_li_aif\\sieve+0x20	218.248ms
-000000000	3	R:00001080	snoop		\\arm1a\\Global_main+0x30	218.233ms
-000000000	0	R:00002324	snoop		\\arm1a\\a_li_aif\\sieve+0x7C	218.468ms
-000000000	3	R:000010AC	snoop		\\arm1a\\Global_main+0x2C	218.506ms

A 0 in the **run** column stands for core 0.
3 stands for core 3.

See also

■ [SNOOPer.<specific_cmds>](#)

▲ ['Release Information' in 'Legacy Release History'](#)

SNOOPer.ERRORSTOP

Set behavior on sampling errors

Format: **SNOOPer.ERRORSTOP [ON | OFF]**

Default: ON.

Set SNOOPer behavior when on sampling error. When this command is set to OFF, the SNOOPer continues sampling after a sampling error occurs.

Format:

SNOOPPer.Mode <mode>

<mode>:

Memory

PC

PC+MMU

BMC

DCC

ETM

ETM32

SFT

Fifo

Stack

Changes [ON | OFF]

SLAVE [ON | OFF]

StopAndGo [ON | OFF]

AddressTrace [ON | OFF]

FAST [ON | OFF]

ContextID [ON | OFF]

JITTER [ON | OFF]

Selects the operation mode of the SNOOPPer trace. This command can be used to configure the sampling object, the trace recording mode, and various further operation modes.

Sampling objects:

Memory	Samples the contents of up to 16 memory addresses/scalar variables.
PC	Samples the program counter (PC). This operation mode of the SNOOPPer trace is referred to as PC snooping.

PC+MMU	Samples the program counter (PC) and the space ID. This operation mode of the SNOOPer trace is referred to as PC snooping. If the target processor has a memory management unit (MMU) and a target operating system (e.g. Linux) is used, several processes/tasks can run at the same logical addresses. In this scenario, the logical address sampled by the SNOOPer trace is not sufficient to assign the sampled PC to a program location. For a clear assignment, the information about the current task is also required. The PC+MMU mode can be used for this purpose: With every sample, the SNOOPer trace will read the actual program counter and the memory address containing the information about the current task. However, this mode is always intrusive, since the current task and the program counter have to be read exactly at the same time, which can only be achieved by stopping the program execution. For details, refer to your OS Awareness Manual.
BMC	Samples all active benchmark counters .
DCC	Samples data via Debug Communication Channel. (This command is locked if your processor architecture does not provide a Debug Communication Channel.)
ETM	Samples the ETM counter (16-bit). (This command is locked/unknown if your core has no ETM.)
ETM32	Samples the ETM counter (32-bit). (This command is locked/unknown if your core has no ETM.)
SFT	SFT software trace via LPD4 debug mode for RH850 processors. For details refer to “RH850 Debugger and Trace” (debugger_rh850.pdf).

Recording modes:

Fifo	If the SNOOPer trace is full, new records will overwrite older records. The trace always records the last cycles before the program execution is stopped.
Stack	If the SNOOPer trace is full, recording will be stopped. The trace always records the first cycles after starting the program execution.

Further operation modes:

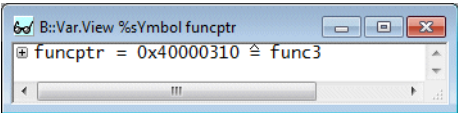
Changes	Samples only data changes.
SLAVE	ON: Ties the trace to the execution of the program, i.e. trace and the trigger work only during user program execution. OFF: Separates the trace from the program execution, i.e. trace is recording even when the program execution is stopped. (This command is only required in exceptional cases).
StopAndGo	Stops the target processor periodically to collect the data of interest. TRACE32 sets this automatically, if no runtime access of the configured sampling object is possible.
AddressTrace	The sampled data values are handled as addresses.
FAST	On certain ARM based derivatives from Texas Instruments (e.g. OMAP3xxx) this mode increases the maximum sampling rate of the snoopers. This mode may not be used in multi-core debug sessions or if the core will be powered down.
ContextID	Samples the ARM Context ID register. This option is only available for some ARM cores.
JITTER	When enabled, this option applies a random jitter to sampling time.

Example for AddressTrace Mode: Sample the content of a function pointer.

```
SNOOPPer.Mode Memory

SNOOPPer.SELect Var.RANGE(funcptr)

...
```



B::SNOOP.List							
Setup... Goto... Find... Chart Draw Profile							
record	run	address	cycle	data	symbol	ti.back	
-0000000004		SD:400040D4	snoop	40000310	\\diabc\Global\funcptr	69.784us	
-0000000003		SD:400040D4	snoop	40000520	\\diabc\Global\funcptr	69.904us	
-0000000002		SD:400040D4	snoop	40000780	\\diabc\Global\funcptr	69.816us	
-0000000001		SD:400040D4	snoop	40000310	\\diabc\Global\funcptr	8.890ms	

```
SNOOPPer.Mode AddressTrace ON ;advise TRACE32 to display the
                                ;data value as address
```

B::SNOOP.List							
Setup... Goto... Find... Chart Draw Profile							
record	run	address	cycle	data	symbol	ti.back	
-0000000004		P:40000310	snoop		\\diabc\diabc\func3	69.784us	
-0000000003		P:40000520	snoop		\\diabc\diabc\func8	69.904us	
-0000000002		P:40000780	snoop		\\diabc\diabc\func4	69.816us	
-0000000001		P:40000310	snoop		\\diabc\diabc\func3	8.890ms	

See also

- [SNOOPPer.<specific_cmds>](#)
- [<trace>.Mode](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format:	SNOOPer.PC [ON OFF]
---------	------------------------------

Reads the PC without stopping the target (PC-snooping).

- SNoop.PC**

Prints the current PC in the [state line](#) (only once).
- SNoop.PC ON | OFF**

Enables or disables permanent updates of the PC in the state line.

See also

- [SNOOPer.<specific_cmds>](#)

Format: **SNOOPer.Rate** *<value>* | *<time>*

Selects the sampling rate. The rate can be specified as time interval or as number of samples/s. The defined rate is not guaranteed. The actual frequency used by the SNOOPER may be lower depending on the target CPU and the sampling object.

Example:

```
SNOOPPer.Rate 1000000.           ; set to 1 MHz sample rate
SNOOPPer.Rate 1.us                ; same operation, 1 MHz sample rate
```

See also

- SNOOPer.<specific_cmds>

Define address for monitoring

[Examples]

Format: **SNOOPer.SELect** [%<format>] <address> | <range>] [%<format>]
 <address> | <range>...] [/<option>]

<format>: Byte | Word | Long | Quad | TByte | PByte | HByte | SByte
CORE <core_number>

<option>: DIALOG

Defines the sampling addresses for the SNOOPer trace **Memory** mode. Up to 16 sampling addresses can be specified using the **SNOOPer.Select** command. The parameter can be an address or an address range. If the parameter is a single address, the access site is per default one byte. This is also true if a symbolic address is used (e.g. HLL variable).

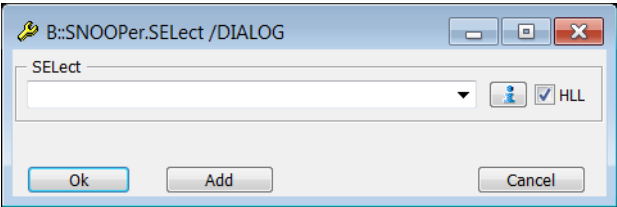
When executed without arguments, the **SNOOPer.SELect** command clears all previously set sampling addresses.

Byte (default), Word , TByte , Long , PByte , HByte , SByte	Access size • Byte (8-bit accesses) • TByte (24-bit accesses) • PByte (40-bit accesses) • SByte (56-bit accesses) • Word (16-bit accesses) • Long (32-bit accesses) • HByte (48-bit accesses) • Quad (64-bit accesses)
CORE <core_number>	Performs the sampling on the specified core.
DIALOG	If the SNOOPer.SELect command is entered with the DIALOG option, a dialog is displayed.

Examples

Example 1:

```
SNOOPer.SELect /DIALOG
```



Example 2:

```
SNOOPer.SELect 0x10000000      ; sample one byte from address 0x1000
SNOOPer.SELect mcount          ; sample one byte from integer variable
                                ; mcount
```

Example 3: If more than one byte have to be sampled, the access size has to be specified using the *<format>* option, e.g. %Word or %Long.

```
SNOOPer.SELect %Word 0x1000    ; sample two bytes from address 0x1000
SNOOPer.SELect %Long mcount    ; sample four bytes from integer
                                ; variable mcount
```

Example 4: If the parameter is an address range, the access size is automatically set to the size of the range.

```
SNOOPer.SELect 0x1000--0x1001      ; sample two bytes from address 0x1000

SNOOPer.SELect Var.RANGE(mcount)   ; sample the address range of the
                                   ; variable mcount
```

Var.RANGE(<hll_expression>)

Returns the address range occupied by the specified HLL expression

Example 5: If more than one address should be sampled, the addresses have to be specified using one single **SNOOPer.SELect** command. The access size has to be specified for each sampling address.

```
SNOOPer.SELect %Long mcount %Word plot1 0x1000--0x1001
```

See also

■ [SNOOPer.<specific_cmds>](#)

SNOOPer.SIZE

Define trace buffer size

Format: **SNOOPer.SIZE** <records>

Sets the size of the SNOOPer trace memory. The size is specified in number of records (samples). TRACE32 PowerView allocates memory on the host for the requested size. The SNOOPer trace buffer size is thus only limited by the resources of the host.

See also

■ [SNOOPer.<specific_cmds>](#)

SNOOPer.TDelay

Define trigger delay

Format: **SNOOPer.TDelay** <records> | <percent>%

Selects the delay between the trigger point and the execution of the trigger action defined with **SNOOPer.TOut**. The delay can be specified in number of records or as percentage of the SNOOPer trace depth.

Example:

```
SNOOPPer.TDelay 1000.      ; sample 1000. records after the ;trigger
                           ; point then execute the trigger action.

SNOOPPer.TDelay 40%        ; continue with the sampling after the
                           ; trigger point until 40% of the trace
                           ; buffer are filled then execute the
                           ; trigger action.
```

See also

■ [SNOOPPer.<specific_cmds>](#)

SNOOPPer.TOut

Define the trigger destination

Format: **SNOOPPer.TOut** *<trigger_action>*

<trigger_
action>: **Trace | Program | PULSE | BUSA**

Defines the *<trigger_action>* that should be executed when the value defined with the [SNOOPPer.TValue](#) command is sampled. This command is used in conjunction with the SNOOPPer **Memory** and **PC** modes.

Trace	Stop the SNOOPPer trace recording.
Program	Stop program execution.
PULSE	Trigger pulse generator.
BUSA	Trigger bus line A.

See also

■ [SNOOPPer.<specific_cmds>](#)

Format:

SNOOPPer.TValue <value> | <range> | <bitmask>

Defines the data value, data value range or bit mask that should trigger the action defined with the **SNOOPPer.TOut** command. This command is used in conjunction with the SNOOPPer **Memory** and **PC** modes.

Example:

```
SNOOPPer.TValue 0x1           ; trigger the TOut action when the value
                               ; 0x1 is sampled

SNOOPPer.TValue 0xA00--0xAFF  ; trigger the TOut action when a value
                               ; within the data range 0xA00--0xAFF is
                               ; sampled

SNOOPPer.TValue !0            ; trigger the TOut action when a value
                               ; different from 0 is sampled
```

See also

- [SNOOPPer.<specific_cmds>](#)

Generic SNOOPer Trace Commands

SNOOPer.ACCESS Define access path to program code for trace decoding

See command [`<trace>.ACCESS`](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 131).

SNOOPer.Arm Arm the trace

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

SNOOPer.AutoArm Arm automatically

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

SNOOPer.AutoInit Automatic initialization

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

SNOOPer.BookMark Set a bookmark in trace listing

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

SNOOPer.BookMarkToggle Toggles a single trace bookmark

See command [`<trace>.BookMarkToggle`](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 143).

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

See command [**<trace>.Chart.DistriB**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 159).

See command [**<trace>.Chart.sYmbol**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 173).

See command [**<trace>.Chart.VarState**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 189).

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

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

See command [**<trace>.DRAW**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 201).

See command [**<trace>.DRAW.channel**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 204).

See command [**<trace>.DRAW.Var**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 210).

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

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

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

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

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

See command [<trace>.Get](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 242).

SNOOPPer.GOTO

Move cursor to specified trace record

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

SNOOPPer.Init

Initialize trace

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

SNOOPPer.List

List trace contents

See command [<trace>.List](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 248).

SNOOPPer.ListVar

List variable recorded to trace

See command [<trace>.ListVar](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 266).

SNOOPPer.LOAD

Load trace file for offline processing

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

SNOOPPer.OFF

Switch off

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

SNOOPPer.PROfileChart

Profile charts

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

See command [**<trace>.PROfileChart.COUNTER**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 293).

SNOOPer.PROfileSTATistic

Statistical analysis in a table versus time

See command [**<trace>.PROfileSTATistic**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 322).

SNOOPer.PROTOcol

Protocol analysis

See command [**<trace>.PROTOcol**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 339).

SNOOPer.PROTOcol.Chart

Graphic display for user-defined protocol

See command [**<trace>.PROTOcol.Chart**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 339).

SNOOPer.PROTOcol.Draw

Graphic display for user-defined protocol

See command [**<trace>.PROTOcol.Draw**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 341).

SNOOPer.PROTOcol.EXPORT

Export trace buffer for user-defined protocol

See command [**<trace>.PROTOcol.EXPORT**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 342).

SNOOPer.PROTOcol.Find

Find in trace buffer for user-defined protocol

See command [**<trace>.PROTOcol.Find**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 343).

See command [**<trace>.PROTOcol.list**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 344).

See command [**<trace>.PROTOcol.PROfileChart**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 347).

See command [**<trace>.PROTOcol.PROfileSTATistic**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 348).

See command [**<trace>.PROTOcol.STATistic**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 350).

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

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

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

See command [**<trace>.SelfArm**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 362).

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

See command [**<trace>.state**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 376).

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

See command [**<trace>.STATistic.DistriB**](#) in 'General Commands Reference Guide T' (general_ref_t.pdf, page 401).

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

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

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

SNOOPer.ZERO

Align timestamps of trace and timing analyzers

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

PowerPC 55xx/85xx only

The **SPE** command group is used to display and modify the SPE (Signal Processing eXtension) registers for PowerPC.

See also

- [SPE.Init](#)
- [SPE.Set](#)
- [SPE.view](#)
- [SPE\(\)](#)
- ▲ 'SPE Function' in 'General Function Reference'

SPE.Init

Initialize SPE registers

PowerPC 55xx/85xx only

Format:

SPE.Init
SPE.RESet (deprecated)

Initializes all SPE registers to zero.

See also

- [SPE](#)
- [SPE.view](#)

SPE.Set

Modify SPE registers

PowerPC 55xx/85xx only

Format:

SPE.Set <register> <value> [/<option>]

<register>:

R0..R31 | ACC | SPEFSCR

Writes the given value to the specified SPE register.

- <option>
- For a description of the options, see [Register.view](#).
- R0..R31 and ACC
- Are 64-bit values that are entered as 16-digits hex values. See [example](#).

Example:

```
SPE.Set R15 0x123456789ABCDEF0
SPE.Set ACC 0xFFFFFFFFFFFFFFFF
```

See also

- [SPE](#)
- [SPE.view](#)

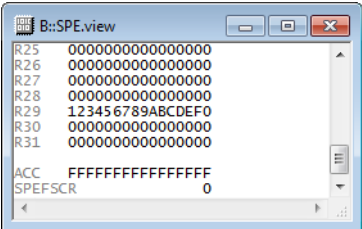
SPE.view

Display SPE register window

PowerPC 55xx/85xx only

Format: **SPE.view** [/<option>]

Opens a window displaying the SPE vector registers **R0..R31**, **ACC** and **SPEFSCR**.



<option> For a description of the options, see [Register.view](#).

See also

- [SPE](#)
- [SPE.Init](#)
- [SPE.Set](#)
- [SPE\(\)](#)

SSE

SSE registers (Streaming SIMD Extension)

Intel® x86

The **SSE** command group is used to display and modify the SSE (Streaming SIMD Extensions) registers for Intel® x86.

See also

- [SSE.Init](#)
- [SSE.Set](#)
- [SSE.view](#)
- [SSE\(\)](#)
- ▲ 'Command Groups for Special Registers' in 'Intel® x86/x64 Debugger'
- ▲ 'SSE Function' in 'General Function Reference'

SSE.Init

Initialize SSE registers

Intel® x86

Format: **SSE.Init**

Sets the SSE registers to their default values.

See also

- [SSE](#)

SSE.Set

Modify SSE registers

Intel® x86

Format: **SSE.Set** <register> <value> ... [/<option>]

Modifies the SSE registers.

<option>

For a description of the options, see [Register.view](#).

See also

- [SSE](#)

Format: **SSE.view** [/<option>]

Displays the SSE registers.

<option>

For a description of the options, see [Register.view](#).

See also

- [SSE](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

For a description of the **StatCol** commands, see “[System Trace User’s Guide](#)” (trace_stm.pdf).

Step

Single-step

Using the **Step** command group, you can step through the program in a controlled way, executing an assembly opcode, a source line or a function at a time.

See also

- [Step.Asm](#)
- [Step.Back](#)
- [Step.BackChange](#)
- [Step.BackOver](#)
- [Step.BackTill](#)
- [Step.Change](#)
- [Step.Diverge](#)
- [Step.Hll](#)
- [Step.Mix](#)
- [Step.Over](#)
- [Step.single](#)
- [Step.Till](#)
- [Go](#)
- [List](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Step.Asm

Assembler single-step

Format:

Step.Asm [*<count>*]

Switches to assembler mode before performing the required single steps via the **Step** command. The performed steps are assembly steps.

See also

- [Step](#)
- [Step.single](#)

Step.Back

Step backwards

Format:

Step.Back

This command can only be used together with the Context Tracking System (**CTS**). The command steps back one assembler instruction or one HLL line. Under certain conditions, the command automatically activates **CTS** when it is turned off.

See also

- [Step](#)
- [Step.single](#)

Format: **Step.BackChange**

Steps back till the expression changes. The command will stop also, if the expression cannot be evaluated.

Example:

```
Step.BackChange Register(A7)           ; steps till register A7 changes

Step.BackChange                        ; steps till the longword at
Data.Long(sd:0x100)                  location
                                   ; 100 changes
```

See also

■ [Step](#)

■ [Step.single](#)

▲ ['Release Information' in 'Legacy Release History'](#)

Format: **Step.BackOver**

This command can only be used together with the Context Tracking System (**CTS**). The command steps back one assembler instruction or one HLL line.

Under certain conditions, the command automatically activates **CTS** when it is turned off.

See also

■ [Step](#)

■ [Step.single](#)

▲ ['Release Information' in 'Legacy Release History'](#)

Format: **Step.BackTill** [*<boolean_expression>*]

Steps back till the boolean expression becomes true. The command will stop also, if the expression cannot be evaluated.

Example:

```
Step.BackTill Register(A7)>0x1000      ; steps till register A7 is larger
                                         ; than 1000

Step.BackTill                           ; steps till the longword at
Data.Long(sd:0x100)==0x0                ; location 100 gets the value 0
```

See also

- [Step](#)
- [Step.single](#)

Step.Change

Step until expression changes

Format:

Step.Change [*<expression>*]

Steps till the expression changes. The command will stop also, if the expression cannot be evaluated.

Example:

```
Step.Change Register(A7)                ; steps till register A7 changes

Step.Change Data.Long(SD:0x100)         ; steps till the long word at
                                         ; location 100 changes
```

See also

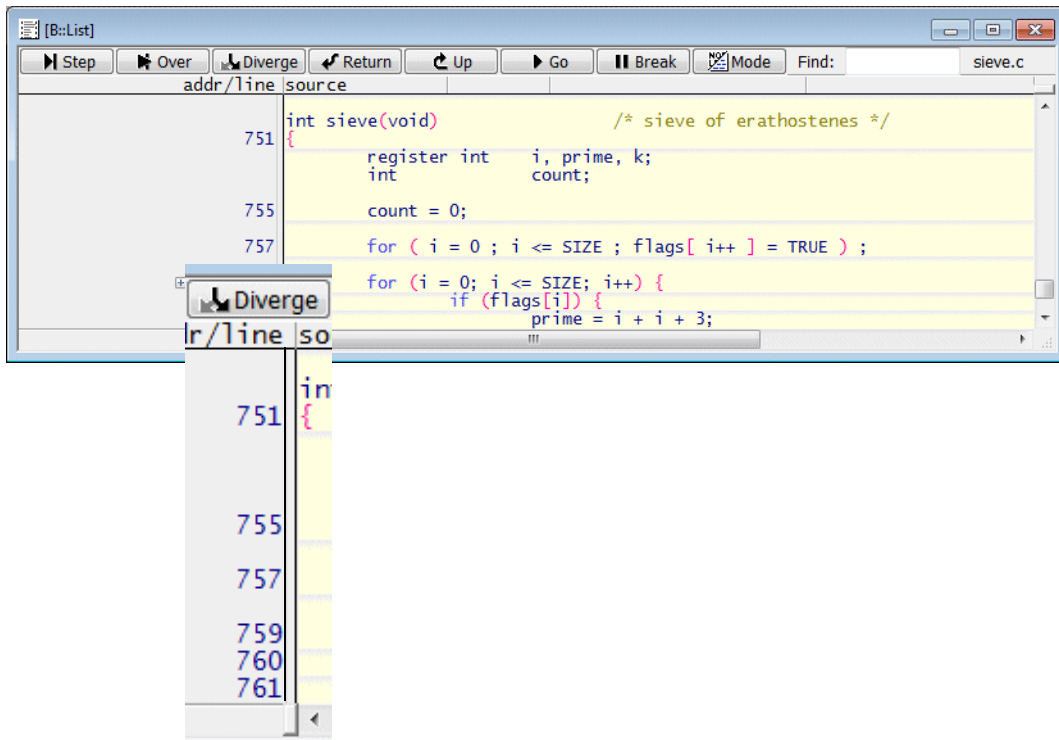
- [Step](#)
- [Step.single](#)

Format: **Step.Diverge**

The **Step.Diverge** command can be used to exit loops or to fast forward to not yet reached HLL lines. It performs **Step.Over** repeatedly until an HLL line is reached which has not been reached in the previous steps.

TRACE32 maintains a list of all HLL lines which were already reached. These reached lines are marked with a slim grey line in the List window (see picture below).

In ASM/MIX mode, **Step.Diverge** applies to assembler code lines instead of HLL lines.

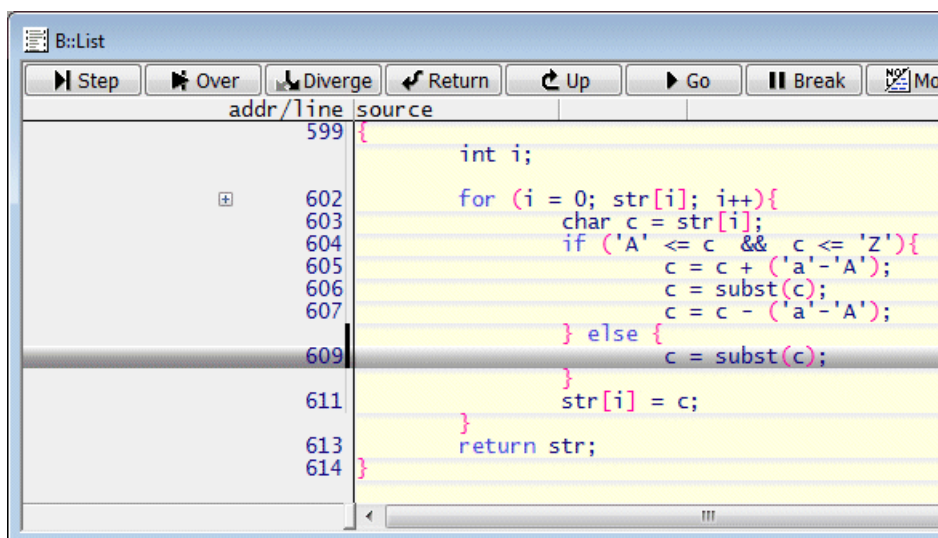
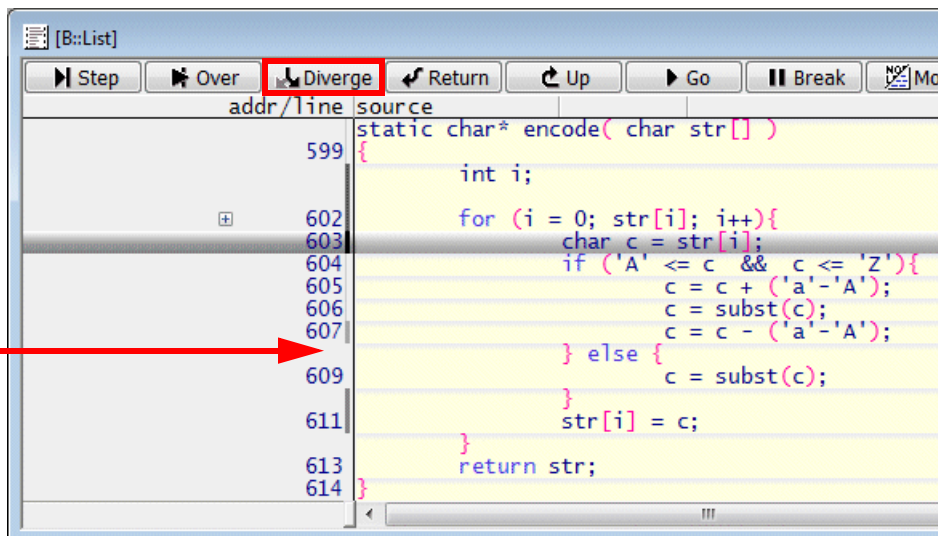


The *reached lines list* is cleared when you use the **Go.direct** command without address or the **Break** command while the program execution is stopped.

The *reached lines list* is not cleared at the following commands:

- **Step.single**, **Step.Over**, **Step.Change** <expression>, **Step.Till** <condition>
- **Var.Step.Change** <hll_expression>, **Var.Step.Till** <hll_condition>
- **Go.Return**, **Go.Up**, **Go.direct** <address>
- **Var.Go.direct** <hll_expression>

The debugger did not reach the **else** branch yet



See also

■ [Step](#)

■ [Step.single](#)

▲ ['Release Information' in 'Legacy Release History'](#)

Format: **Step.Hll** [*<count>*]

Similar to the **Step.single** command, except that simultaneous switching into high-level language mode occurs.

See also

[■ Step](#)[■ Step.Mix](#)[■ Step.single](#)

[▲ 'Release Information' in 'Legacy Release History'](#)

Format: **Step.Mix** [*<count>*]

Similar to the **Step.single** command, except that simultaneous switching into mixed mode takes place.

See also

[■ Step](#)[■ Step.Hll](#)[■ Step.single](#)

Format:	Step.Over
---------	------------------

Steps within a function and runs called functions in real-time.

The method for this command is depends, whether the operation-mode is HLL or assembler (ASM/MIX).

ASM	In assembler mode the debugger reads the instruction at the current PC. On a CALL instruction a Go.Next command is executed. All other instructions will cause a regular single-step command.
HLL	In HLL mode the system first executes an HLL single-step. After this step it checks, whether the PC is still in the same function. If the PC has left the function it will check the value addressed by the SP. With that value being within the original function, the program is continued to that point. If this address contains no HLL breakpoint the above procedure will be repeated (HLL step ...).

See also

- [Step](#)
- [Step.single](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Step.single

Single-step

Format:	Step.single [<i><count></i>]
---------	---

Executes one program step until the next assembler instruction or HLL line, depending on the current debug mode, is reached. *<count>* is the number of command executions (default is 1).

Examples:

Step.single	; single step
Step.single 10.	; 10 steps

See also

- [Step](#)
- [Step.BackOver](#)
- [Step.Hll](#)
- [SETUP:IMASKASM](#)
- [Step.Asm](#)
- [Step.BackTill](#)
- [Step.Mix](#)
- [SETUP:IMASKHLL](#)
- [Step.Back](#)
- [Step.Change](#)
- [Step.Over](#)
- [Var.Step](#)
- [Step.BackChange](#)
- [Step.Diverge](#)
- [Step.Till](#)

Format:

Step.Till [*<boolean_expression>*]

Steps till the boolean expression becomes true. The command will stop also if the expression cannot be evaluated.

Examples:

```
Step.Till Register(A7)>0x1000      ; steps till register A7 is larger
                                   ; than 1000

Step.Till Data.Long(SD:0x100)==0x0 ; steps till the long word at
                                   ; location 100 gets the value 0
```

See also

- [Step](#)
- [Step.single](#)

STM by ARM, STM and STDI by Texas Instruments

A system trace is a hardware module on a SoC which enables the developer to output predefined hardware or software messages without affecting the run-time behavior of the system.

For a description of the **STM** commands, see [“System Trace User’s Guide”](#) (trace_stm.pdf).

STOre

Store settings as PRACTICE script

Format:	STOre <file> [[%<format>]<item> ...] [/<option>]
<format>:	sYmbol NosYmbol
<item>:	default ALL Win WinPAGE WinTOP SYStem ...
<option>:	NoDate

Stores settings in the format of a PRACTICE script (*.cmm). The script can be executed by using the **DO** command.

<format>	Description
sYmbol	Addresses (e.g. for the commands Break or GROUP) are stored as symbols. With this option, breakpoints can be stored and recalled for a newer version of the program with different addresses. The keyword must be entered before the item which shall be stored. The default can be set with SETUP.STOre.SYMBOLIC .
NosYmbol	Addresses (e.g. for the commands Break or GROUP) are stored as scalar values (plain hex). With this option, stored breakpoints can be recalled when no debug symbols are available. The keyword must be entered before the item which shall be stored.

<item>	Description
no item specified	If no item is specified, then the default setting is used; see default below.
AREA	Store the current AREA settings.
ALL	Store all settings.
Analyzer	See Analyzer command.
AnalyzerFocus	Save the current AUTOFOCUS configuration to a file.
ART	See ART command.
BookMark	Store the settings of trace bookmarks and address bookmarks - see BookMark command. To export bookmarks as an XML file, see BookMark.EXPORT .
Break	Store breakpoints - see Break command.
BSDL	Store the boundary scan settings. See BSDL command.

<item>	Description
CAnalyzer	See CAnalyzer command.
CAnalyzerFocus	Store the electrical settings for the CAnalyzer. See CAnalyzer.ShowFocus .
CIProbe	See CIProbe command.
Count	See Count command.
default	Some settings are stored by default, except for the window setting .
EDITOR	Store the auto-indentation settings, etc. for all TRACE32 editors. See SETUP.EDITOR .
FDX	See FDX command.
FLASH	(For diagnostic purposes only.) Store the FLASH declaration displayed in the FLASH.List window and the settings made with the FLASH.TARGET command.
GLOBAL	Stores global PRACTICE macros with the current values - see GLOBAL command.
GROUP	See GROUP command.
HELP	Stores help settings and bookmarks - see HELP command
HISTory	See HISTory command.
LA	Logic Analyzer - see LA command.
LOGGER	See LOGGER command.
MAP	See MAP command.
MARKER	See sYmbol.MARKER .
NAME	See NAME command.
NoDate (deprecated)	Omit the date at the beginning of the generated script. Use the option /NoDate instead. This item is deprecated since Sep/2023.
On-chip	See Onchip command.
PBREAK	Stores the breakpoints created for PRACTICE scripts (*.cmm). See PBREAK command group.
PERF	Performance Analyzer - see PERF command.
POD	See POD command.
Register, RegSet	See Register command.
SNOOP	See SNOOP command.
SPATH	Source search path - see sYmbol.SPATH command.
SPATHCACHE	Stores cached directories from the sYmbol.SPATH command.

<item>	Description
Symbolic HEX (deprecated)	Addresses are stored as symbols or scalar values (plain hex). Use the format parameter %sYmbol or %NosYmbol instead.
SYnch	See SYnch command.
SYStem	See SYS command.
TRANSlation	Store all static address TRANSlations as well as all common, transparent and protected address ranges as displayed by command TRANSlation.List .
TrOnchip	Trigger Onchip - see TrOnchip command.
TrPOD	Trigger Probe - see TP command.
VCO	See VCO command.
Win	Store entire window configuration.
WinPAGE	Store the current window page. See WinPAGE command.
WinTOP	Store the active window. See WinTOP command.

<option>	Description
NoDate	Omit the comment containing the current date at the beginning of the generated PRACTICE script file (*.cmm).

Example: executing the following STOrE command in TRACE32 PowerView for ARM64 connected to Zynq-Ultrascale+ processor

```
STOrE ~~~\test.cmm SYStem
```

produces the following file:

```
B::  
  
SYStem.RESet  
SYStem.CPU ZYNQ-ULTRASCALE+-APU  
SYStem.CONFIG CoreNumber 4.  
SYStem.CONFIG CORE 1. 1.  
CORE.ASSIGN 1. 2. 3. 4.  
SYStem.MemAccess DAP  
SYStem.CpuBreak Enable  
SYStem.CpuSpot Enable  
SYStem.JtagClock 10.MHz  
SYStem.Option.MMUPLM OFF  
SYStem.Option.ENRESET OFF  
SYStem.Option.TRST OFF  
SYStem.CONFIG.DAPIRPRE 0.  
SYStem.CONFIG.DAPDRPRE 0.  
SYStem.CONFIG.DAPIRPOST 12.  
SYStem.CONFIG.DAPDRPOST 1.  
SYStem.CONFIG.SLAVE OFF  
SYStem.CONFIG.TAPState SElectDRscan  
SYStem.Mode Up  
  
ENDDO
```

See also

■ [AutoSTOre](#)
■ [SETUP:STOre](#)

■ [BookMark.List](#)

■ [ClipSTOre](#)

■ [SETUP.QUITDO](#)

SVE

SVE

Access the scalable vector extension SVE

This command group allows to access the scalable vector extension (SVE). They are available for the processor architecture ARMv8.2 and newer.

SVE.Init

Initialize SVE registers

Format:	SVE.Init
---------	-----------------

Sets the SVE registers to their default values.

SVE.RESet

Reset SVE settings

Format:	SVE.RESet
---------	------------------

Resets debugger SVE settings.

SVE.Set

Modify SVE registers

Format:	SVE.Set
---------	----------------

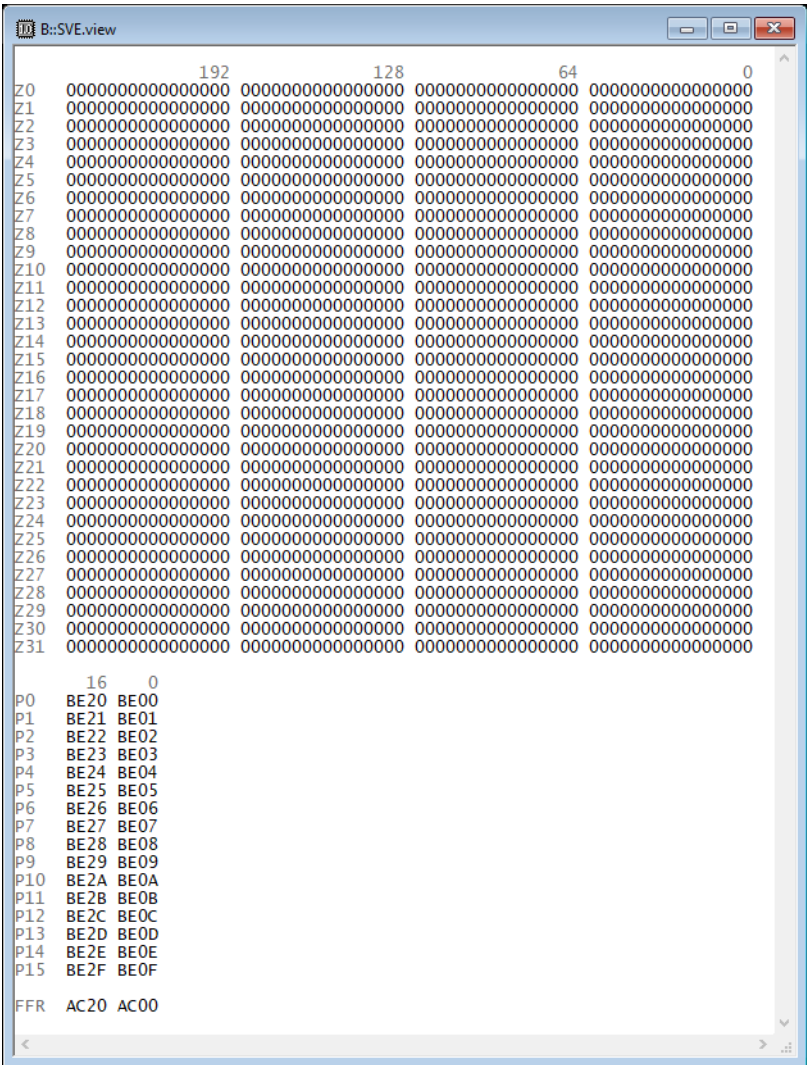
Allows to modify SVE registers. Predicate registers and the FFR cannot be modified.

Registers can be modified with either floating-point values, or hexadecimal values.

Format: SVE.view

All accessible SVE registers are displayed in a window. The contents of this window depend on the implemented vector size.

The FFR register cannot be displayed. Predicate registers are read only.



See also

- [■ sYmbol.AddInfo](#)
[■ sYmbol.CHECK](#)
[■ sYmbol.ColorDef](#)
[■ sYmbol.DeleteMACRO](#)
[■ sYmbol.DONE](#)
[■ sYmbol.LANGUAGE](#)
[■ sYmbol.MATCH](#)
[■ sYmbol.NAMESPACES](#)
[■ sYmbol.POINTER](#)
[■ sYmbol.RESet](#)
[■ sYmbol.SourcePATH](#)
[■ sYmbol.TYPEINFO](#)
- [■ sYmbol.AutoLOAD](#)
[■ sYmbol.Class](#)
[■ sYmbol.CREATE](#)
[■ sYmbol.DeletePATtern](#)
[■ sYmbol.ECA](#)
[■ sYmbol.List](#)
[■ sYmbol.MEMory](#)
[■ sYmbol.NEW](#)
[■ sYmbol.POSTFIX](#)
[■ sYmbol.SourceBeautify](#)
[■ sYmbol.SourceRELOAD](#)
[■ sYmbol.View](#)
- [■ sYmbol.Browse](#)
[■ sYmbol.CLEANUP](#)
[■ sYmbol.CUTLINE](#)
[■ sYmbol.DEMangle](#)
[■ sYmbol.ForEach](#)
[■ sYmbol.LSTLOAD](#)
[■ sYmbol.Modify](#)
[■ sYmbol.OVERLAY](#)
[■ sYmbol.PREFIX](#)
[■ sYmbol.SourceCONVert](#)
[■ sYmbol.STATE](#)
- [■ sYmbol.CASE](#)
[■ sYmbol.ColorCode](#)
[■ sYmbol.Delete](#)
[■ sYmbol.DEOBFUSCATE](#)
[■ sYmbol.INFO](#)
[■ sYmbol.MARKER](#)
[■ sYmbol.name](#)
[■ sYmbol.PATCH](#)
[■ sYmbol.RELOCate](#)
[■ sYmbol.SourceLOAD](#)
[■ sYmbol.STRIP](#)
- [▲ 'sYmbol Functions' in 'General Function Reference'](#)
[▲ 'Release Information' in 'Legacy Release History'](#)

Overview sYmbol

Statics	Contains all static symbols, i.e. symbols with a fixed address. The symbol may be local related to a function, a module or a program.
Functions	Contains all function blocks and additional information about functions, i.e. virtual frame pointers, register usage, stack frame layout.
Locals	All local symbols of a function. The ranges of validity are also contained in this table.
Modules	A module is one separately compiled program unit, i.e. one source file.
Types	High level language types and the physical description. Only named types are included in this table.
Sources	Contains a list of all HLL source files and the path names to the sources.
SPATH	List of directories of the source search path.

Lines	High level language source lines, respectively blocks. On one address can be one high level language block only.
Sections	Logical and physical program address ranges. According to the compiler a differentiation between 'CODE', 'DATA', 'BSS', 'ROMABLE' etc. is made. For each section special access rights are valid.
Programs	Usually only one program is loaded. If more than one has been loaded, the option NoClear must be used together with the Data.LOAD command.
Stacks	Contains information about the stack frame. Usually this is the offset between a register in the processor and a “virtual frame pointer”. This information is needed when there is no real frame pointer register used by highly optimizing compilers.
Attributes	This table is target dependent. It may contain information about different processor executing environments (e.g. ARM/Thumb) or special code constructs (e.g. jump tables or literal data in code).
Macros	Contains the contents of preprocessor macros. This information is not available for all compilers. However it may be generated manually with the sYmbol.CREATE.MACRO command.
Map	Holds a log of all memory operations during download. Only maintained when the option MAP was set with the load command.
Compilers	This listing contains compiler specific information. It cannot be displayed and is determined on internal use by the HLL debugger

PRACTICE Functions

Refer to “**sYmbol Functions**” in General Function Reference, page 304 (general_func.pdf).

Format:	sYmbol.AddInfo
---------	----------------

The command can provide additional information about structures, pointers or variables. The information can scale the display, make typecasts or provide application specific interpretation of information (e.g. C++ descriptor displays).

Here is a list of additional symbolic information types that can be set (not all symbolic information types are available depending on what the information is assigned to):

Scaled <multiplier> [<offset> <explanation>]]	Scales the value of the element: $trueValue = (srcValue * multiplier) + offset.$
RScaled <multiplier> [<offset> [<explanation>]]	Scales the value of the element: $trueValue = (multiplier / srcValue) + offset$
<explanation>	Any user-defined text, such as class name, meaning, type, unit of measurement, etc. (See example)
JSTRING	Handles the element as pointer pointing to a string stored in the Java jstring representation. (See example)
NSTRING <bitmask>	Handles the element as pointer pointing to a structure, where the first element is the string length and the next element is the string. (See example)
ZSTRING	Handles the element as pointer pointing to a C-like string (string terminated by zero). (See example)
MaskedPointer <mask> [<offset>]	Modifies the pointer target address: $truePointer = (srcPointer \& mask) offset$ (See example)
MostDerived <struct/class_name>	Forces data referenced by a pointer to be displayed as data of the declared struct or class (See example)
DESCRIPTOR	Forces data references by a pointer to be displayed according to a descriptor.
ENUM <item_value> <item_name>	Adds a name for a value inside a variable.
Hex	Marks the element to be displayed in hexadecimal.
Decimal	Marks the element to be displayed in decimal.

Ascii	Marks the element to be displayed as ASCII.
sYmbol	Marks the element to be displayed as a pointer to a symbol.
HIDE	Hides the element from watch windows.
BigEndian	Forces access to element in BigEndian byte order.
LittleEndian	Forces access to element in LittleEndian byte order.
LINK <link_name>	Format an element as defined by the sYmbol.AddInfo.LINK command.

See also

- [sYmbol.AddInfo.Address](#)
 - [sYmbol.AddInfo.LINK](#)
 - [sYmbol.AddInfo.LOADASAP2](#)
 - [sYmbol.AddInfo.RESet](#)
 - [sYmbol.AddInfo.Var](#)
- [sYmbol.AddInfo.Delete](#)
 - [sYmbol.AddInfo.List](#)
 - [sYmbol.AddInfo.Member](#)
 - [sYmbol.AddInfo.Type](#)
 - [sYmbol](#)

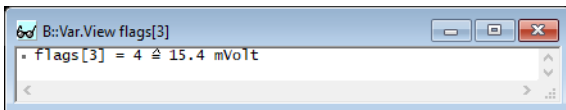
Format: **sYmbol.AddInfo.Address** <range> | <address> <info> [<parameters>]

<info>: **Scaled** <multiplier> [<offset> [<explanation>]]
RScaled <multiplier> [<offset> [<explanation>]]
...

Adds scaling information to an address or an address range. All symbolic information types are described in [sYmbol.AddInfo](#).

Example 1:

```
; multiply each HLL variable that is located in the address range  
; 0xA1080000++0xff by 1.34 and add 10. Use mVolt as unit  
  
sYmbol.AddInfo.Address 0xA1080000++0xff Scaled 1.34 +10. " mVolt"
```



Example 2:

```
; multiply the reciprocal contents of each HLL variable that is located  
; in the address range 0xA1080000++0xff by 20. and add +3.3. Use mA as  
; unit  
  
sYmbol.AddInfo.Address 0xA1080000++0xff RScaled 20. +3.3 " mA"
```



See also

■ [sYmbol.AddInfo](#)

Format: **sYmbol.AddInfo.Delete** <name>

Deletes existing information from the given variable or type name.

Example:

```
sYmbol.AddInfo.Var cstr1 ZSTRING  
sYmbol.AddInfo.Delete "cstr1"
```

See also

■ [sYmbol.AddInfo](#)

■ [sYmbol.AddInfo.RESet](#)

Format: **sYmbol.AddInfo.LINK** <link_name> <info> [<parameters>]

<info>: **NSTRING** <bitmask>
JSTRING
ZSTRING
MostDerived <struct | class>
DESCRIPTOR <bitmask> <struct | class>
...

Defines information for other **sYmbol.AddInfo** commands. All symbolic information types are described in [sYmbol.AddInfo](#).

Example:

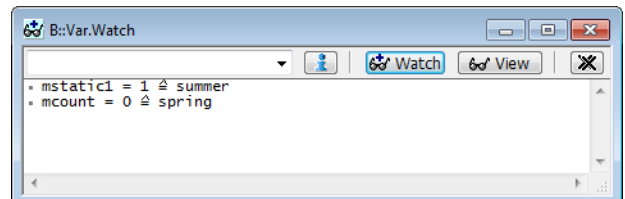
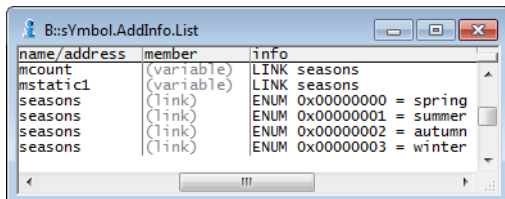
```
sYmbol.AddInfo.List

;1. create description group 'seasons'
;2. link meaningful description to each numerical value
sYmbol.AddInfo.LINK seasons enum 0 "spring"
sYmbol.AddInfo.LINK seasons enum 1 "summer"
sYmbol.AddInfo.LINK seasons enum 2 "autumn"
sYmbol.AddInfo.LINK seasons enum 3 "winter"
; equivalent to C statement symbols for
; "enum seasons {spring, summer, autumn, winter};"

;link the integer variables to the description group 'seasons'
sYmbol.AddInfo.Var mcount LINK seasons
sYmbol.AddInfo.Var mstatic1 LINK seasons

Var.Watch
Var.AddWatch mcount
Var.AddWatch mstatic1

Var.set mcount=0
Var.set mstatic1=1
```



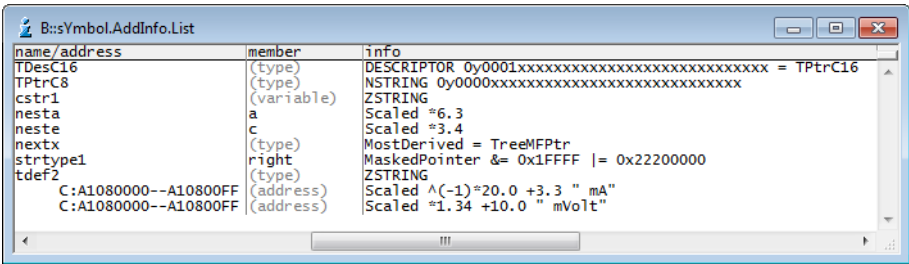
See also

■ [sYmbol.AddInfo](#)

Format:

sYmbol.AddInfo.List

Shows all available additional information which has been declared by **sYmbol.AddInfo** commands.



See also

- [sYmbol.AddInfo](#)

sYmbol.AddInfo.LOADASAP2

Load scaling information from ASAP2 file

Format:

sYmbol.AddInfo.LOADASAP2 <file>

Loads the scaling and physical unit information from an ASAP2 file.

See also

- [sYmbol.AddInfo](#)
- [Data.LOAD.ASAP2](#)
- ▲ 'Release Information' in 'Legacy Release History'

Format: sYmbol.AddInfo.Member <type> <member> <info> [<parameters>]

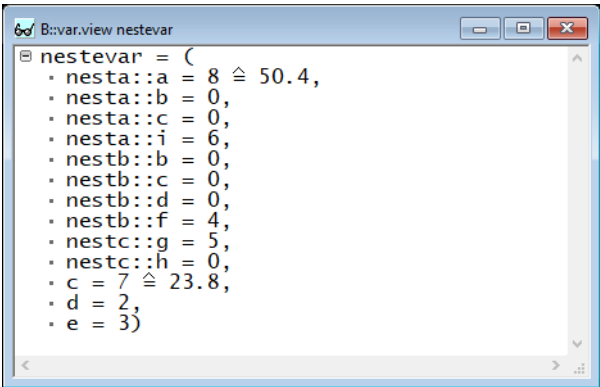
<info>: Scaled <multiplier> [<offset> [<explanation>]]
RScaled <multiplier> [<offset> [<explanation>]]
ZSTRING
MaskedPointer <mask> <offset>
MostDerived <struct/class_name>
...

Add information to specific member of a specific struct or class. All symbolic information types are described in [sYmbol.AddInfo](#).

- <type>Name of struct or class.
- <member>Name of struct or class element.

Example 1

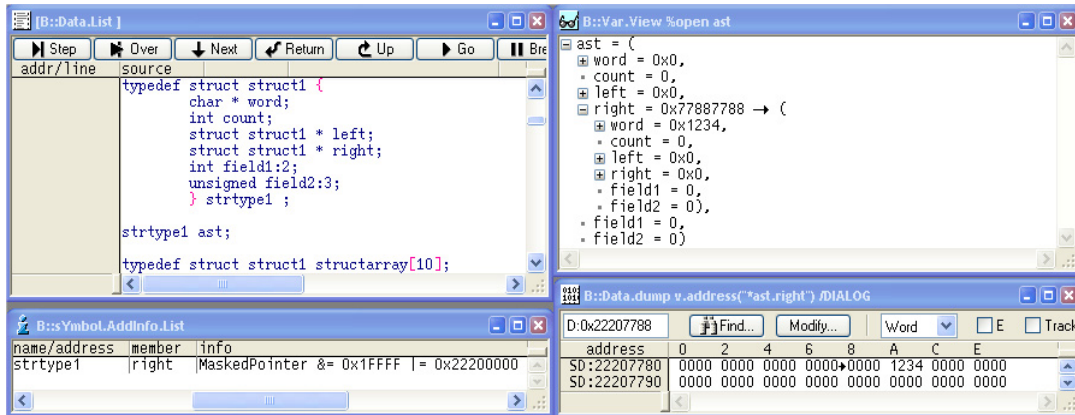
```
sYmbol.AddInfo.Member neste c Scaled 3.4  
sYmbol.AddInfo.Member nesta a Scaled 6.3
```



Example 2

A masked pointer is a pointer where only a part of the “pointer value” is stored. In the following example, the element `ast->right` is a value, whose lower 17 bits are the part of an address. The target address is calculated using the lower 17 bits and adding `0x22200000` as base address. The pointer in this example is declared using the command:

```
sYmbol.AddInfo.Member strttype1 right MaskedPointer 0x0001FFFF 0x22200000
```



Example 3

C Code:

```
struct example{ uint32 mode : 2; } instance;  
enum values { on = 0, off, flicker };
```

PRACTICE script:

```
; <class_name> <member> ENUM <item_value> <item_name>  
sYmbol.AddInfo.Member example mode ENUM 0 "on"  
sYmbol.AddInfo.Member example mode ENUM 1 "off"  
sYmbol.AddInfo.Member example mode ENUM 2 "flicker"  
  
sYmbol.CREATE.MACRO on 0  
sYmbol.CREATE.MACRO off 1  
sYmbol.CREATE.MACRO flicker 2  
sYmbol.CREATE.Done  
  
Var.Set instance.mode=flicker
```

See also

■ [sYmbol.AddInfo](#)

Format:

sYmbol.AddInfo.RESet

Removes all additional information.

See also

- sYmbol.AddInfo
- sYmbol.AddInfo.Delete

sYmbol.AddInfo.Type

Add information to a data type

Format:

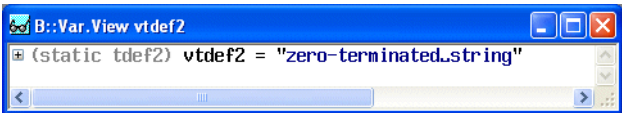
sYmbol.AddInfo.Type <type> <info> [<parameters>]

<info>:

NSTRING <bitmask>
JSTRING
ZSTRING
MostDerived <struct | class>
DESCRIPTOR <bitmask> <struct | class>
...

Add information to a specific data type. All symbolic information types are described in [sYmbol.AddInfo](#).

```
sYmbol.AddInfo.Type tdef2 ZSTRING      ; the data type tdef2 is a zero-  
                                       ; terminated string
```



The following shows examples to display Symbian OS descriptors and strings correctly in the debugger window.

```
sYmbol.AddInfo.Type "TDesC16" DESCRIPTOR 0x1xxxxxxx "TPtrC16"  
  
sYmbol.AddInfo.Type "TPtrC8" NSTRING 0x0xxxxxxx  
  
sYmbol.AddInfo.Type nextx MostDerived "TreeMFPtr"
```

See also

- sYmbol.AddInfo

Format: sYmbol.AddInfo.Var <var> <info> [<parameters>]

<info>: **Scaled** <multiplier> [<offset> [<explanation>]]
 RScaled <multiplier> [<offset> [<explanation>]]
 ZSTRING
 MaskedPointer <mask> <offset>
 MostDerived <string>
 ...

Adds type information to a variable. All symbolic information types are described in [sYmbol.AddInfo](#).

Example:

```
sYmbol.AddInfo.Var cstr1 ZSTRING           ;The contents of cstr1 is a zero-
                                           ;terminated string

sYmbol.AddInfo.List                        ;Display definition list
```



See also

- [sYmbol.AddInfo](#)

Format: **sYmbol.AutoLOAD**.<sub_cmd>

The command **sYmbol.AutoLOAD** allows to automate the loading of symbol files. This is helpful if a boot loader or an RTOS downloads code to the target. To debug this downloaded code loading of the appropriate symbol information is required.

The **sYmbol.AutoLOAD** command maintains a list for automatic loading of symbol information. This list contains:

- A list of address ranges
- For each address range a component name and an appropriate load command

Whenever the user wants to display an address within a specified address range, and TRACE32 also needs symbol information for the display, the appropriate load command is automatically started.

See also

- | | |
|--|--|
| ■ sYmbol.AutoLOAD.CHECK | ■ sYmbol.AutoLOAD.CHECKCoMmanD |
| ■ sYmbol.AutoLOAD.CHECKDLL | ■ sYmbol.AutoLOAD.CHECKEPOC |
| ■ sYmbol.AutoLOAD.CHECKLINUX | ■ sYmbol.AutoLOAD.CHECKQNX |
| ■ sYmbol.AutoLOAD.CHECKUEFI | ■ sYmbol.AutoLOAD.CHECKWIN |
| ■ sYmbol.AutoLOAD.CHECKWINCE | ■ sYmbol.AutoLOAD.CLEAR |
| ■ sYmbol.AutoLOAD.config | ■ sYmbol.AutoLOAD.Create |
| ■ sYmbol.AutoLOAD.Delete | ■ sYmbol.AutoLOAD.List |
| ■ sYmbol.AutoLOAD.LOADEPOC | ■ sYmbol.AutoLOAD.RESet |
| ■ sYmbol.AutoLOAD.SET | ■ sYmbol.AutoLOAD.TOUCH |
| ■ sYmbol | |

Format:	sYmbol.AutoLOAD.CHECK [now [<i>/<option></i>] ON OFF ONGO]
<option>:	MACHINE <machine_magic> <machine_id> <machine_name>

A single **sYmbol.AutoLOAD.CHECK** command triggers the refresh of the Autoloader table ([sYmbol.AutoLOAD.List](#)).

now	Update the Autoloader table now.
ON	If set to ON , TRACE32 updates the autoloader table after every single step and whenever the program execution is stopped. This significantly slows down the speed of TRACE32.
OFF	If set to OFF , no automatic update of the autoloader table is done.
ONGO	TRACE32 updates the autoloader table whenever the program execution is stopped.
MACHINE	Updates the autoloader table only for the specified machine. See also “What to know about the Machine Parameters” (general_ref_t.pdf).

See also

- [sYmbol.AutoLOAD](#)
- [sYmbol.AutoLOAD.config](#)

Format:

sYmbol.AutoLOAD.CHECKCoMmanD <load_command> [/<option>]

<option>:

MACHINE <machine_magic> | <machine_id> | <machine_name>

The dynamic autoloader reads the target's component table and fills the autoloader list with the components found on the target. All necessary information, such as load addresses and [space IDs](#), are retrieved from kernel-internal information. The dynamic autoloader is activated by the command [sYmbol.AutoLOAD.CHECK](#).

<load_command>	If an address is accessed that is covered by the autoloader list, the autoloader calls <load_command> and appends parameters e.g. addresses and space ID of the component to the action. Usually, <load_command> is a call to a PRACTICE script (*.cmm) that handles the parameters and loads the symbols. Please see the example scripts in the ~/demo directory.
MACHINE	Allows to specify different autoloader scripts for different machines. See also “What to know about the Machine Parameters” (general_ref_t.pdf).

Example:

```
sYmbol.AutoLOAD.CHECKCoMmanD "DO autoload"

sYmbol.AutoLOAD.CHECK

sYmbol.AutoLOAD.List
```

This command needs an OS Awareness configured for the OS running on the target. Please see the [OS Awareness Manuals](#) (rtos_<os>.pdf) for further information.

NOTE:

The dynamic autoloader covers only components that are already started. Components that are not in the current process/library table are not covered.

See also

- sYmbol.AutoLOAD
- sYmbol.AutoLOAD.config
- sYmbol.AutoLOAD.CHECKCMD()

Format: sYmbol.AutoLOAD.CHECKDLL [<i><address></i>] [<i><load_command></i>]
--

This command can only be used with Texas Instruments DSPs.

If the symbol **__DLModules** is not available, please specify the *<address>* for the automatic DLL file loader. If no *<load_command>* is specified **DO autoload** is used. The automatic DLL file loader is activated by the command **sYmbol.AutoLOAD.CHECK**.

Please refer also to the examples in `~/demo/c5000/etc/dll/`

See also

■ [sYmbol.AutoLOAD](#)

■ [sYmbol.AutoLOAD.config](#)

Format: sYmbol.AutoLOAD.CHECKEPOC <load_command>

The dynamic autoloader reads the target's process table and fills the autoloader list with the modules found on the target. All necessary information, such as load addresses and space IDs, are retrieved from kernel-internal information. The dynamic autoloader also covers dynamically loaded modules. The dynamic autoloader is activated by the command **sYmbol.AutoLOAD.CHECK**.

If an address is accessed that is covered by the autoloader list, the autoloader calls <load_command> and appends the load addresses and the space ID of the module to the action. Usually, <load_command> is a call to a PRACTICE script (*.cmm) that handles the parameters and loads the symbols. Please see the example scripts in the ~/demo directory.

Example:

```
sYmbol.AutoLOAD.CHECKEPOC "DO autoload.cmm"
```

NOTE:

- The dynamic autoloader covers only modules that are already started. Modules that are not in the current process/library table are not covered.
- If a process symbol file is loaded, the dynamic autoloader adds the space ID, which may be used to load the symbols to the appropriate space.

See also

- [sYmbol.AutoLOAD](#)
- [sYmbol.AutoLOAD.config](#)

sYmbol.AutoLOAD.CHECKLINUX Configure autoloader for Linux debugging

Format: sYmbol.AutoLOAD.CHECKLINUX <action> (deprecated)

This command is deprecated. Use **sYmbol.AutoLOAD.CHECKCoMmand** instead.

See also

- [sYmbol.AutoLOAD](#)
- [sYmbol.AutoLOAD.config](#)

Format:

sYmbol.AutoLOAD.CHECKQNX <action> (deprecated)

This command is deprecated. Use [sYmbol.AutoLOAD.CHECKCoMmand](#) instead.

See also

- [sYmbol.AutoLOAD](#)

sYmbol.AutoLOAD.CHECKUEFI

Configure autoloader for UEFI debugging

Format:

sYmbol.AutoLOAD.CHECKUEFI <load_command>

The UEFI code is provided by the boot FLASH, but debugging becomes more comfortable when debug symbols are available. TRACE32 uses the so-called [Autoloader](#) to realize the automatic loading of debug symbols whenever they are required.

The command **sYmbol.AutoLOAD.CHECKUEFI** specifies the command that is automatically used by the Autoloader to load the symbol information. Usually a script called `autoload.cmm` provided by Lauterbach is used.

The command **sYmbol.AutoLOAD.CHECKUEFI** implicitly also defines the parameters that TRACE32 uses internally for the Autoloader (see screenshot below).

When the Autoloader is configured, the command [sYmbol.AutoLOAD.CHECK](#) can be used to scan the UEFI module table and to activate the Autoloader. The command [sYmbol.AutoLOAD.List](#) allows to inspect the scanned module information.

Since the UEFI module table is updated by UEFI, a re-scan might be necessary.

B:\sYmbol\AutoLoad.List					
address	name	dyn	load	cmd	
N:FFFFD0120--FFFD60BF	PeiCore	✓	✓	do ~~/demo/x64/bootloader/uefi/bldk/autoload	"PeiCore" 0xFFFFD00E8 0x1 0xFFFFD0120 0x0
N:FFFFD63A0--FFFD87BF	CpuPeim	✓		do ~~/demo/x64/bootloader/uefi/bldk/autoload	"CpuPeim" 0xFFFFD6368 0x1 0xFFFFD63A0 0x0
N:FFFFD8AA0--FFFD9CFF	FaultTolerantWritePei	✓		do ~~/demo/x64/bootloader/uefi/bldk/autoload	"FaultTolerantWritePei" 0xFFFFD8A68 0x1 0xFFFFD8AA0 0x0
N:FFFFDA020--FFFD8CFF	PeiVariableAuth	✓		do ~~/demo/x64/bootloader/uefi/bldk/autoload	"PeiVariableAuth" 0xFFFFD9FE8 0x1 0xFFFFDA020 0x0
N:FFFFDC020--FFFD8BFF	PlatformPeim	✓		do ~~/demo/x64/bootloader/uefi/bldk/autoload	"PlatformPeim" 0xFFFFD8FE8 0x1 0xFFFFDC020 0x0
N:FFFFDEEA0--FFFE069F	SecUma	✓		do ~~/demo/x64/bootloader/uefi/bldk/autoload	"SecUma" 0xFFFFDEE68 0x1 0xFFFFDEEA0 0x0
N:FFFFE09A0--FFFE419F	PchEarlyInitPeim	✓		do ~~/demo/x64/bootloader/uefi/bldk/autoload	"PchEarlyInitPeim" 0xFFFFE0968 0x1 0xFFFFE09A0 0x0
N:FFFFE44A0--FFFE287F	MemoryInit	✓		do ~~/demo/x64/bootloader/uefi/bldk/autoload	"MemoryInit" 0xFFFFE4468 0x1 0xFFFFE44A0 0x0
N:FFFFF2C60--FFFF573F	PcdPeim	✓		do ~~/demo/x64/bootloader/uefi/bldk/autoload	"PcdPeim" 0xFFFFF2B68 0x1 0xFFFFF2C60 0x0
N:FFFFFEA88--FFFFFC7F	SecCore	✓	✓	do ~~/demo/x64/bootloader/uefi/bldk/autoload	"SecCore" 0xFFFFFEA98 0x1 0xFFFFFEA88 0x0

AutoLoader parameter list

```
sYmbol.AutoLOAD.CHECKUEFI \
    "DO ~~/demo/x86/bootloader/uefi/h2o/autoload.cmm"

sYmbol.AutoLOAD.CHECK

sYmbol.AutoLOAD.List
```

See also

- [sYmbol.AutoLOAD](#)
- [sYmbol.AutoLOAD.config](#)

sYmbol.AutoLOAD.CHECKWIN

Configure autoloader

Format: sYmbol.AutoLOAD.CHECKWIN <action> (deprecated)

This command is deprecated. Use [sYmbol.AutoLOAD.CHECKCoMmanD](#) instead.

See also

- [sYmbol.AutoLOAD](#)

sYmbol.AutoLOAD.CHECKWINCE

Configure autoloader

Format: sYmbol.AutoLOAD.CHECKWINCE <action> (deprecated)

This command is deprecated. Use [sYmbol.AutoLOAD.CHECKCoMmanD](#) instead.

See also

- [sYmbol.AutoLOAD](#)

Format:

sYmbol.AutoLOAD.CLEAR <address> | <component_name> [/<option>]

<option>:

MACHINE <machine_magic> | <machine_id> | <machine_name>

Removes symbol information for the specified <address> or <component_name>.

MACHINE	Removes symbol information only for the specified machine. See also “What to know about the Machine Parameters” (general_ref_t.pdf).
---------	---

Examples:

```
sYmbol.AutoLOAD.CLEAR C:0x5009B420  
  
sYmbol.AutoLOAD.CLEAR "*trkengine*"
```

See also

- [sYmbol.AutoLOAD](#)

Format:

sYmbol.AutoLOAD.config

Opens a configuration dialog for the symbol autoloader.

See also

- [sYmbol.AutoLOAD](#)
 - [sYmbol.AutoLOAD.CHECKCoMmanD](#)
 - [sYmbol.AutoLOAD.CHECKEPOC](#)
 - [sYmbol.AutoLOAD.CHECKUEFI](#)
 - [sYmbol.AutoLOAD.CONFIG\(\)](#)
- [sYmbol.AutoLOAD.CHECK](#)
 - [sYmbol.AutoLOAD.CHECKDLL](#)
 - [sYmbol.AutoLOAD.CHECKLINUX](#)
 - [sYmbol.AutoLOAD.List](#)

Format:

sYmbol.AutoLOAD.Create <range> <component_name> <load_command>

Specify an entry for the Autoloader table. The complete Autoloader table can be displayed with the [sYmbol.AutoLOAD.List](#) command.

- <component_name>
- Is the TRACE32 internal name for the symbol file. Please use the command [sYmbol.List.Program](#) to get this name.
- <load_command>
- Can be either a command of the [Data.LOAD](#) command group or a PRACTICE script (*.cmm).

Example:

```
sYmbol.AutoLOAD.Create 0x1000--0x2fff "thumble" \  
                                "Data.LOAD.Elf thumble.axf /NoClear /NoCODE"  
  
sYmbol.AutoLOAD.Create 0x10000--0x1ffff "can" "DO Load_CAN"
```

See also

[sYmbol.AutoLOAD](#)

Format:

sYmbol.AutoLOAD.Delete [<name> | <address>]

Deletes entries from the autoloader table. Without parameters, this commands clears the entire autoloader table.

Example:

```
sYmbol.AutoLOAD.Delete "init"
```

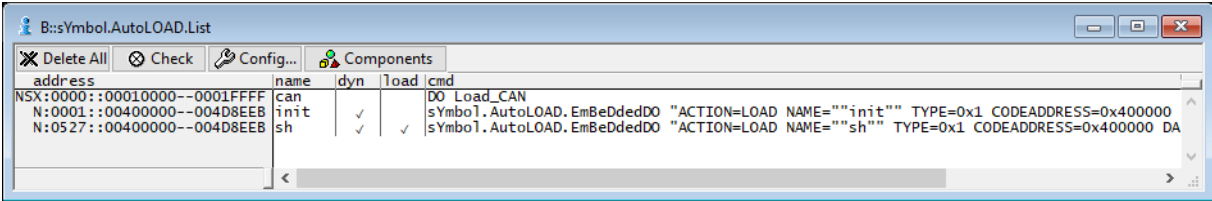
See also

[sYmbol.AutoLOAD](#)

Format:

sYmbol.AutoLOAD.List

Lists the Autoloader table with the related information for each entry.



Column	description
dyn	If selected, indicates that the autoloader entry is dynamic, e.g. created by an OS Awareness. Otherwise, if the field is not selected, it means that the autoloader entry is static, i.e. it was created manually using sYmbol.AutoLOAD.Create .
load	Selected once one of the sYmbol.AutoLOAD.SET or sYmbol.AutoLOAD.TOUCH commands has been executed for the autoloader entry.
cmd	Shows the <code><load_command></code> . If an address is accessed that is covered by the autoloader list, the autoloader calls <code><load_command></code> and appends parameters e.g. addresses and space ID of the component to the action. Usually, <code><load_command></code> is a call to a PRACTICE script (*.cmm) that handles the parameters and loads the symbols. Please see the example scripts in the <code>~/demo</code> directory.

Right-clicking a line shows additional submenu:

- **Touch:** same as executing [sYmbol.AutoLOAD.TOUCH](#) for the specified autoloader entry.
- **Set:** same as executing [sYmbol.AutoLOAD.SET](#) for the specified autoloader entry.
- **Clear:** same as executing [sYmbol.AutoLOAD.CLEAR](#) for the specified autoloader entry.

See also

- [sYmbol.AutoLOAD](#)
- [sYmbol.AutoLOAD.config](#)

Format:

sYmbol.AutoLOAD.LOADEPOC <log_file> <load_command> [/<option>]

<option>:

StripPATH | CutPATH | LowerPATH
StripPART <number> | <string>

When generating a Symbian OS ROM image (e.g. with “buildrom”), the builder generates a log file as well (usually “rombuild.log”). This log file contains the section addresses of all modules included in the image. The static autoloader reads this log file and fills the autoloader list with the modules found in the log file with it’s appropriate load addresses.

If an address is accessed that is covered by the autoloader list, the autoloader calls <load_command> and appends the load addresses of the module to the action. Usually, <load_command> is a call to a PRACTICE script that handles the parameters and loads the symbols. Please see the example scripts in the ~/demo directory.

NOTE:

- The static autoloader addresses only modules that are linked into the ROM image. Modules loaded to the target dynamically are not covered.
- The log file does not include the process ID that a process will get when started. Thus, the static autoloader loads the symbols of a process to space ID zero.

Example:

```
sYmbol.AutoLOAD.LOADEPOC "la_001.techview.log" "DO autoloader.cmm"
```

B::sYmbol.AutoLoad.List				
address	to	name	dwn	load cmd
C:50011F9C-5005CFA7		ekern.exe		DO autoloader"\\bin\\TechView\\epoc32\\release\\MC0T\\udeb\\ekern.exe" 0x50012000 0x5005C4D0 0x00000000
C:50050000-5009B303		user.dll		DO autoloader"\\bin\\TechView\\epoc32\\release\\MC0T\\udeb\\user.dll" 0x50050074 0x0 0x100000
C:5009B420-5009CD73		halla.dll		DO autoloader"\\bin\\TechView\\epoc32\\release\\MC0T\\udeb\\halla.dll" 0x5009B404 0x0 0x100000
C:5009CDE0-5009CF5F		gdbseal-full.dll		DO autoloader"\\bin\\TechView\\epoc32\\release\\THUMB\\udeb\\gdbseal-full.dll" 0x5009CE44 0x0 0x100000
C:5009CFD0-500A6563		gdbseeng.dll		DO autoloader"\\bin\\TechView\\epoc32\\release\\THUMB\\udeb\\gdbseeng.dll" 0x5009D034 0x0 0x100000
C:500A65D0-500A6F53		gdbstub.exe		DO autoloader"\\bin\\TechView\\epoc32\\release\\THUMB\\udeb\\gdbstub.exe" 0x500A6634 0x500A6F90 0x40
C:500A6FC0-500A6F4B		trkengine.dll		DO autoloader"\\bin\\TechView\\epoc32\\release\\THUMB\\udeb\\trkengine.dll" 0x500A6024 0x0 0x100000
C:500A6F00-500A6FC7		metrotrk.exe		DO autoloader"\\bin\\TechView\\epoc32\\release\\THUMB\\udeb\\metrotrk.exe" 0x500A6114 0x500A6FC14 0x4

See also

- sYmbol.AutoLOAD

Format: **sYmbol.AutoLOAD.RESet**

Clears the autoloader table.

See also

■ [sYmbol.AutoLOAD](#)

sYmbol.AutoLOAD.SET

Mark symbol information manually as loaded

Format: **sYmbol.AutoLOAD.SET** <name> | <address> | <addressrange>

The command **sYmbol.AutoLOAD.Set** allows to manually mark symbol information as loaded. This is helpful to suppress an error message when no symbol information was generated for a specified address or component range.

Example:

```
sYmbol.AutoLOAD.SET 0x5015E030
```

```
sYmbol.AutoLOAD.SET "*efsrv"
```

See also

■ [sYmbol.AutoLOAD](#)

Format:	sYmbol.AutoLOAD.TOUCH <component_name> <addressrange> <address> [/<option>]
<option>:	MACHINE <machine_magic> <machine_id> <machine_name>

Initiates the loading of symbol information as defined by the [sYmbol.AutoLOAD.CHECKCoMmanD](#).

<component_name>	Wildcards possible.
<address>	Any address within the address range used by a component.
<addressrange>	Touches all components mapped within the given address, e.g. libraries of a single process.
MACHINE	Touches only the component of the specified machine. See also “What to know about the Machine Parameters” (general_ref_t.pdf).

Examples:

```
sYmbol.AutoLOAD.TOUCH  "*efsrv*"
sYmbol.AutoLOAD.TOUCH  0x50011f9c
```

See also

- [sYmbol.AutoLOAD](#)

See also

- [sYmbol.Browse.Class](#)[sYmbol.Browse.Enum](#)[sYmbol.Browse.Function](#)[sYmbol.Browse.Module](#)[sYmbol.Browse.MVar](#)[sYmbol.Browse.name](#)[sYmbol.Browse.SFunction](#)[sYmbol.Browse.SModule](#)[sYmbol.Browse.SOURCE](#)[sYmbol.Browse.Struct](#)[sYmbol.Browse.sYmbol](#)[sYmbol.Browse.Type](#)[sYmbol.Browse.TypeDef](#)[sYmbol.Browse.Union](#)[sYmbol.Browse.Var](#)[sYmbol](#)[SETUP.sYmbol](#)[sYmbol.MATCHES\(\)](#)
- ▲ 'Release Information' in 'Legacy Release History'

sYmbol.Browse.Class

Browse classes

Format: **sYmbol.Browse.Class** [*<name>*]

Lets you browse through the list of classes that have been loaded to the internal TRACE32 symbol database with **Data.LOAD**.

See also

- [sYmbol.Browse](#)[sYmbol.Browse.name](#)

sYmbol.Browse.Enum

Browse enumeration types

Format: **sYmbol.Browse.Enum** [*<name>*]

Lets you browse through the list of enumeration types that have been loaded to the internal TRACE32 symbol database with **Data.LOAD**.

See also

- [sYmbol.Browse](#)[sYmbol.Browse.name](#)

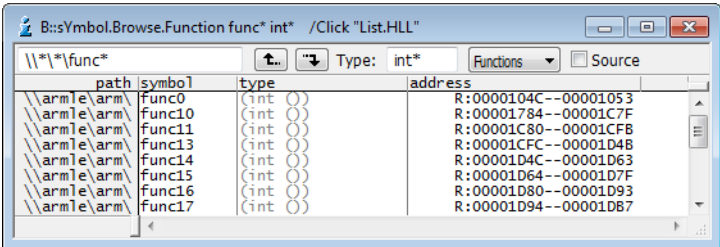
Format:

sYmbol.Browse.Function [<name_pattern> [<type_pattern>]] [/<option>]

<option>:

Click <cmd>
Delete

Lets you browse through the list of functions that have been loaded to the internal TRACE32 symbol database with [Data.LOAD](#).



Alternatively, you can browse through the function list by selecting **Functions** from the drop-down list in the [sYmbol.Browse.sYmbol](#) window.

For a description of the command line arguments, see [sYmbol.Browse.sYmbol](#).

In addition, the browser command is automatically executed when:

- A symbol is entered that ends with a wildcard or
- The symbol is not unique and the [sYmbol.MATCH](#) command is set to **Choose**.

Example:

```
List.auto func*      ;First, the symbol browser opens, where you can select
                      ;the desired function with a double-click.

                      ;Then the selected function is displayed in the
                      ;List.auto window.
```

See also

- sYmbol.Browse
- sYmbol.Browse.name

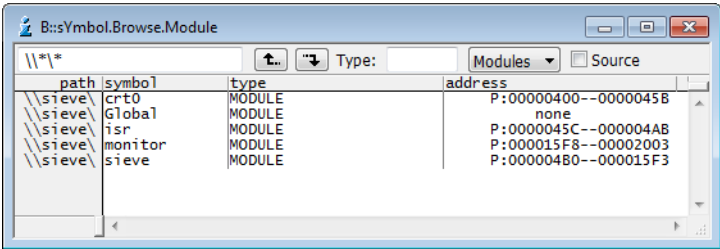
Format:

sYmbol.Browse.Module [<name_pattern> [<type_pattern>]] [/<option>]

<option>:

Click <cmd>
Delete

Lets you browse through the list of modules that have been loaded to the internal TRACE32 symbol database with [Data.LOAD](#).



Alternatively, you can browse through the module list by selecting **Modules** from the drop-down list in the [sYmbol.Browse.sYmbol](#) window.

<name_pattern>, etc.	For a description of the command line arguments, see sYmbol.Browse.sYmbol .
-------------------------	---

See also

- sYmbol.Browse
- sYmbol.Browse.name

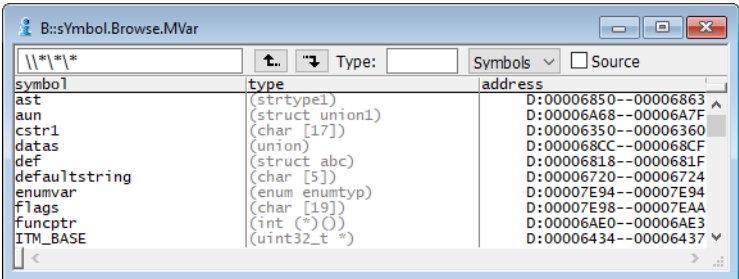
Format:

sYmbol.Browse.MVar [<name_pattern> [<type_pattern>]] [/<option>]

<option>:

Click <cmd>
Delete

Lets you browse through the list of module variables that have been loaded to the internal TRACE32 symbol database with **Data.LOAD**. Functions and local variables are not displayed in the **sYmbol.Browse.MVar** window.



<name_pattern>, etc.

For a description of the command line arguments, see [sYmbol.Browse.sYmbol](#).

See also

- [sYmbol.Browse](#)
- [sYmbol.Browse.name](#)

sYmbol.Browse.name

Browse symbols (flat)

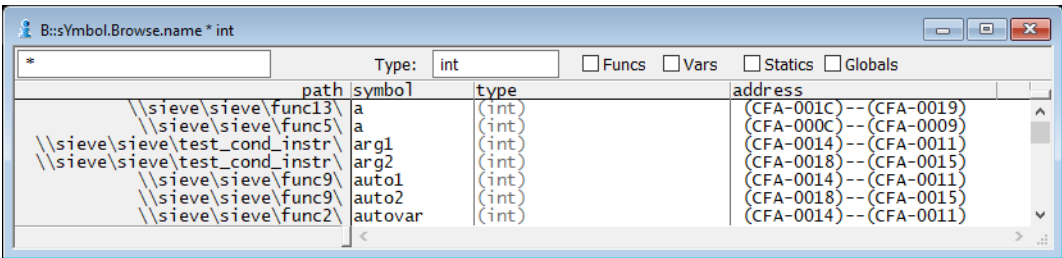
Format:

sYmbol.Browse.name [<name_pattern> [<type_pattern>]] [/<option>]

<option>:

Click <cmd>
Delete

Display symbols sorted alphabetically. This command is an alias for **sYmbol.name**.



Lower and upper case characters are not distinguished. For mangled C++ symbols the search order is based on the function signatures. A complex search function is implemented to find symbol name very fast, if the complete name will be not known. The search patterns are:

- '*' Matches any string, empty strings too.
- '?' Matches any character, but not an empty character.
- ''' Can be used to input special characters like '*' or '?'

Click	Defines a command, that can be executed by a short click with the left mouse button. The characters '*' or '?' can be used as placeholder for the complete name of the symbol. Using the '*' will force the command to be executed without further interaction and without leaving the window. The character '?' will cause the cursor to leave the window and build a command line, that can be modified before entering.
Delete	Deletes the window after the selection has been made.

Refer to [sYmbol.name](#) for more information.

See also

- [sYmbol.Browse](#)[sYmbol.Browse.Class](#)[sYmbol.Browse.Enum](#)[sYmbol.Browse.Function](#)[sYmbol.Browse.Module](#)[sYmbol.Browse.MVar](#)[sYmbol.Browse.SFunction](#)[sYmbol.Browse.SModule](#)[sYmbol.Browse.SOURCE](#)[sYmbol.Browse.Struct](#)[sYmbol.Browse.sYmbol](#)[sYmbol.Browse.Type](#)[sYmbol.Browse.TypeDef](#)[sYmbol.Browse.Union](#)[sYmbol.Browse.Var](#)[sYmbol.INFO](#)
- ▲ 'The Symbol Database' in 'Training Source Level Debugging'

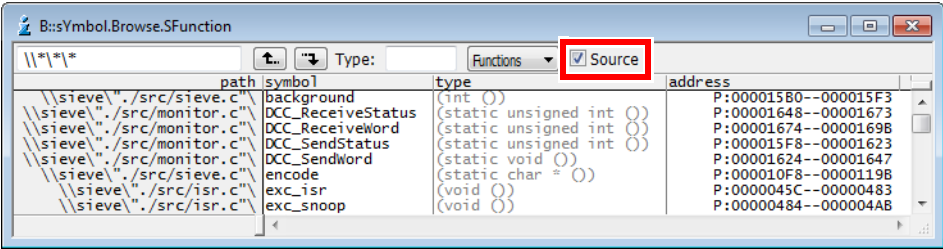
sYmbol.Browse.SFunction

Browse functions

Format:	sYmbol.Browse.SFunction [<i><name_pattern></i>] [<i><type_pattern></i>] [<i>/<option></i>]
<i><option></i> :	Click <i><cmd></i> Delete

Lets you browse through the list of functions that have been loaded to the internal TRACE32 symbol database with the [Data.LOAD](#) command.

The **S** in **SFunction** selects the **Source** check box. With the **Source** check box selected, the names of the source files are displayed instead of the module names.



Alternatively, you can browse through the function list by selecting **Functions** from the drop-down list in the **sYmbol.Browse.sYmbol** window.

<name_pattern>, etc.	For a description of the command line arguments, see sYmbol.Browse.sYmbol .
-------------------------	--

See also

- [sYmbol.Browse](#)
- [sYmbol.Browse.name](#)

Format:

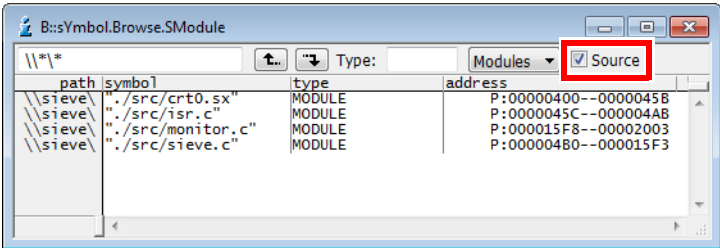
sYmbol.Browse.SModule [*<name_pattern>*] [*<type_pattern>*] [*/<option>*]

<option>:

Click *<cmd>*
Delete

Lets you browse through the modules that have been loaded to the internal TRACE32 symbol database with [Data.LOAD](#).

The **S** in **SModule** selects the **Source** check box. With the **Source** check box selected, the names of the source files are displayed instead of the module names.



Alternatively, you can browse through the module list by selecting **Modules** from the drop-down list in the [sYmbol.Browse.sYmbol](#) window.

<i><name_pattern></i> , etc.	For a description of the command line arguments, see sYmbol.Browse.sYmbol .
---------------------------------------	---

See also

- sYmbol.Browse
- sYmbol.Browse.name

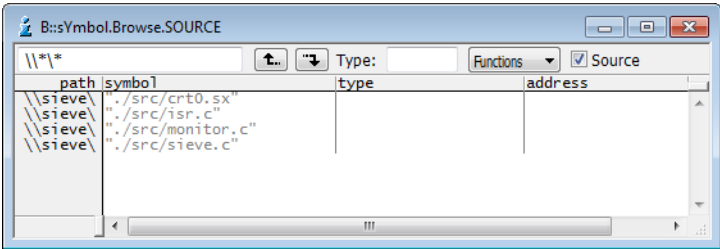
Format:

sYmbol.Browse.SOURCE [*<name_pattern>*] [*<type_pattern>*]] [*/<option>*]

<option>:

Click *<cmd>*
Delete

Lets you browse through the list of source files that have been loaded to the internal TRACE32 symbol database with the **Data.LOAD** command.



<i><name_pattern></i> , etc.	For a description of the command line arguments, see sYmbol.Browse.sYmbol .
---------------------------------------	---

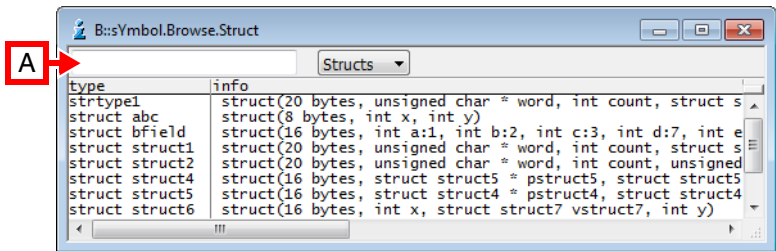
See also

- [sYmbol.Browse](#)
- [sYmbol.Browse.name](#)

Format:

sYmbol.Browse.Struct [<name>]

Lets you browse through the containers for different variable types that have been loaded to the internal TRACE32 symbol database with the **Data.LOAD** command.



A You can use the wildcards ‘*’ and ‘?’ to filter the <name> list.

See also

- [sYmbol.Browse](#)
- [sYmbol.Browse.name](#)

Format:

sYmbol.Browse.sYmbol [<name_pattern> [<type_pattern>]] [/<option>]

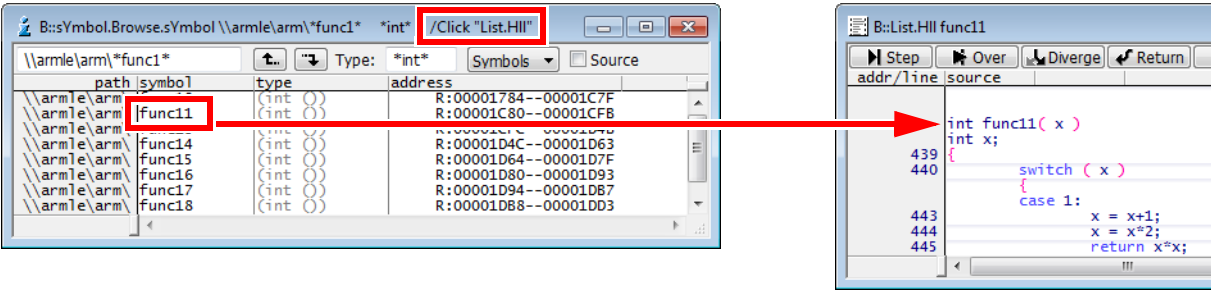
<option>:

Click <cmd>
Delete

Lets you browse through the symbol and debug information that has been loaded to the internal TRACE32 symbol database with the **Data.LOAD** command.

Entering an ASCII string searches for symbols beginning with this string. The **Up** and **Down** buttons can be used to navigate up and down in the symbol tree.

Double-clicking a symbol in a **sYmbol.Browse.sYmbol** window opens the symbol in its default window. Alternatively, you can customize TRACE32's response to the double-click by using the **Click <cmd>** option. An example is shown in the screenshots below.



<name_pattern>	Path and symbol name, or just the symbol name.
<type_pattern>	HLL types.
The patterns support the wildcards * and ? for filtering the display in the sYmbol.Browse.* windows.	
Click <cmd>	Command <cmd> will be executed after you double-click a symbol entry. Without the Click option, the double-clicked symbol opens in its default window.
Delete	Dialog will be closed after you double-click a symbol entry.

See also

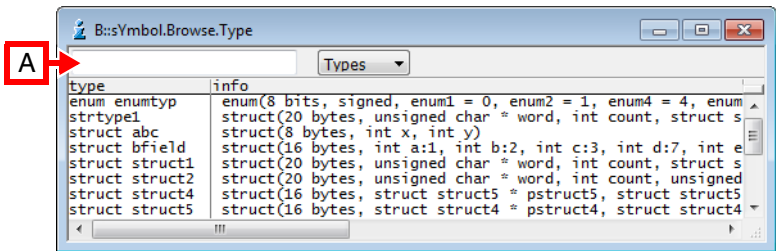
- sYmbol.Browse
- sYmbol.Browse.name
- sYmbol.STATE
- ▲

'The Symbol Database' in 'Training Source Level Debugging'

Format:

sYmbol.Browse.Type [<name>]

Lets you browse through the list of HLL types that have been loaded to the internal TRACE32 symbol database with the **Data.LOAD** command.



A You can use the wildcards * and ? to filter the list. See also <name> below.

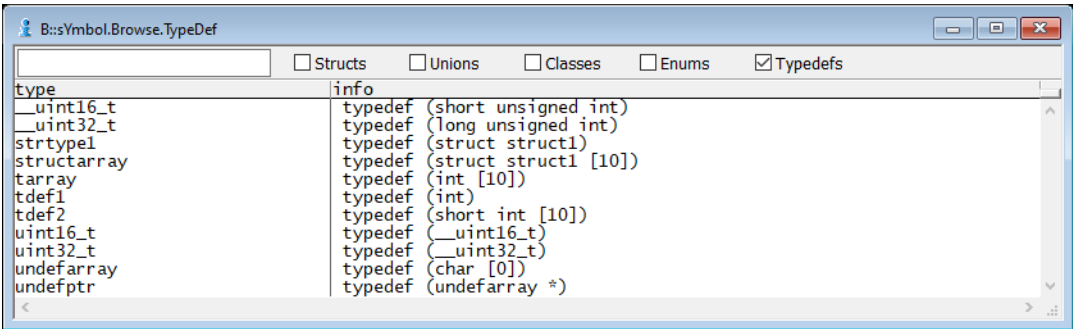
<name>	Name of an HLL type. HLL types include int, char, unsigned int, int*, struct <name>, enum <name>, etc.
--------	--

See also

- [sYmbol.Browse](#)
- [sYmbol.Browse.name](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format: sYmbol.Browse.TypeDef [<name>]

Lets you browse through the list of type definitions that have been loaded to the internal TRACE32 symbol database with [Data.LOAD](#).

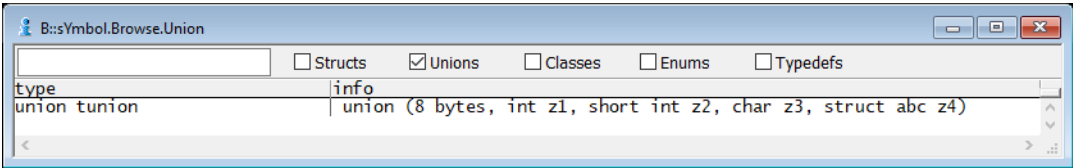


See also

[sYmbol.Browse](#)[sYmbol.Browse.name](#)

Format: sYmbol.Browse.Union [<name>]

Lets you browse through the list of unions that have been loaded to the internal TRACE32 symbol database with [Data.LOAD](#).



See also

[sYmbol.Browse](#)[sYmbol.Browse.name](#)

Format:

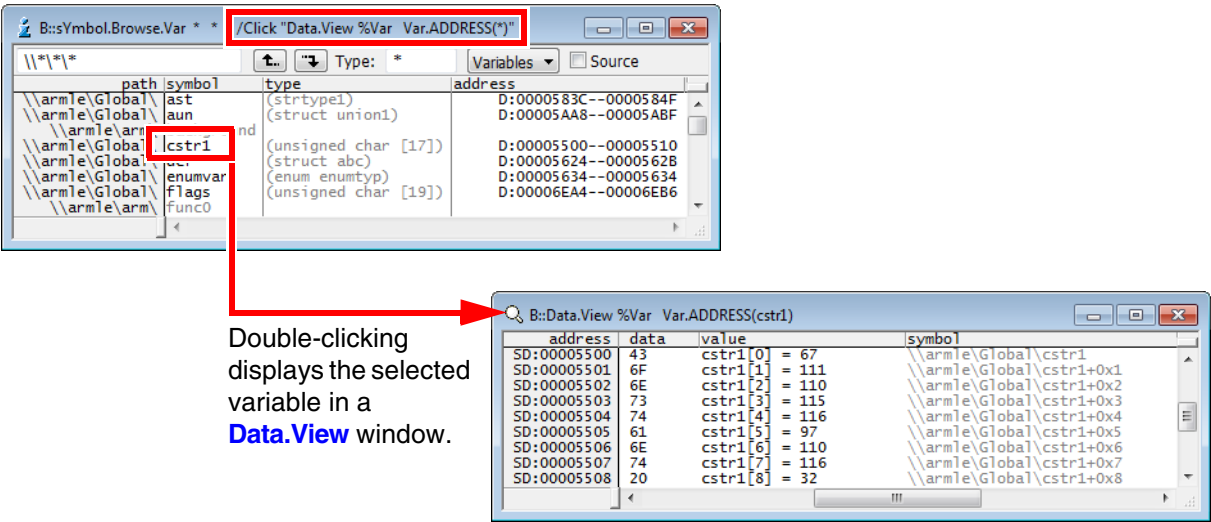
sYmbol.Browse.Var[<name_pattern> [<type_pattern>]] [/<option>]

<option>:

Click <cmd>
Delete

Lets you browse through the list of variables that have been loaded to the internal TRACE32 symbol database. Alternatively, you can browse through the variables list by selecting **Variables** from the drop-down list in the [sYmbol.Browse.sYmbol](#) window.

Double-clicking a symbol in a [sYmbol.Browse.sYmbol](#) window opens the symbol in its default window. Alternatively, you can customize TRACE32's response to the double-click by using the **Click <cmd>** option. An example is shown in the figure below.



For a description of the command line arguments and another example for the **Click <cmd>** option, see [sYmbol.Browse.sYmbol](#).

In addition, the browser command is automatically executed when:

- A symbol that ends with a wildcard is entered at the TRACE32 command line or
- The symbol is not unique and the [sYmbol.MATCH](#) command is set to **Choose**.

Var.Watch vd*

;First, the symbol browser opens, where you can select
the desired variable with a double-click.

;Then the selected variable is displayed in the
Var.Watch window.

See also

- [sYmbol.Browse](#)
- [sYmbol.Browse.name](#)

Format:

sYmbol.CASE [ON | OFF]

If the option will be set (default), there is a differentiation between small and capital letters.

See also

- [sYmbol](#)
- [Var](#)

Format:

sYmbol.CHECK

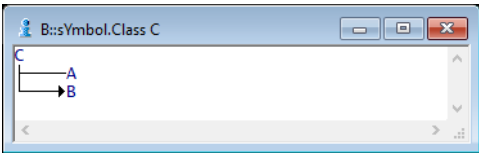
Checks the internal symbol database for errors. Can be used when the compiler output format is questionable.

See also

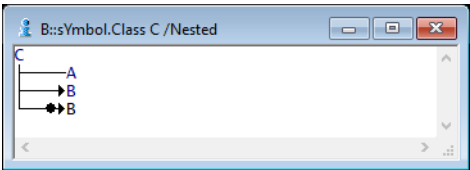
- [sYmbol](#)

Format: **sYmbol.Class** <class> [/Nested]

Displays the base classes inherited by the class as a tree. As a default the inherited classes for C++ are displayed. With the **Nested** option regular included structures and pointers to structures are displayed also. This allows the usage of this command for non C++ applications.



View the hierarchy of a specific class.



See also

- [sYmbol](#)

Format:	sYmbol.CLEANUP.<sub_cmd>
<sub_cmd>:	sYmbols CodeLiterals AsmFrames MidInstLines [/<option>] FunctionRanges DataInCode LineRanges DOUBLES AlignmentPaddings
<option>:	VM Forward Backward AlignToSymbols

sYmbols	Removes all double symbols, all double type information, empty type information and redundant symbols for the common bank address range.
CodeLiterals	Architecture specific. Tries to fix debug information about literals that really contain executable code.
AsmFrames	Removes frame information for assembler frames.
MidInstLines	Ignores HLL line information which points to odd addresses plus terminates the disassembly of code lines. For option descriptions, see below.
FunctionRanges	Fixes overlapping function end addresses.
DataInCode	Identifies and marks data inside code areas (PowerPC only).
LineRanges	Tries to fix bad address range information of source lines.
DOUBLES	For a command description, see sYmbol.CLEANUP.DOUBLES .
Alignement-Paddings	Detects memory address ranges at the end of functions that were inserted due to memory alignment and removes them from the function address ranges.

Options only for command sYmbol.CLEANUP.MidInstLines	
VM	Fixes code in VM (VM access class).
Forward, Backward	Moves questionable lines forward/backward.
AlignToSymbols	Keeps alignment of lines to symbols in code block.

See also

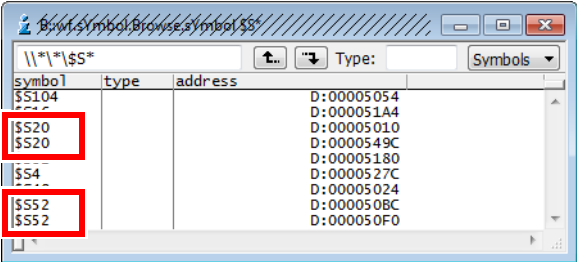
- [■ sYmbol.CLEANUP.DOUBLES](#)
- [■ sYmbol](#)
- [▲ 'Release Information' in 'Legacy Release History'](#)

Format:

sYmbol.CLEANUP.DOUBLES

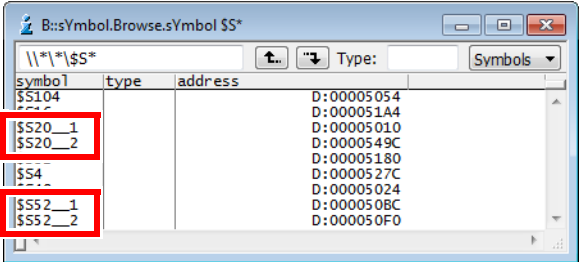
Makes symbols loaded to the TRACE32 symbol database unique by appending two underscores and a serial number to ambiguous symbols: `<ambiguous_symbol>__<serial_number>`

Ambiguous symbols



symbol	type	address
\$S104		D:00005054
\$S20		D:000051A4
\$S20		D:00005010
		D:0000549C
\$S4		D:00005180
		D:0000527C
		D:00005024
\$S52		D:000050BC
\$S52		D:000050F0

Unique symbols after using
sYmbol.CLEANUP.DOUBLES



symbol	type	address
\$S104		D:00005054
\$S20_1		D:000051A4
\$S20_2		D:00005010
		D:0000549C
\$S4		D:00005180
		D:0000527C
		D:00005024
\$S52_1		D:000050BC
\$S52_2		D:000050F0

See also

■ [sYmbol.CLEANUP](#)

Format:

sYmbol.ColorCode [ON | OFF]

Enables the source text color coding. Source color coding improves the readability of source files by using multiple colors. Enabled by default.

See also

■ [sYmbol](#)

Format:

sYmbol.ColorDef <keyword> <value>

Specify a color used for the display of keywords in your HLL code.

<value>

All color definitions can be listed with the command [sYmbol.List.ColorDef](#).

Example:

```
sYmbol.ColorDef "printf" 2          ; display all printf in green
```

See also

- [■ sYmbol](#)
- [■ sYmbol.List.ColorDef](#)
- [■ SETUP.COLOR](#)
- [▲ 'PowerView - Screen Display' in 'PowerView User's Guide'](#)

sYmbol.CREATE

Create and modify user-defined symbols

The **sYmbol.CREATE** command group allows to create new symbols or modify existing used-defined symbols. The created symbols will be made available to the debugger with the command [sYmbol.CREATE.DONE](#).

See also

- [■ sYmbol.CREATE.ATTRibute](#)
- [■ sYmbol.CREATE.Done](#)
- [■ sYmbol.CREATE.Function](#)
- [■ sYmbol.CREATE.Label](#)
- [■ sYmbol.CREATE.LocalVar](#)
- [■ sYmbol.CREATE.MACRO](#)
- [■ sYmbol.CREATE.Module](#)
- [■ sYmbol.CREATE.RESet](#)
- [■ sYmbol.CREATE.Var](#)
- [■ sYmbol](#)
- [■ sYmbol.Delete](#)
- [■ sYmbol.NEW](#)
- [▲ 'Release Information' in 'Legacy Release History'](#)

Format: sYmbol.CREATE.ATTRibute<attribute> <address>|<range>

Creates a new user-defined attribute. The new attribute only becomes visible (e.g. in the **sYmbol.List.ATTRibute** window) after calling **sYmbol.CREATE.Done**.

Example:

```
sYmbol.CREATE.ATTRibute DATA 0x1000--0x1fff
sYmbol.CREATE.Done
```

See also

■

[sYmbol.CREATE](#)

▲

['Release Information' in 'Legacy Release History'](#)

Format: sYmbol.CREATE.Done

Finishes the creation of new symbols and makes them available to the debugger. This command is only required for **sYmbol.CREATE** commands. **sYmbol.NEW** commands already contain this functionality, but will execute slower.

Example:

```
sYmbol.CREATE.Label mylab1 0x1000      ; creates "mylab1" at 1000
sYmbol.CREATE.Label mylab2 0x1010      ; creates "mylab2" at 1010
sYmbol.CREATE.Done                     ; make labels available to program
```

See also

■

[sYmbol.CREATE](#)

Format: **sYmbol.CREATE.Function** <name> <addressrange>

Creates symbol information for a new function. The function has no parameters or local variables. It can only be used to define a range for a piece of code (e.g. for performance analysis).

Note that functions created with **sYmbol.CREATE.Function** will only become visible (e.g. in the **sYmbol.List** window) after calling **sYmbol.CREATE.Done**.

```
; function ends before mylabel2
sYmbol.CREATE.Function myfunc mylabel1--(mylabel2-1)
sYmbol.CREATE.Done
```

This is not required for **sYmbol.NEW.Function** which immediately commits all changes to the symbol table and thus executes slower.

```
; function ends before mylabel2
sYmbol.NEW.Function myfunc mylabel1--(mylabel2-1)
```

See also

■ [sYmbol.CREATE](#)

■ [sYmbol.NEW.Function](#)

Format: **sYmbol.CREATE.Label** <name> <address>

Creates a new label. A label is a symbol without type information that refers to a single memory location.

Example 1:

```
sYmbol.CREATE.Label mylab1 0x1000      ; creates "mylab1" at 1000
sYmbol.CREATE.Label mylab2 0x1010      ; creates "mylab2" at 1010
sYmbol.CREATE.Done                     ; make labels available to program
```

Example 2:

```
sYmbol.NEW.Label mylab3 0x1020          ; "mylab3" is available immediately
```

See also

■ [sYmbol.CREATE](#)

■ [sYmbol.NEW.Label](#)

Format: **sYmbol.CREATE.LocalVar** <function> <var> <address>|<range> <type>

Creates a new local variable in a user-defined function created with [sYmbol.CREATE.Function](#) or [sYmbol.NEW.Function](#).

Note that local variables created with **sYmbol.CREATE.LocalVar** will only become visible (e.g. in the [sYmbol.List](#) window) after calling [sYmbol.CREATE.Done](#). This is not required for [sYmbol.NEW.LocalVar](#) which immediately commits all changes to the symbol table and thus executes slower.

Example:

```
sYmbol.CREATE.Function myfunc 0x1000--0x10ff
sYmbol.CREATE.LocalVar myfunc loc_var 0x2000 int
sYmbol.CREATE.Done
```

See also

■ [sYmbol.CREATE](#)

Format: **sYmbol.CREATE.MACRO** <name> <contents>

Creates a new macro. The macro can be used like a C-preprocessor macro. Parameters can be supplied in the same way.

Example:

```
; creation and usage of macro MY_NEXT(<arg>)
sYmbol.CREATE.MACRO MY_NEXT(p)  ((p)->next)
sYmbol.CREATE.Done
Var.View MY_NEXT(myvar)
```

See also

- [sYmbol.CREATE](#)
- [sYmbol.NEW.MACRO](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format: **sYmbol.CREATE.Module** <name> <range>

Creates a user-defined module.

Example:

```
sYmbol.CREATE.Module test 50000--5fffff
sYmbol.CREATE.Done
sYmbol.Browse.Module
```

See also

- [sYmbol.CREATE](#)
- [sYmbol.NEW.Module](#)

Format:

sYmbol.CREATE.RESet

Removes all symbols defined by [sYmbol.CREATE](#) or [sYmbol.NEW](#) commands.

See also

[■ sYmbol.CREATE](#)[■ sYmbol.NEW](#)

sYmbol.CREATE.Var

Create user-defined variable

Format:

sYmbol.CREATE.Var <variable_name> <address>|<range> <type>

Creates a user-defined variable.

Examples:

```
sYmbol.List.Type /Unnamed           ; list all types
sYmbol.CREATE.Var my_char 0x1000 char ; create variable
sYmbol.CREATE.Done                  ; finish creation
sYmbol.INFO my_char                 ; display all information
                                   ; about the created variable
```

```
sYmbol.List.Type /Unnamed
sYmbol.CREATE.Var my_abc D:0xa000 struct abc
sYmbol.CREATE.Done
sYmbol.INFO my_abc
```

See also

[■ sYmbol.CREATE](#)[■ sYmbol.NEW.Var](#)

Format: **sYmbol.CUTLINE** [*<length>*]

The number of source lines, displayed for one high-level line is limited. This prevents the display of large parts of source text in analyzer or data windows, if the source contains many definitions. The last lines are always displayed. Without arguments function is turned off, i.e. all lines are displayed again.

Example:

```
sYmbol.CUTLINE 3.           ; display max. 3 lines for each HLL line
```

See also

■ [sYmbol](#)

sYmbol.Delete

Delete symbols of one program

Format: **sYmbol.Delete** [*<program>*]

Deletes all symbols of one program.

Example:

```
sYmbol.Delete \\mccp       ; delete only symbols of program "mccp"
```

See also

■ [sYmbol](#) ■ [sYmbol.CREATE](#) ■ [sYmbol.NEW](#)
▲ ['Release Information' in 'Legacy Release History'](#)

Format: **sYmbol.DeleteMACRO** *<macro>*

Deletes certain macro information.

See also

■ [sYmbol](#)

▲ ['Release Information' in 'Legacy Release History'](#)

sYmbol.DeletePATtern Delete labels from symbol database using wildcards

Format: **sYmbol.DeletePATtern** *<symbol_pattern>*

Delete all labels from the symbol database that match the specified *<symbol_pattern>*. Allowed wildcards are * and ?.

Example:

```
sYmbol.DeletePATtern _d_*
```

See also

■ [sYmbol](#)

sYmbol.DEMangle

C++ demangler

Format: **sYmbol.DEMangle** [ON | OFF] [ON | OFF]

The first argument activates the demangler, the second enables the demangling of function arguments.

Example:

```
sYmbol.DEMangle OFF           ; get_branches__6ForestFP4Tree (ANSI)
                               ; @Forest@get_branches$qp4Tree (TURBO-C++)
sYmbol.DEMangle ON OFF       ; Forest::get_branches
sYmbol.DEMangle ON           ; Forest::get_branches(Tree *)
```

See also

■ [sYmbol](#)

sYmbol.DEOBFUSCATE

Deobfuscate global and static symbol

[build 145539 - DVD 09/2022]

Format: **sYmbol.DEOBFUSCATE** <file> [<module>]

This command allows to deobfuscate global and static symbols for the defined module on user request.

The input <file> contains a list of real symbol name and obfuscated symbol name separated by a blank.

All symbols that match an obfuscated name are replaced with the real symbol name.

See also

■ [sYmbol](#) ■ [sYmbol.Modify.NAMES](#)

sYmbol.DONE

Finish load of symbols

[build 133356 - DVD 09/2021]

Format: **sYmbol.DONE**

Finishes load of symbols when using [Data.LOAD MORE](#).

See also

■ [sYmbol](#)

The TRACE32 debugger receives all the necessary information about the source code via the debug information generated by the compiler. However, some commands require more details. Lauterbach provides the command line tool t32cast (see [“Application Note for t32cast”](#) (app_t32cast.pdf)) to generate these details.

t32cast analyzes the individual source code files and generates a .eca file per source code file, which is stored in the same directory. ECA stands for Extended Code Analysis.

The **sYmbol.ECA** command group allows to manage the .eca files.

ECA files are currently used by the [code coverage system](#).

See also

- | | | | |
|-------------------------------------|--------------------------------------|-----------------------------------|--------------------------------------|
| ■ sYmbol.ECA.BINary | ■ sYmbol.ECA.Delete | ■ sYmbol.ECA.Init | ■ sYmbol.ECA.List |
| ■ sYmbol.ECA.LOAD | ■ sYmbol.ECA.LOADALL | ■ sYmbol | ■ sYmbol.List.SOURCE |

See also

■ [sYmbol.ECA](#)

sYmbol.ECA.BINary.CollapseAll

Collapse all trees

[build 134348 - DVD 09/2021]

Format:

sYmbol.ECA.BINary.CollapseAll

Command to collapse all trees in an open [sYmbol.ECA.BINary.view](#) window.

See also

■ [sYmbol.ECA.BINary.ExpandAll](#) ■ [sYmbol.ECA.BINary.view](#)

sYmbol.ECA.BINary.ControlFlowMode.INSTR

Consider instrumentation

[build 157096 - DVD 2023/02]

Format:

sYmbol.ECA.BINary.ControlFlowMode.INSTR [ON | OFF]

This command is used to configure the [sYmbol.ECA.BINary.PROCESS](#) command.

ON	Consider source code instrumentation sites as source for monitoring decisions/conditions for code coverage.
OFF (default)	Ignore source code instrumentation sites as source for monitoring decisions/conditions for code coverage.

See also

■ [sYmbol.ECA.BINary.ControlFlowMode.Trace](#) ■ [sYmbol.ECA.BINary.PROCESS](#)
▲ 'Steps in Preparation for Trace Data Collection' in 'Application Note for Trace-Based Code Coverage'

Format: **sYmbol.ECA.BINary.ControlFlowMode.Trace** [ON | OFF]

This command is used to configure the **sYmbol.ECA.BINary.PROCESS** command.

ON (default)	Consider trace event of conditional branches/instructions as source for monitoring decisions/conditions for code coverage.
OFF	Ignore trace events of conditional branches/instructions as source for monitoring decisions/conditions for code coverage.

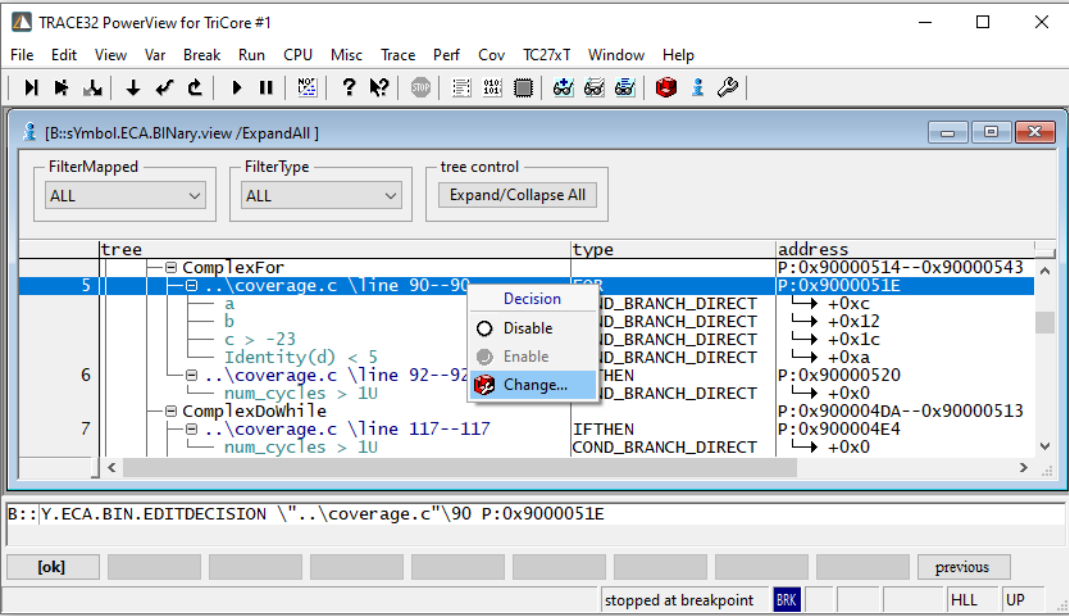
See also

- [■ sYmbol.ECA.BINary.ControlFlowMode.INSTR](#)
- [■ sYmbol.ECA.BINary.PROCESS](#)
- [▲ 'Build Process' in 'Application Note for Trace-Based Code Coverage'](#)

Format: **sYmbol.ECA.BINary.EditDecision** <source> [<index>] <address>
sYmbol.ECA.EditDecision (deprecated)

Allows editing the start address of a decision as loaded from an ECA file. This is an expert command that should only be used under the guidance of Lauterbach support.

This command can also be accessed via the **Decision** pull-down in [sYmbol.ECA.BINary.view](#) window.



Example:

```
; modify start address of the decision in source code line 90
sYmbol.ECA.BINary.EditDecision \"..\coverage.c\"90 P:0x9000051E
```

sYmbol.ECA.BINary.ExpandAll

Expand all trees

[build 134348 - DVD 09/2021]

Format: sYmbol.ECA.BINary.ExpandAll

Command to expand all trees in an open [sYmbol.ECA.BINary.view](#) window. It expands all trees.

See also

- sYmbol.ECA.BINary.CollapseAll
- sYmbol.ECA.BINary.view

Format:	sYmbol.ECA.BINary.EXPORT.GAPS [/<option>]
<option>:	CLOBber

Splits up the observability gaps per source file and place each gap file in the same directory as the corresponding ECA file. Existing gap files are preserved if no more recent data is available.

CLOBber	Allows existing files to be overwritten.
---------	--

Format:	sYmbol.ECA.BINary.EXPORT.Decisions <file> [/<option>]
<option>:	FilterMapped FilterType StripPATH

Export decision details as displayed in the [sYmbol.ECA.BINary.view](#) window to a CSV file. For documentation of the command options refer to [sYmbol.ECA.BINary.view](#).

Format:	sYmbol.ECA.BINary.EXPORT.GAPS <file>
---------	--------------------------------------

Export the observability gaps detected with the [sYmbol.ECA.BINary.PROCESS](#) command to a JSON file. The JSON file is used as input for the t32cast command line tool to control the targeted instrumentation, see [“Application Note for t32cast”](#) (app_t32cast.pdf).

- See also
- [sYmbol.ECA.BINary.PROCESS](#)

■ [sYmbol.ECA.BINary.view](#)

▲ 'Build Process' in 'Application Note for Trace-Based Code Coverage'

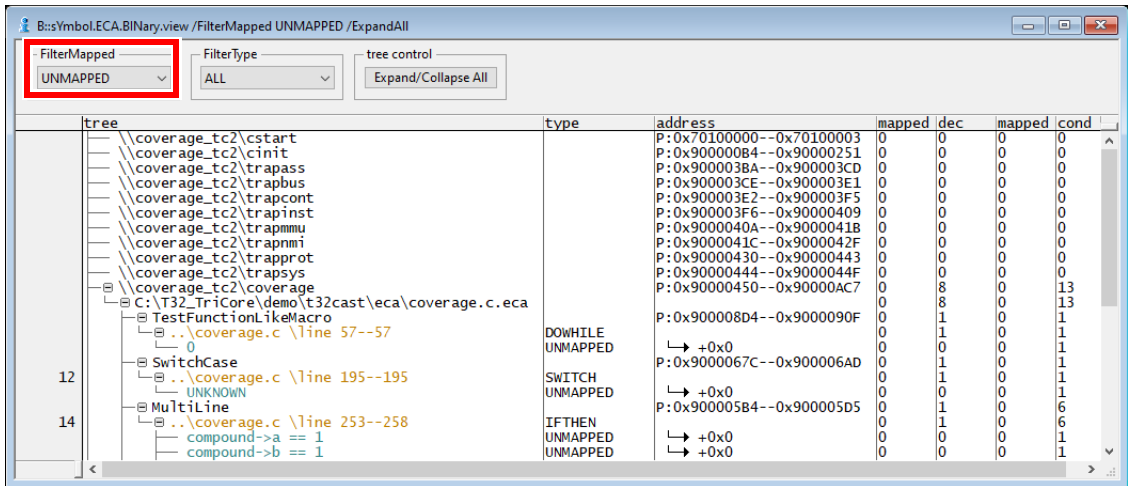
Format:

sYmbol.ECA.BINary.FilterMapped <display>

<type>:

ALL | MAPPED | UNMAPPED | NOTINBINARY

Command to filter the display by the mapping states in an open [sYmbol.ECA.BINary.view](#) window.



<display>	
ALL	Display the mapping result of all decisions (default).
MAPPED	Display only the mapped decisions. The source code lines for mapped decisions are displayed in blue.
UNMAPPED	Display only the unmapped decisions. The source code lines for unmapped decisions are displayed in orange.
NOTINBINARXY	No mapping is possible because there is no object code for this decisions. The compiler has optimized this location.

See also

■ [sYmbol.ECA.BINary.view](#)

Format:	sYmbol.ECA.BINary.FilterType <type>
<type>:	ALL IFTHEN FOR DOWHILE WHILE SWITCH ASSIGN NESTEDASSIGN TERNARY ARGUMENT

Command to filter by decision types in an open [sYmbol.ECA.BINary.view](#) window.

See also

■ [sYmbol.ECA.BINary.view](#)

Format:	sYmbol.ECA.BINary.PROCESS COVerge.StaticInfo (deprecated)
---------	--

The **sYmbol.ECA.BINary.PROCESS** command performs static analysis required for MC/DC, condition, and decision coverage.

- It preprocesses the object code to improve code coverage results for instruction set specialties such as IT blocks (Arm Thumb).
- It determines the best monitoring strategy for decisions/conditions.

In the first two steps, the analysis performs the following:

1. The number and the places of all conditional branches/instructions in the object code of the application is detected.
2. The result of the first step is matched with the decision/conditions details located in the .eca files.

The outcome is a mapping of all conditions/decisions to the object code. This is sufficient for the build process.

Build Process

The analysis outcome can be used to close [observability gaps](#) through targeted instrumentation. For the analysis, the [sYmbol.ECA.BINary.ControlFlowMode.Trace](#) command must be set ON. Decisions/conditions that cannot be monitored by conditional branches/statements are detected by the analysis and marked as unmapped. In this way, the observability gaps are identified. The gaps can be exported to a JSON file using the [sYmbol.ECA.BINary.EXPORT.GAPS](#) command. This file forms the input for the targeted source code instrumentation by the command line tool t32cast, see [“Application Note for t32cast”](#) (app_t32cast.pdf).

To detect the observability gaps, TRACE32 must have access to the full context of the application for which the code coverage analysis will be performed later. What is meant in this context is the memory layout and the symbol information. If the application context can be fully setup in a TRACE32 Instruction Set Simulator, the ISS can be used in the build process.

Code Coverage Analysis

For the code coverage analysis it must be configured which information should be considered for a complete result:

- the conditional branches/instructions in the trace recording of the program flow based on the mapping of all conditions/decisions to the object code performed in the first two steps of the static analysis (command **sYmbol.ECA.BINary.ControlFlowMode.Trace ON**).
- the source code instrumentation sites detected by further analysis steps (command **sYmbol.ECA.BINary.ControlFlowMode.INSTR ON**).
- the breakpoints.

The **sYmbol.ECA.BINary.view** command displays the result of monitoring.

Here is a sample command sequence for the build process.

```
; target setup

; load the application
Data.LOAD.ElF my_demo

; load the OS Awareness
TASK.CONFIG ~/demo/arm/kernel/freertos/freertos.t32

; detects memory address ranges at the end of functions that were
; inserted due to memory alignment and removes them from the function
; address ranges.
sYmbol.CLEANUP.AlignmentPaddings

; consider conditional branches/instructions in the object code for the
; static analysis
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; perform the static analysis for condition, decision and MC/DC coverage
sYmbol.ECA.BINary.PROCESS

; display the observability gaps
sYmbol.ECA.BINary.view /FilterMapped UNMAPPED /ExpandAll

; export the observability gap details
sYmbol.ECA.BINary.EXPORT.GAPS mygaps.json
```

Here is a sample command sequence for the code coverage analysis.

```
; target setup

; load the application
Data.LOAD.ElF my_demo

; load the OS Awareness
TASK.CONFIG ~/demo/arm/kernel/freertos/freertos.t32

; detects memory address ranges at the end of functions that were
; inserted due to memory alignment and removes them from the function
; address ranges.
sYmbol.CLEANUP.AlignmentPaddings

; consider conditional branches/instructions in the object code for the
; static analysis
sYmbol.ECA.BINary.ControlFlowMode.Trace ON

; consider source code instrumentation probe points for the
; static analysis
sYmbol.ECA.BINary.ControlFlowMode.INSTR ON

; perform the static analysis for condition, decision and MC/DC coverage
sYmbol.ECA.BINary.PROCESS

; start code coverage analysis
```

See also

- [sYmbol.ECA.BINary.ControlFlowMode.INSTR](#)

■ [sYmbol.ECA.BINary.EXPORT.GAPS](#)

□ [sYmbol.ECA.BINary.GAPNUMBER\(\)](#)

▲ 'Build Process' in 'Application Note for Trace-Based Code Coverage'

▲ 'Build Process' in 'Application Note for Trace-Based Code Coverage'
- [sYmbol.ECA.BINary.ControlFlowMode.Trace](#)

■ [sYmbol.ECA.BINary.view](#)

sYmbol.ECA.BINary.SetCONDitionOffset

Set condition offset

[build 134102 - DVD 09/2021]

Format:

sYmbol.ECA.BINary.SetCONDitionOffset <source> [<dec_index>] <index>
<address>
sYmbol.ECA.SetConditionOffset (deprecated)

Sets the address offset of a condition in relation to its decision. This is an expert command that should only be used under the guidance of Lauterbach support.

This command can also be accessed via the **Condition** pull-down in [sYmbol.ECA.BINary.view](#) window.

Example:

```
; set offset of condition 0 of decision in line 171 to 0x8
sYmbol.ECA.BINary.SetConditionOffset \"..\coverage.c\"171 0. 0x8

; set offset of condition 0 of second decision in line 327 to 0x4C
sYmbol.ECA.BINary.SetConditionOffset \"..\coverage.c\"327 1. 0. 0x4C
```

sYmbol.ECA.BINary.SetDecisionState

Disable/Enable decision evaluation

[build 134102 - DVD 09/2021]

Format:

sYmbol.ECA.BINary.SetDecisionState <source> [<dec_index>] <address>
ON | OFF
sYmbol.ECA.SetDecisionState (deprecated)

Disable or enable a decision for evaluation during code coverage. When a decision is disabled it is ignored when measuring code coverage with the following metrics: *Decision*, *Condition*, *MCDC*.

The current status of all decisions can be viewed with [sYmbol.ECA.BINary.view](#).

Example:

```
; Disable decision in line 171
sYmbol.ECA.BINary.SetDecisionState \"..\coverage.c\"171 OFF

; Enable second decision in line 327
sYmbol.ECA.BINary.SetDecisionState \"..\coverage.c\"327 1. ON
```

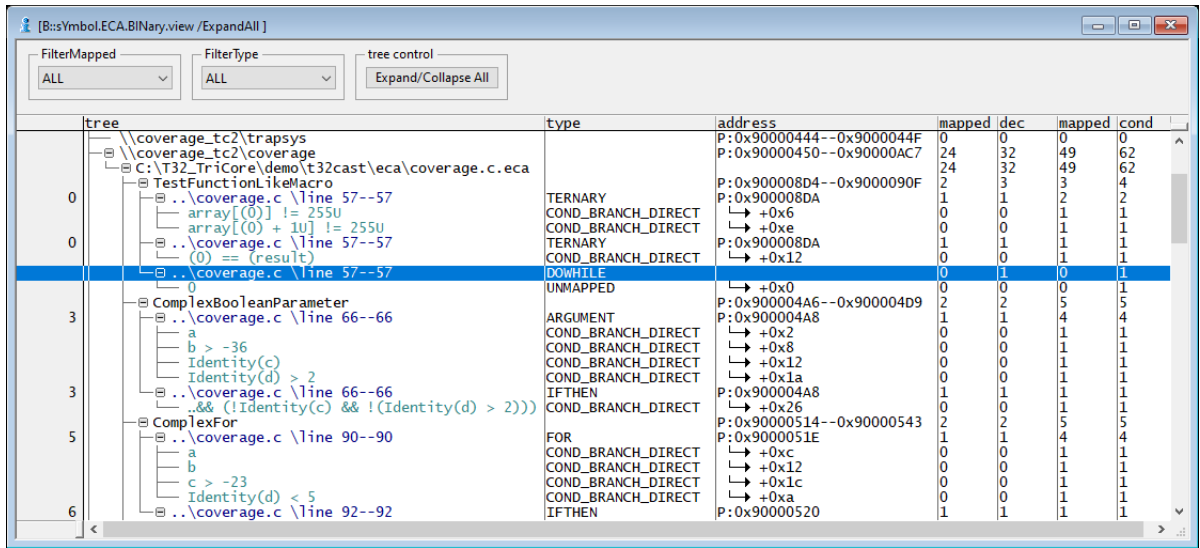
Format:

sYmbol.ECA.BINary.view [/<option>]
sYmbol.ECA.ViewDecisions (deprecated)

<option>:

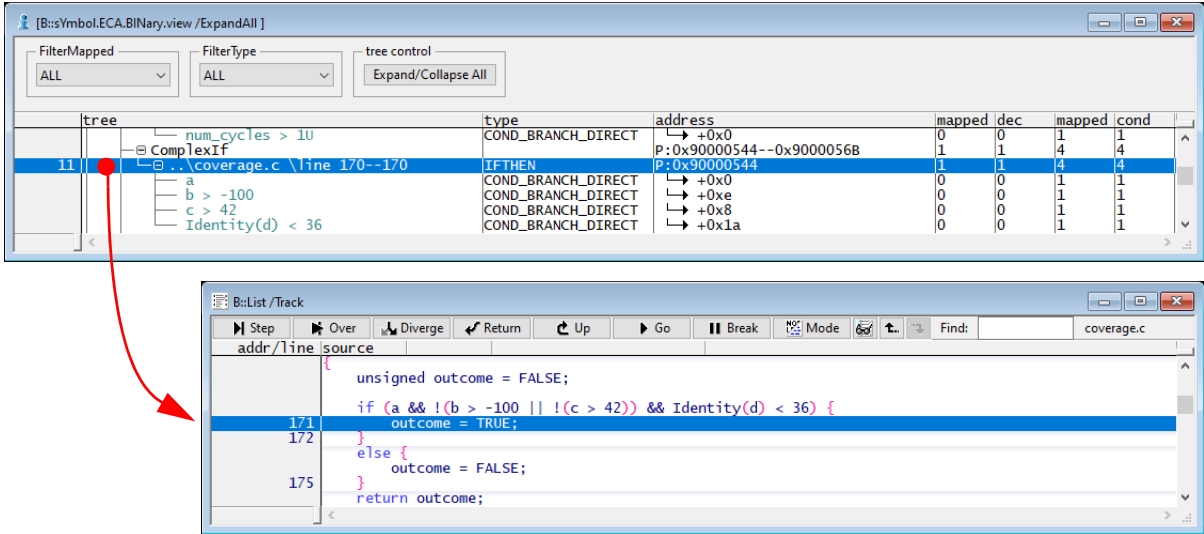
FilterMapped | FilterType | ExpandAll | StripPATH

This command allows to display the results of the static analysis performed by the command **sYmbol.ECA.BINary.PROCESS**.



Columns	
	Decision ID (TRACE32 numbers the decisions).
tree	TRACE32 internal tree: - module name - eca file with full path - function name - source code line including decision - atomic conditions of the decision

If a **List /Track** window is open, the source code line selected in the **sYmbol.ECA.BINary.view** window is referenced there. The exact type of the decision can be inspected.



Columns	
mapped dec	Number of mapped decisions.
dec	Overall number of decisions.
mapped cond	Mapped conditions.
cond	Overall number of conditions.

Options	
FilterMapped	Filter display by the mapping state, see command sYmbol.ECA.BINary.FilterMapped .
FilterType	Filter display by types, see command sYmbol.ECA.BINary.FilterType .
ExpandAll	Open window with all trees expanded. See also command sYmbol.ECA.BINary.ExpandAll .
StripPATH	Overall number of conditions.

See also

- [sYmbol.ECA.BINary.CollapseAll](#)
 - [sYmbol.ECA.BINary.EXPORT.GAPS](#)
 - [sYmbol.ECA.BINary.FilterType](#)
- [sYmbol.ECA.BINary.ExpandAll](#)
 - [sYmbol.ECA.BINary.FilterMapped](#)
 - [sYmbol.ECA.BINary.PROCESS](#)
- ▲ 'Build Process' in 'Application Note for Trace-Based Code Coverage'

Format:	sYmbol.ECA.Delete
---------	-------------------

Deletes all loaded ECA data and clears the [sYmbol.ECA.List](#) window.

See also

■ [sYmbol.ECA](#)

[build 142398 - DVD 02/2022]

Format:	sYmbol.ECA.Init
---------	-----------------

Clears data gathered by [sYmbol.ECA](#) commands.

See also

■ [sYmbol.ECA](#)

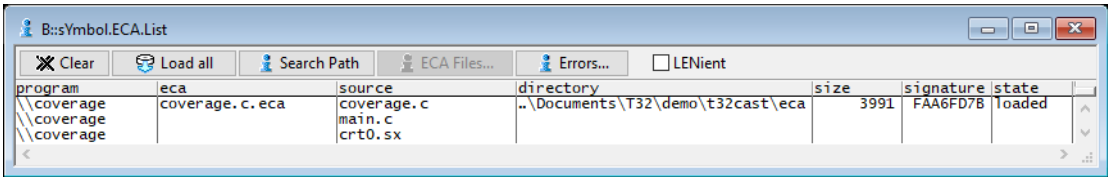
[\[Columns\]](#) [\[Toolbar\]](#) [\[Example\]](#)

Format:	sYmbol.ECA.List [/ERRORS]
---------	---------------------------

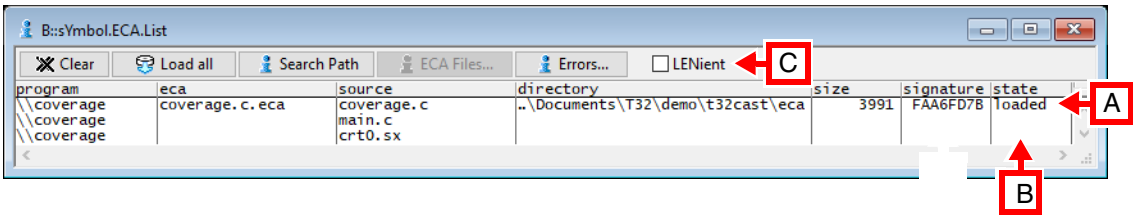
Lists details about the loaded .eca files.

The option **/ERRORS** advises TRACE32 PowerView to only display the .eca files that were loaded with an error.

Description of Columns in the sYmbol.ECA.List Window



source	List of all source files from the TRACE32 symbol database. The source file names are part of the debug information loaded with the program.
program	Name of the program that includes the object code generated for the source file.
eca	Name of the loaded .eca file.
directory	Directory containing the source file and the loaded .eca file.
size	Size of the loaded .eca file in bytes.
signature	Checksum signature of the loaded .eca file as a hex number.
state	The state loaded indicates that the ECA file was loaded successfully. The state error indicates one of the following: .eca file not found. .eca file contains invalid data. The state n/a indicates that TRACE32 does not support .eca files for the file type. E.g. no .eca data are supported for .asm files.



- A state** column => **loaded**: Double-clicking the row opens the ECA file in the TRACE32 **EDIT** window. The ECA file opens in an external editor if you have configured one with the **SETUP.EDITTEXT** command.
- B state** column => (empty): Double-clicking a row loads the ECA file, if available.
- C** Check box allows to set the **LENient** option for loading operations.

Clear	Deletes all previously loaded ECA data and clears the sYmbol.ECA.List window.
Touch all	Loads the .eca file for all source files. (command sYmbol.ECA.LOADALL /SkipErrors)
Search Path	Lists the search paths specified for the source files (command sYmbol.SourcePATH.List). TRACE32 expects the .eca files in the same directory as their source files.
ECA Files...	Displays full sYmbol.ECA.List window. The button is only active in the sYmbol.ECA.List /ERRORS window.
Errors...	Opens the sYmbol.ECA.List /ERRORS window, displaying only .eca files tagged with error .

Example: This script loads the .eca data for the module \coverage.

```
;defines directory as base for relative source paths
sYmbol.SourcePATH.SetBaseDir "J:/user1/projects/sources"
;load program
Data.LOAD Elf "coverage.elf" /RelPath
;load ECA file for the module \coverage
sYmbol.ECA.LOAD \coverage
sYmbol.ECA.List
```

See also

■ [sYmbol.ECA](#)

Format:

sYmbol.ECA.LOAD <module> [/<option>]

<option>:

SkipErrors | LENient | SetBaseDir

Loads the ECA data pertaining to the specified <module>.

- SkipErrors

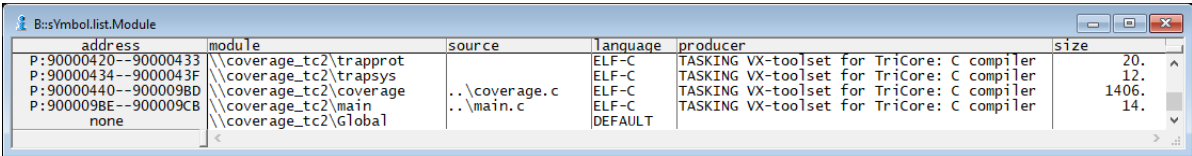
Ensures that warnings are issued instead of error messages. For scripts, error messages cause the script to stop. Warnings keep the script running.
- LENient

Allows loading of ECA files with minor errors as invalid file version or checksum mismatch.
- SetBaseDir

Forces the search of matching ECA files relative to the specified base directory.

```
;load ECA file for the module \coverage
sYmbol.ECA.LOAD \coverage
```

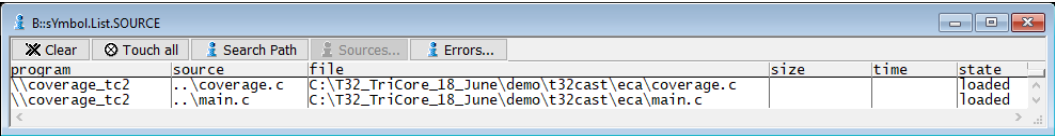
The **sYmbol.List.Module** command lists all modules from the TRACE32 symbol database.



address	module	source	language	producer	size
p:90000420--90000433	coverage_tc2\trapprot		ELF-C	TASKING VX-toolset for TriCore: C compiler	20.
p:90000434--9000043F	coverage_tc2\trapsys		ELF-C	TASKING VX-toolset for TriCore: C compiler	12.
p:90000440--900009BD	coverage_tc2\coverage	..\coverage.c	ELF-C	TASKING VX-toolset for TriCore: C compiler	1406.
p:900009BE--900009CB	coverage_tc2\main	..\main.c	ELF-C	TASKING VX-toolset for TriCore: C compiler	14.
none	coverage_tc2\Global		DEFAULT		

The column **source** shows the name of the corresponding source file.

After the source file is loaded, the column **file** in the **sYmbol.List.SOURCE** window shows the full source path.



program	source	file	size	time	state
coverage_tc2	..\coverage.c	C:\T32_TriCore_18_June\demo\t32cast\eca\coverage.c			loaded
coverage_tc2	..\main.c	C:\T32_TriCore_18_June\demo\t32cast\eca\main.c			loaded

TRACE32 searches for the .eca file under this source path.

An error message is displayed when the .eca file is not found or contains invalid data.

See also

- [sYmbol.ECA](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format:	sYmbol.ECA.LOADALL [/<option>]
<option>:	SkipErrors LENient SetBaseDir

Loads the ECA files for all modules listed in the TRACE32 symbol database.

SkipErrors	Ensures that warnings are issued instead of error messages. For scripts, error messages cause the script to stop. Warnings keep the script running.
LENient	Allows loading of ECA files with minor errors as invalid file version or checksum mismatch.
SetBaseDir	Forces the search of matching ECA files relative to the specified base directory.

Example: This script shows how to load all ECA data for the program `my_demo.elf`.

```
;load the elf file
Data.LOAD.elf my_demo.elf

;load ECA data for all modules
sYmbol.ECA.LOADALL /SkipErrors
```

See also

■ [sYmbol.ECA](#)

▲ ['Release Information' in 'Legacy Release History'](#)

Format:

sYmbol.FILTER.ADD.SOURCE <filter_name> <list_of_sources>

<list_of_sources>

<source0> [<source1> ...<source9>]

Several source files are combined under a filter name. The filter can then be used in commands as a representative of these source files. The syntax of the pathname is oriented towards the **source** column in the **sYmbol.Browse.SOURCE** window.

```
sYmbol.Browse.SOURCE

sYmbol.FILTER.ADD.SOURCE jd_files \
"D:/work/demo/mpc5xxx/mpc5646c_jpeg/jdapistd.c" \
"D:/work/demo/mpc5xxx/mpc5646c_jpeg/jdcolor.c" \
"D:/work/demo/mpc5xxx/mpc5646c_jpeg/jdmainct.c"

COVerage.ListFunc.preset jd_files

; Add a further source file
sYmbol.FILTER.ADD.SOURCE jd_files \
"D:/work/demo/mpc5xxx/mpc5646c_jpeg/jdhuff.c"
```

Format:

sYmbol.FILTER.ADD.sYmbol <filter_name> <list_of_symbols>

<list_of_symbols>

<symbol0> [<symbol1> ...<symbol9>]

Several symbols are combined under a filter name. The filter can then be used in commands as a representative of the symbols.

The example below combines modules to a filter.

```
sYmbol.Browse.Module  
  
sYmbol.FILTER.ADD.sYmbol jd_modules \jdcolor \jdmarker \jdtrans  
  
COVerage.ListFunc.preset jd_modules  
  
sYmbol.FILTER.ADD.sYmbol jd_modules \jdsample
```

sYmbol.FILTER.Delete

Delete filter

Format:

sYmbol.FILTER.Delete *<filter_name>*

Delete specified filter.

Format:	sYmbol.ForEach "<cmd>" [<name_pattern> [<type_pattern>]]
---------	---

Executes a PRACTICE command for each symbol matching the specified name and type patterns.

<cmd>	The command to be executed has to be specified with quotation marks. It may contain the characters '*' or '?' as placeholders that are replaced by the complete name of a matching symbol.
<name_pattern> <type_pattern>	The patterns are case-insensitive. Therefore lower and upper case characters are not distinguished. The following wildcards can be used in pattern expressions:
'*'	Matches any string, including empty strings. For the <type_pattern>, only symbols with HLL type information match.
'?'	Matches one (non-empty) character.
'\"'	Can be used to input special characters like '*' or '?'

Examples:

sYmbol.ForEach "Break.Set" *func*	; will execute the command ; Break.Set <symbol> ; for all symbols containing ; 'func'
sYmbol.ForEach "Break.Set Var.END(" *") /Charly" * *	; execute the command ; Break.Set ;Var.END(<symbol>) /Charly ; which sets a "Charly : ; breakpoint" to the ; last address of each HLL ;function or variable

For more examples on wildcards, see command [sYmbol.name](#).

See also

- [sYmbol](#)
- [sYmbol.MATCHES\(\)](#)

Format:

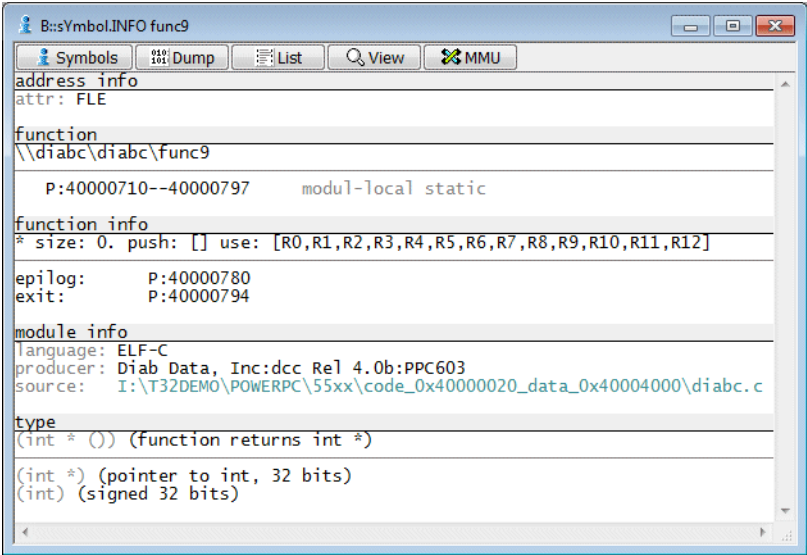
sYmbol.INFO <symbol> | <address> [/Track]

Displays symbolic address, location, scope and layout of the specified debug <symbol>. If an <address> is specified, the details about the debug symbol located at <address> are displayed.

The option /Track enables the address tracking. See [example 2](#).

Example 1:

```
sYmbol.INFO func9           ; display detailed debug information for
                             ; the function func9
```



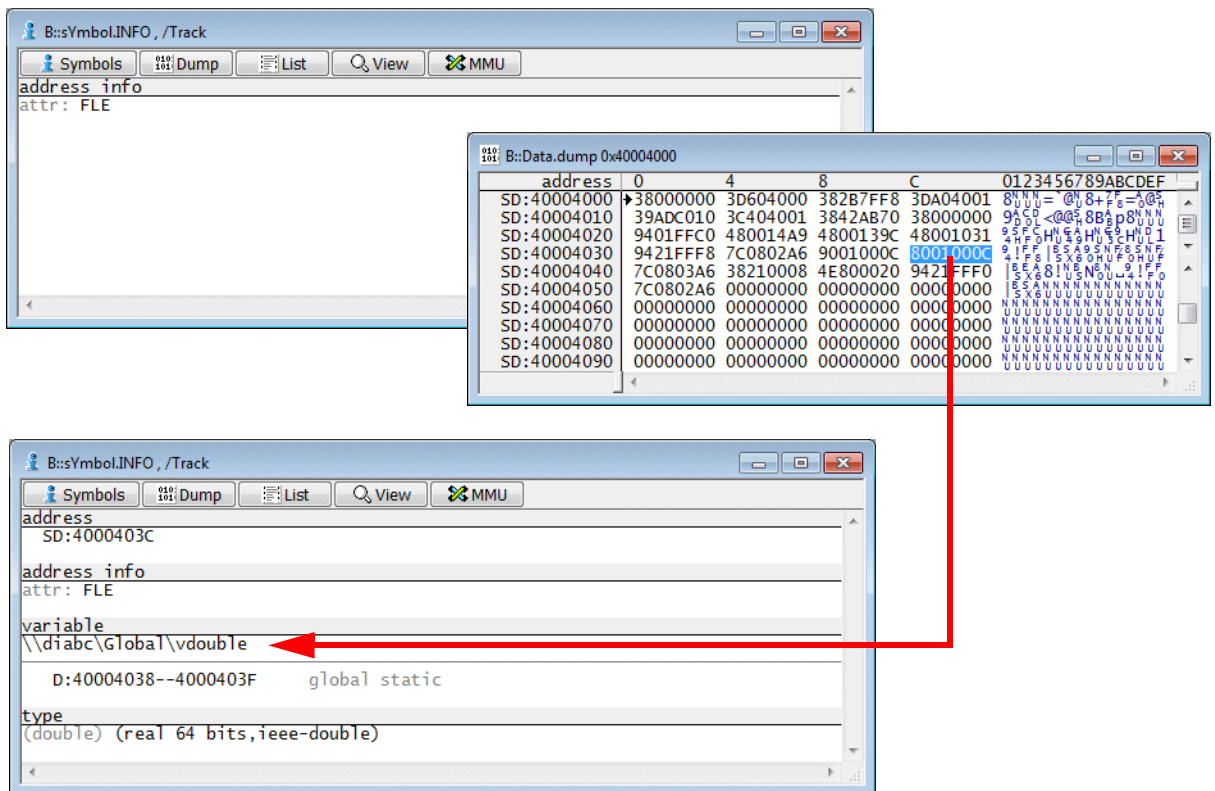
Example 2:

```
sYmbol.INFO , /Track          ; display a sYmbol.INFO window with address
                               ; tracking enabled

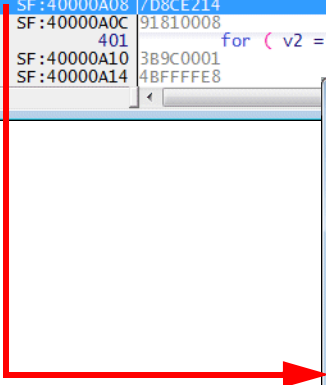
                               ; you have to use a ",", if you do not want
                               ; to specify the <symbol> parameter

Data.dump 0x40004000          ; display a hex dump starting at address
                               ; 0x40004000

; address tracking works as follows:
; as soon as you select an address in the Data.dump window, details about
; the symbol located at selected address are displayed
```



```
; Analogous, if a line is selected in the
; Source Listing all the details about the
; related function are displayed
```



- sYmbol
- sYmbol.Browse.name
- sYmbol.STATE
- Data.dump
- MMU.INFO
- Var.INFO

▲ 'The Symbol Database' in 'Training Source Level Debugging'

Format: **sYmbol.LANGUAGE** [*<language>*]

Selects the language and style, that is used for HLL expressions.

Example:

```
sYmbol.LANGUAGE MCC68K
```

See also

■ [sYmbol](#)

Format: sYmbol.List [<address>]

Displays the **sYmbol.List** window with a list of all symbols. The list is ordered by the symbols' addresses and scrolled so that <address> is shown at the top of the window.

The <address> can be entered as a literal (e.g. P: 0xFC004AB) or using a symbol path (e.g. \\sieve\page1\main). When using wildcards, the symbol path needs to evaluate to a single symbol.

See also

- | | | | |
|--|--|--|--|
| ■ sYmbol.List.ATTRibute | ■ sYmbol.List.BUILTIN | ■ sYmbol.List.ColorDef | ■ sYmbol.List.Enum |
| ■ sYmbol.List.FRAME | ■ sYmbol.List.Function | ■ sYmbol.List.IMPORT | ■ sYmbol.List.InlineBlock |
| ■ sYmbol.List.InlineFunction | ■ sYmbol.List.LINE | ■ sYmbol.List.Local | ■ sYmbol.List.MACRO |
| ■ sYmbol.List.MAP | ■ sYmbol.List.Module | ■ sYmbol.List.PATCH | ■ sYmbol.List.Program |
| ■ sYmbol.List.REFerence | ■ sYmbol.List.SECTION | ■ sYmbol.List.SOURCE | ■ sYmbol.List.SourceFunction |
| ■ sYmbol.List.SOURCETREE | ■ sYmbol.List.STACK | ■ sYmbol.List.Static | ■ sYmbol.List.TREE |
| ■ sYmbol.List.Type | ■ sYmbol | ■ sYmbol.STATE | |

sYmbol.List.ATTRibute

Display memory attributes

Format: sYmbol.List.ATTRibute [<address>]

Displays memory attributes. Memory attributes can classify the code or execution model at a specific address. This information is highly compiler dependent.

See also

- [sYmbol.List](#)

sYmbol.List.BUILTIN

List built-in data types

Format: sYmbol.List.BUILTIN

Lists all built-in data types of the used programming language.

See also

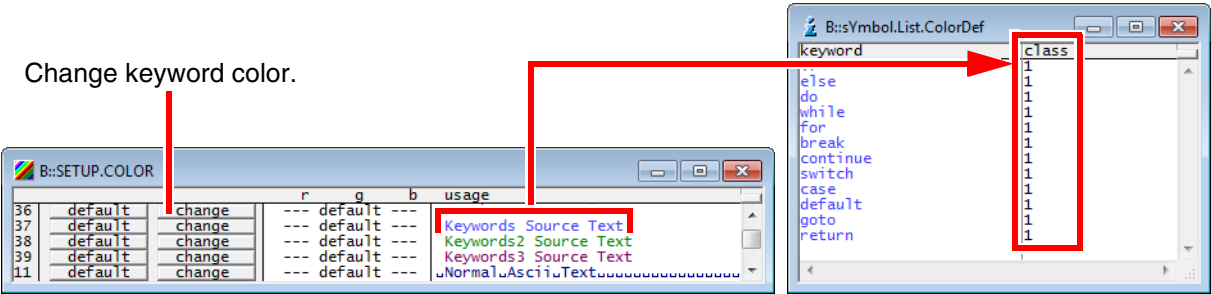
- [sYmbol.List](#)

Format:

sYmbol.List.ColorDef

Lists the color definition for the keywords of the HLL code displayed in the **Data.List** window (if, else, while, etc.). The **class** column in the **sYmbol.List.ColorDef** window shows the currently assigned formatting class (= style in a word processing application such as OpenOffice.Org Writer). By default, the keywords are assigned to the formatting class **1**, and its default color is blue.

By assigning keywords to class 2 or 3, you can format keywords green or purple. If you want to pick another color for the classes 2 and 3, then click the **change** button in the **SETUP.COLOR** window.



Example 1: This script shows how to change the color of the `for` keyword. To try this script, copy it to a `test.cmm` file, and then step through it in TRACE32 (See “[How to...](#)”).

```
sYmbol.List.ColorDef      ;Display the keywords, their colors, and
                          ;the classes to which the keywords are assigned

SETUP.COLOR              ;Display the current color settings

sYmbol.ColorDef "for" 2.  ;Assign the keyword 'for' to class 2 =>
                          ;'for' turns green

sYmbol.ColorDef "for" 3.  ;Assign the keyword 'for' to class 3 =>
                          ;'for' turns purple

SETUP.COLOR 39. 255. 128. 0. ;Change the color of class 3 to orange

SETUP.COLOR 39.           ;Restore the default color of class 3 again

sYmbol.ColorDef "for" 1.  ;Apply class 1 to the 'for' keyword again
```

Example 2: A PRACTICE script that adds more keywords to the **sYmbol.List.ColorDef** window is included in your TRACE32 installation. To access the script, run this command:
B::CD.PSTEP ~/demo/practice/colors/syntaxcolor.cmm

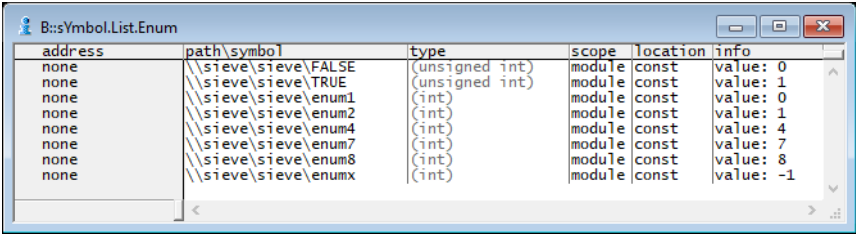
See also

- sYmbol.List
- sYmbol.ColorDef
- SETUP.COLOR
- ▲ 'PowerView - Screen Display' in 'PowerView User's Guide'

Format:

sYmbol.List.Enum

Lists all enumeration constants with the related information.



address	path\symbol	type	scope	location	info
none	\\sieve\sieve\FALSE	(unsigned int)	module	const	value: 0
none	\\sieve\sieve\TRUE	(unsigned int)	module	const	value: 1
none	\\sieve\sieve\enum1	(int)	module	const	value: 0
none	\\sieve\sieve\enum2	(int)	module	const	value: 1
none	\\sieve\sieve\enum4	(int)	module	const	value: 4
none	\\sieve\sieve\enum7	(int)	module	const	value: 7
none	\\sieve\sieve\enum8	(int)	module	const	value: 8
none	\\sieve\sieve\enumx	(int)	module	const	value: -1

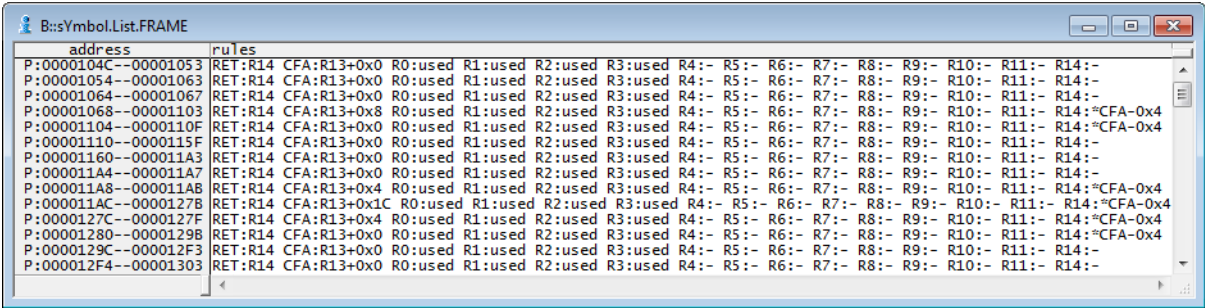
See also

- [sYmbol.List](#)

Format:

sYmbol.List.FRAME [<address>]

Lists the location and further related information about the frames. In the **sYmbol.List.FRAME** window, each entry tells the debugger for a certain program range where the registers are saved, e.g. relative to the current stack pointer (SP).



Description of Columns in the sYmbol.List.FRAME Window

address	Address range of a single frame
rules	Rules used by TRACE32 PowerView to recover the previous stack frames.

Description of Values in the “rules” Column

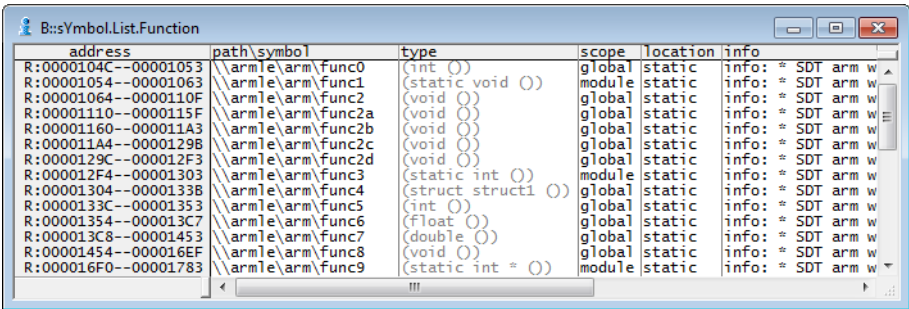
RET	Location of the return value, e.g. a register
R0 to R14	Register names (architecture-specific)
CFA	Canonical frame address
<register>:used	Registers which cannot be recovered
<register>:-	Registers with valid content
*CFA-<offset> or *CFA+<offset>	Address where the register content can be recovered from, e.g. *CFA-0x4

See also

- sYmbol.List
- Frame

Format: sYmbol.List.Function [<range> | <address>]

Lists the location and further related information about the loaded functions.



The screenshot shows a debugger window titled "B::sYmbol.List.Function" displaying a table of loaded functions. The table has six columns: address, path\symbol, type, scope, location, and info. The data is as follows:

address	path\symbol	type	scope	location	info
R:0000104C--00001053	armle\arm\func0	(int ())	global	static	info: * SDT arm w
R:00001054--00001063	armle\arm\func1	(static void ())	module	static	info: * SDT arm w
R:00001064--0000110F	armle\arm\func2	(void ())	global	static	info: * SDT arm w
R:00001110--0000115F	armle\arm\func2a	(void ())	global	static	info: * SDT arm w
R:00001160--000011A3	armle\arm\func2b	(void ())	global	static	info: * SDT arm w
R:000011A4--00001298	armle\arm\func2c	(void ())	global	static	info: * SDT arm w
R:0000129C--000012F3	armle\arm\func2d	(void ())	global	static	info: * SDT arm w
R:000012F4--00001303	armle\arm\func3	(static int ())	module	static	info: * SDT arm w
R:00001304--0000133B	armle\arm\func4	(struct struct1 ())	global	static	info: * SDT arm w
R:0000133C--00001353	armle\arm\func5	(int ())	global	static	info: * SDT arm w
R:00001354--000013C7	armle\arm\func6	(Float ())	global	static	info: * SDT arm w
R:000013C8--00001453	armle\arm\func7	(double ())	global	static	info: * SDT arm w
R:00001454--000016EF	armle\arm\func8	(void ())	global	static	info: * SDT arm w
R:000016F0--00001783	armle\arm\func9	(static int * ())	module	static	info: * SDT arm w

See also

- [sYmbol.List](#)

sYmbol.List.IMPORT

List imported symbols

Format: sYmbol.List.IMPORT

Lists the symbols that are loaded by imported DLLs (required e.g. for Symbian or Windows CE).

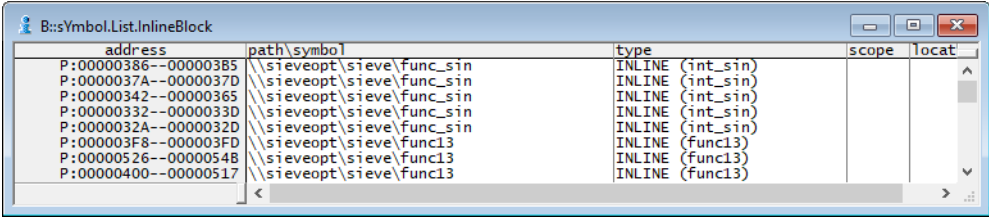
See also

- [sYmbol.List](#)

Format:

sYmbol.List.InlineBlock

When compiling with optimization the compiler may insert functions or parts of a function directly instead of adding a call to the function. This command lists all parts of the code where function parts have been inlined by the compiler.



The screenshot shows a window titled "B::sYmbol.List.InlineBlock" with a table of inlined code blocks. The table has five columns: address, path\symbol, type, scope, and location. The data is as follows:

address	path\symbol	type	scope	location
P:00000386--00000385	\\sieveopt\sieve\func_sin	INLINE (int_sin)		
P:0000037A--0000037D	\\sieveopt\sieve\func_sin	INLINE (int_sin)		
P:00000342--00000365	\\sieveopt\sieve\func_sin	INLINE (int_sin)		
P:00000332--0000033D	\\sieveopt\sieve\func_sin	INLINE (int_sin)		
P:0000032A--0000032D	\\sieveopt\sieve\func_sin	INLINE (int_sin)		
P:000003F8--000003FD	\\sieveopt\sieve\func13	INLINE (func13)		
P:00000526--00000548	\\sieveopt\sieve\func13	INLINE (func13)		
P:00000400--00000517	\\sieveopt\sieve\func13	INLINE (func13)		

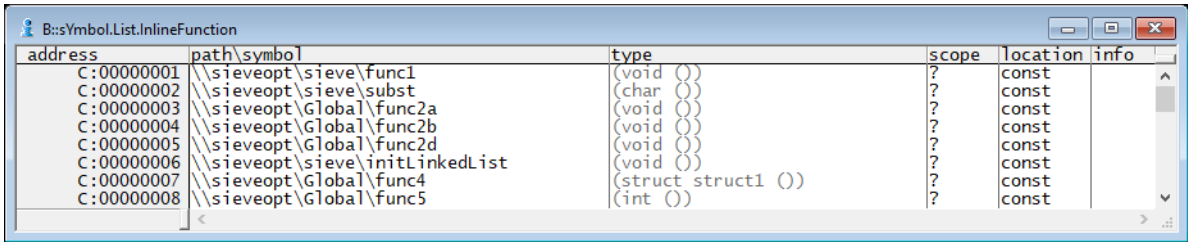
See also

- sYmbol.List

Format:

sYmbol.List.InlineFunction

Lists the location and further related information about the loaded inline functions.



The screenshot shows a window titled "B::sYmbol.List.InlineFunction" with a table of inlined functions. The table has six columns: address, path\symbol, type, scope, location, and info. The data is as follows:

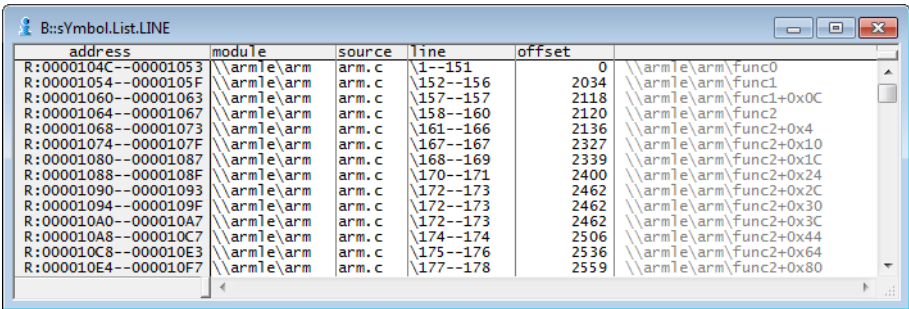
address	path\symbol	type	scope	location	info
C:00000001	\\sieveopt\sieve\func1	(void ())	?	const	
C:00000002	\\sieveopt\sieve\subst	(char ())	?	const	
C:00000003	\\sieveopt\Global\func2a	(void ())	?	const	
C:00000004	\\sieveopt\Global\func2b	(void ())	?	const	
C:00000005	\\sieveopt\Global\func2d	(void ())	?	const	
C:00000006	\\sieveopt\sieve\initLinkedList	(void ())	?	const	
C:00000007	\\sieveopt\Global\func4	(struct struct1 ())	?	const	
C:00000008	\\sieveopt\Global\func5	(int ())	?	const	

See also

- sYmbol.List

Format: sYmbol.List.LINE [<address>]

Displays the location and further related information about the loaded lines.



The screenshot shows a debugger window titled "B::sYmbol.List.LINE" displaying a table of loaded lines. The table has six columns: address, module, source, line, offset, and a final column for the symbol name. The data is as follows:

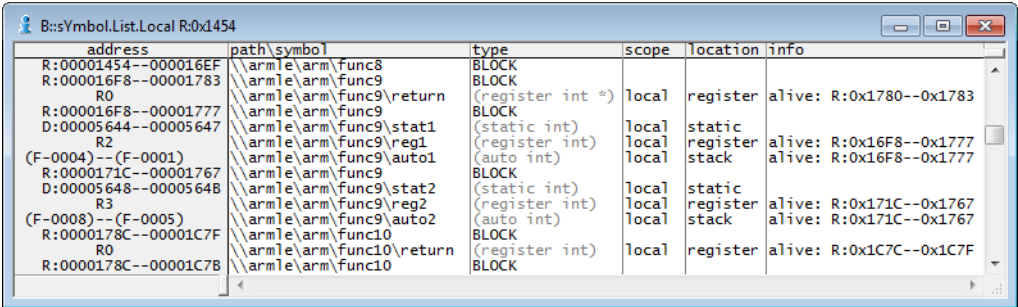
address	module	source	line	offset	
R:0000104C--00001053	\armle\arm	arm.c	1--151	0	\armle\arm\func0
R:00001054--0000105F	\armle\arm	arm.c	152--156	2034	\armle\arm\func1
R:00001060--00001063	\armle\arm	arm.c	157--157	2118	\armle\arm\func1+0x0C
R:00001064--00001067	\armle\arm	arm.c	158--160	2120	\armle\arm\func2
R:00001068--00001073	\armle\arm	arm.c	161--166	2136	\armle\arm\func2+0x4
R:00001074--0000107F	\armle\arm	arm.c	167--167	2327	\armle\arm\func2+0x10
R:00001080--00001087	\armle\arm	arm.c	168--169	2339	\armle\arm\func2+0x1C
R:00001088--0000108F	\armle\arm	arm.c	170--171	2400	\armle\arm\func2+0x24
R:00001090--00001093	\armle\arm	arm.c	172--173	2462	\armle\arm\func2+0x2C
R:00001094--0000109F	\armle\arm	arm.c	172--173	2462	\armle\arm\func2+0x30
R:000010A0--000010A7	\armle\arm	arm.c	172--173	2462	\armle\arm\func2+0x3C
R:000010A8--000010C7	\armle\arm	arm.c	174--174	2506	\armle\arm\func2+0x44
R:000010C8--000010E3	\armle\arm	arm.c	175--176	2536	\armle\arm\func2+0x64
R:000010E4--000010F7	\armle\arm	arm.c	177--178	2559	\armle\arm\func2+0x80

See also

- sYmbol.List

Format: sYmbol.List.Local [<range> | <address>]

Displays all symbols local to functions and blocks.

The screenshot shows a debugger window titled "B::sYmbol.List.Local R:0x1454". It displays a table of local symbols. The table has columns: address, path\symbol, type, scope, location, and info. The data rows show symbols for functions func8, func9, and func10, including their return values, static variables, and registers. Some entries are marked as "alive" with specific address ranges.

address	path\symbol	type	scope	location	info
R:00001454--000016EF	armle\arm\func8	BLOCK			
R:000016F8--00001783	armle\arm\func9	BLOCK			
R0	armle\arm\func9\return	(register int *)	local	register	alive: R:0x1780--0x1783
R:000016F8--00001777	armle\arm\func9	BLOCK			
D:00005644--00005647	armle\arm\func9\stat1	(static int)	local	static	
R2	armle\arm\func9\reg1	(register int)	local	register	alive: R:0x16F8--0x1777
(F-0004)--(F-0001)	armle\arm\func9\auto1	(auto int)	local	stack	alive: R:0x16F8--0x1777
R:0000171C--00001767	armle\arm\func9	BLOCK			
D:00005648--0000564B	armle\arm\func9\stat2	(static int)	local	static	
R3	armle\arm\func9\reg2	(register int)	local	register	alive: R:0x171C--0x1767
(F-0008)--(F-0005)	armle\arm\func9\auto2	(auto int)	local	stack	alive: R:0x171C--0x1767
R:0000178C--00001C7F	armle\arm\func10	BLOCK			
R0	armle\arm\func10\return	(register int)	local	register	alive: R:0x1C7C--0x1C7F
R:0000178C--00001C7B	armle\arm\func10	BLOCK			

See also

■ [sYmbol.List](#)

sYmbol.List.MACRO

List all C macros

Format: sYmbol.List.MACRO

List all C macros. C macros can either be loaded with [Data.LOAD](#) <file> /MACRO or they can be created with the command [sYmbol.Create.MACRO](#).

See also

■ [sYmbol.List](#)

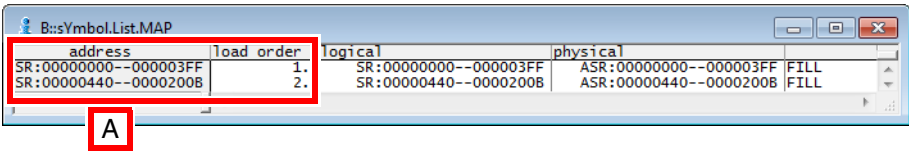
Format: **sYmbol.List.MAP** [<address>]

Displays address ranges where code was saved during download and the order in which the code was saved. Can be used to find out “where the code has gone”.

Example:

```
<your_code>
SYStem.Up                               ;connect to target
Data.LOAD.Elf  sieve_flash_thumb_ii_v7m.elf ;load application to target

sYmbol.List.MAP                          ;let's find out
                                           ;"where the code has gone"
```



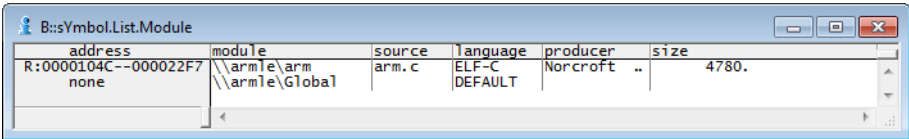
A The loaded *.elf file contains two section, and the code has been loaded to these two address ranges.

See also

- sYmbol.List
- sYmbol.List.MAP.COUNT()
- sYmbol.List.MAP.RANGE()

Format: **sYmbol.List.Module** [<address>]

Displays information about the loaded models, e.g. the location or the loaded source file.



See also

- sYmbol.List

Format:

sYmbol.List.PATCH [<address>]

Is a alias for [sYmbol.PATCH.List](#).

See also

[sYmbol.List](#)

sYmbol.List.Program

Display programs

Format:

sYmbol.List.Program

Displays the location and further related information about the loaded programs.

B:sYmbol.List.Program

address	name	format	file	command	sourcepath
P:00001000--0000559B	arm1e	ELF/DWARF2	C:\T32\demo\arm\compiler\arm\arm1e.axf	Data.LOAD.ELF "~~~~/arm1e.axf" /SPATH /LPATH C:\T32\demo	

sourcepath	types	statics	locals	modules	sources	lines
C:\T32\demo\arm\compiler\arm	216.	286.	177.	2.	1.	410.

lines	stacks	frames	attribute	imports	exports
410.	73.	118.	222.	0.	0.

See also

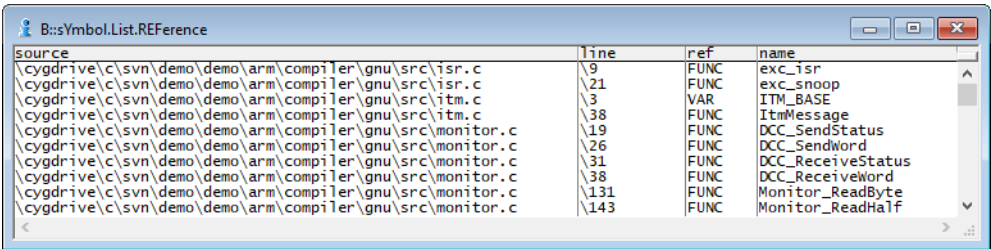
[sYmbol.List](#) [sYmbol.LIST.PROGRAM\(\)](#)

Format:

sYmbol.List.REFerence

Displays DWARF declaration info. The ELF file ha to loaded with **/REFerence** option:

```
Data.LOAD.ELF <elf_file> /REFerence
```



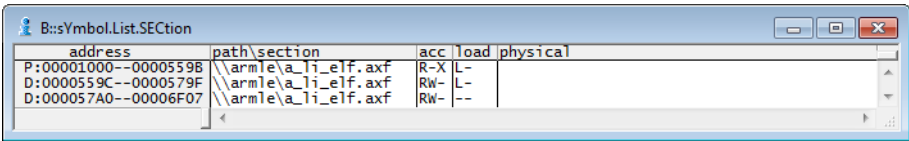
source	line	ref	name
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\isr.c	9	FUNC	exc_isr
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\isr.c	21	FUNC	exc_snoop
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\itm.c	3	VAR	ITM_BASE
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\itm.c	38	FUNC	ItmMessage
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\monitor.c	19	FUNC	DCC_SendStatus
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\monitor.c	26	FUNC	DCC_SendWord
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\monitor.c	31	FUNC	DCC_ReceiveStatus
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\monitor.c	38	FUNC	DCC_ReceiveWord
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\monitor.c	131	FUNC	Monitor_ReadByte
cygdrive\c\svn\demo\demo\arm\compiler\gnu\src\monitor.c	143	FUNC	Monitor_ReadHalf

See also

- [sYmbol.List](#)

Format: **sYmbol.List.SEction** [<address>]

Opens the **sYmbol.List.SEction** window, displaying the logical address ranges, section names, and access rights for the sections. Physical address ranges are also displayed, provided the file type contains any.



Description of Columns in the sYmbol.List.SEctions Window

address	Logical address range of a section.
path\section	Section names in source files.
acc	Access rights: - R: Read access - W: Write access - X: Section can execute code
load	- L: Section is loaded by the debugger. - O: Section is set to zero by the debugger.
physical	Physical address range of a section - only for certain file types and architectures.

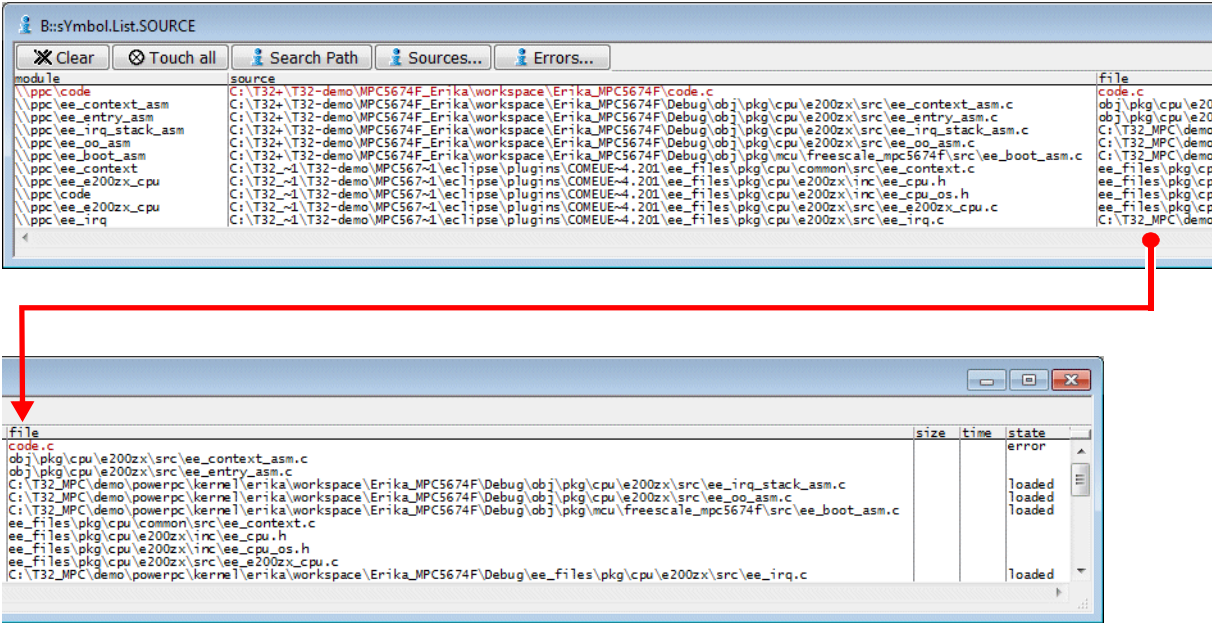
See also

- [sYmbol.List](#)
[sYmbol.SECPRANGE\(\)](#)
- [sYmbol.SECADDRESS\(\)](#)
[sYmbol.SECRANGE\(\)](#)
- [sYmbol.SECEND\(\)](#)
- [sYmbol.SECNAME\(\)](#)

Format:

sYmbol.List.SOURCE [/ERRORS]

If a files with debug information is loaded (**Data.LOAD.<file_format>**), this file also provides the paths for the HLL (C/C++/JAVA etc.) source files. The command **sYmbol.List.SOURCE** lists the file names and paths in the source column.



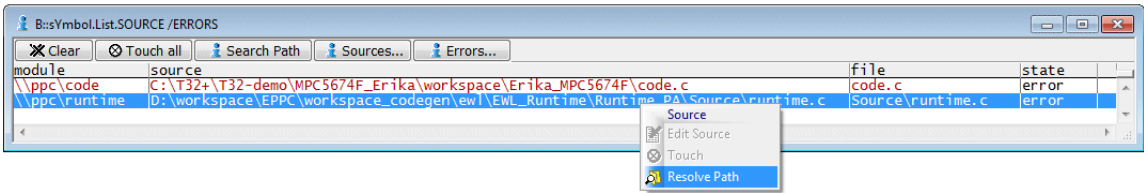
Description of Toolbar Buttons in the sYmbol.List.SOURCE Window

Clear	Invalidates the state column for all source files and invalidates the search results displayed in the file column (command sYmbol.SourceRELOAD).
Touch All	By default an HLL source file is only loaded by TRACE32 when the contents of the HLL source file is required during debugging. This button advises TRACE32 to load all source files (command sYmbol.SourceLOAD).
Search Path	Displays details on source search paths (command sYmbol.SourcePATH.List).
Errors	Display only modules tagged with error in the state column.

module	Path of the compiled file as it is maintained by TRACE32.
source	Source path for C/C++/JAVA source file after the completion of the Data.LOAD.<file_format> command.
file	Source path from which the C/C++/JAVA source file was actually loaded. Required adjustments of the source path can be performed with the help of the sYmbol.SourcePATH command group.
size	Size information as provided by the loaded file.
time	Time information as provided by the loaded file.
state	The state <i>loaded</i> indicates that the C/C++/JAVA source file was loaded successfully. The state <i>error</i> indicates that the C/C++/JAVA source file could not be loaded. Modules tagged with <i>error</i> are printed in red.

Example:

```
sYmbol.List.SOURCE /ERRORS           // Advise TRACE32 PowerView to display
                                     // only modules tagged with error.
```



Resolve Path in the **Source** context menu opens a dialog where you can choose the correct path for the selected module/source. TRACE32 adds the result to the source search paths. The result can be inspected via the **sYmbol.SourcePATH.List** command. The procedure is as follows:

- Absolute paths are fixed with the help of the command **sYmbol.SourcePATH.Translate**.
- Relative paths are fixed with the help of the command **sYmbol.SourcePATH.SetBaseDir**.

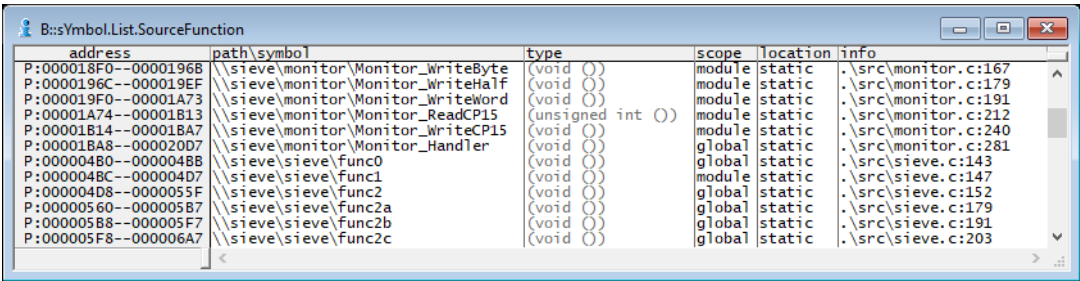
See also

- **sYmbol.List**
- **sYmbol.ECA**
- **sYmbol.LIST.SOURCE()**
- ▲ 'Release Information' in 'Legacy Release History'

Format:

sYmbol.List.SourceFunction [<range> | <address>]

Lists the location and source file information about the loaded functions.



address	path\symbol	type	scope	location info
P:000018F0--00001968	\\sieve\monitor\Monitor_WriteByte	(void ())	module static	.\src\monitor.c:167
P:0000196C--000019EF	\\sieve\monitor\Monitor_WriteHalf	(void ())	module static	.\src\monitor.c:179
P:000019F0--00001A73	\\sieve\monitor\Monitor_WriteWord	(void ())	module static	.\src\monitor.c:191
P:00001A74--00001B13	\\sieve\monitor\Monitor_ReadCP15	(unsigned int ())	module static	.\src\monitor.c:212
P:00001B14--00001BA7	\\sieve\monitor\Monitor_WriteCP15	(void ())	module static	.\src\monitor.c:240
P:00001BA8--000020D7	\\sieve\monitor\Monitor_Handler	(void ())	global static	.\src\monitor.c:281
P:00000480--0000048B	\\sieve\sieve\func0	(void ())	global static	.\src\sieve.c:143
P:0000048C--000004D7	\\sieve\sieve\func1	(void ())	module static	.\src\sieve.c:147
P:000004D8--0000055F	\\sieve\sieve\func2	(void ())	global static	.\src\sieve.c:152
P:00000560--000005B7	\\sieve\sieve\func2a	(void ())	global static	.\src\sieve.c:179
P:000005B8--000005F7	\\sieve\sieve\func2b	(void ())	global static	.\src\sieve.c:191
P:000005F8--000006A7	\\sieve\sieve\func2c	(void ())	global static	.\src\sieve.c:203

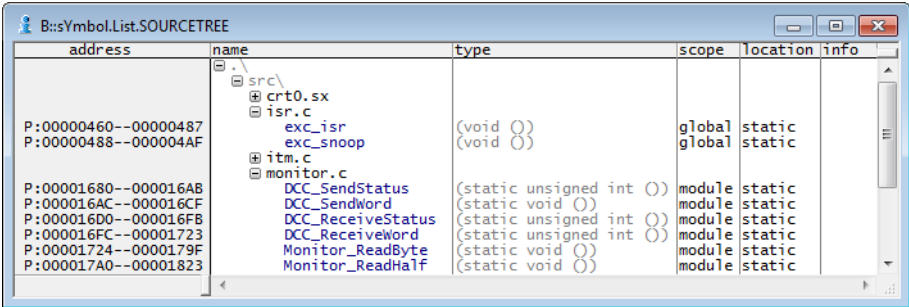
See also

- sYmbol.List

Format:

sYmbol.List.SOURCETREE

Displays the hierarchy of the source files in a tree structure.



See also

■ [sYmbol.List](#)

Format:

sYmbol.List.STACK [<address>]

Displays information about virtual stack pointers. This information is highly compiler dependent.

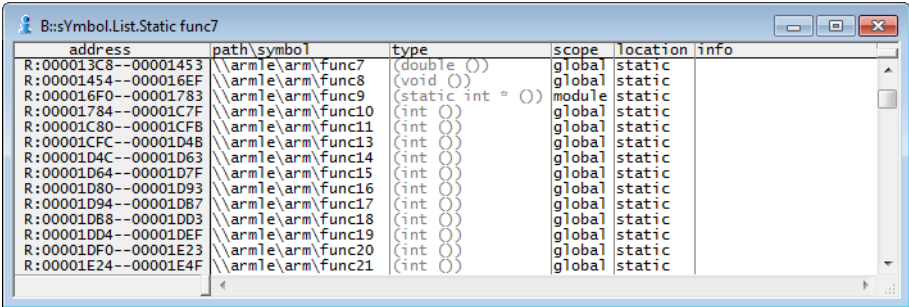
See also

■ [sYmbol.List](#)

Format:

sYmbol.List.Static [<range> | <address>]

Displays all symbols with a fixed address.



address	path\symbol	type	scope	location	info
R:000013C8--00001453	armle\arm\func7	(double ())	global	static	
R:00001454--000016EF	armle\arm\func8	(void ())	global	static	
R:000016F0--00001783	armle\arm\func9	(static int * ())	module	static	
R:00001784--00001C7F	armle\arm\func10	(int ())	global	static	
R:00001C80--00001CFB	armle\arm\func11	(int ())	global	static	
R:00001D4C--00001D48	armle\arm\func13	(int ())	global	static	
R:00001D4C--00001D63	armle\arm\func14	(int ())	global	static	
R:00001D64--00001D7F	armle\arm\func15	(int ())	global	static	
R:00001D80--00001D93	armle\arm\func16	(int ())	global	static	
R:00001D94--00001DB7	armle\arm\func17	(int ())	global	static	
R:00001DB8--00001DD3	armle\arm\func18	(int ())	global	static	
R:00001DD4--00001DEF	armle\arm\func19	(int ())	global	static	
R:00001DF0--00001E23	armle\arm\func20	(int ())	global	static	
R:00001E24--00001E4F	armle\arm\func21	(int ())	global	static	

See also

- sYmbol.List

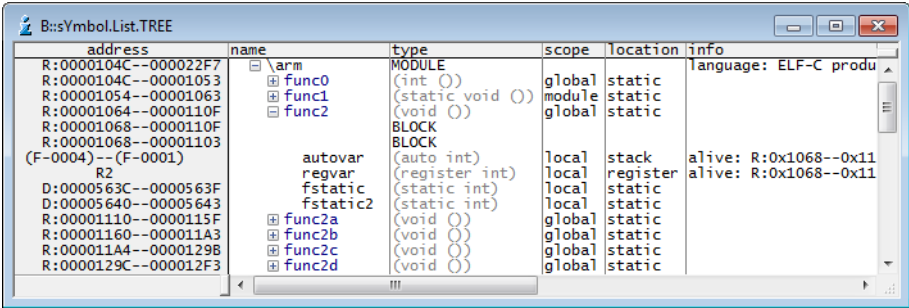
sYmbol.List.TREE

Display symbols in tree form

Format:

sYmbol.List.TREE

Displays modules, symbols, and variables in a tree structure.



address	name	type	scope	location	info
R:0000104C--000022F7	arm	MODULE			language: ELF-C produ
R:0000104C--00001053	func0	(int ())	global	static	
R:00001054--00001063	func1	(static void ())	module	static	
R:00001064--0000110F	func2	(void ())	global	static	
R:00001068--0000110F		BLOCK			
R:00001068--00001103		BLOCK			
(F-0004)--(F-0001)	autovar	(auto int)	local	stack	alive: R:0x1068--0x11
R2	regvar	(register int)	local	register	alive: R:0x1068--0x11
D:0000563C--0000563F	fstatic	(static int)	local	static	
D:00005640--00005643	fstatic2	(static int)	local	static	
R:00001110--0000115F	func2a	(void ())	global	static	
R:00001160--000011A3	func2b	(void ())	global	static	
R:000011A4--0000129B	func2c	(void ())	global	static	
R:0000129C--000012F3	func2d	(void ())	global	static	

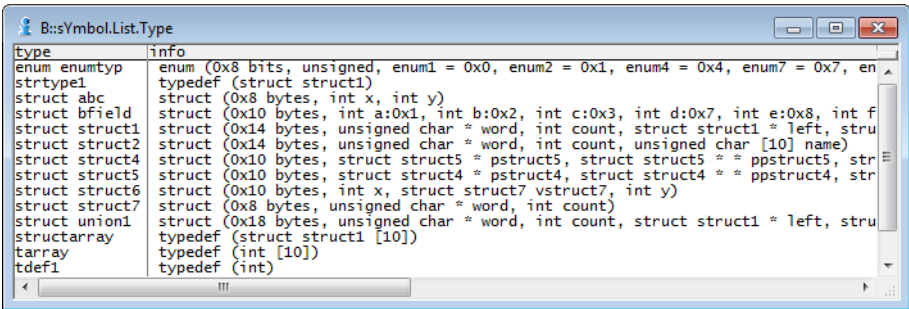
See also

- sYmbol.List

Format:

sYmbol.List.Type [/Unnamed]

Displays all data types used by the compiler.



Unnamed

Displays also unnamed types, e.g. "char *".

See also

- sYmbol.List

Using the **sYmbol.LSTLOAD** command group, you can load various formats of assembler list files for source text debugging on assembler level.

See also

- [sYmbol.LSTLOAD.GHILLS](#)
 - [sYmbol.LSTLOAD.IAR](#)
 - [sYmbol.LSTLOAD.INTEL](#)
 - [sYmbol.LSTLOAD.KEIL](#)
 - [sYmbol.LSTLOAD.MRI68K](#)
 - [sYmbol](#)
- [sYmbol.LSTLOAD.HPASM](#)
 - [sYmbol.LSTLOAD.INT68K](#)
 - [sYmbol.LSTLOAD.INTEL2](#)
 - [sYmbol.LSTLOAD.MicroWare](#)
 - [sYmbol.LSTLOAD.OAK](#)

sYmbol.LSTLOAD.GHILLS

Load GHILLS assembler source file

Format: **sYmbol.LSTLOAD.GHILLS** <module>|<program> <file> [<base_address>]

Loading of a GHILLS assembler list file for source text debugging on assembler level. If the base address of the module doesn't fit, the base address will be given as an argument.

See also

- [sYmbol.LSTLOAD](#)

sYmbol.LSTLOAD.HPASM

Load HP assembler source file

Format: **sYmbol.LSTLOAD.HPASM** <module>|<program> <file> [<base>]

Format: **sYmbol.LSTLOAD.HPASM2** <module>|<program> <file> [<base>]

Loading of an HP assembler list file for source text debugging on assembler level. If the base address of the module doesn't fit, the base address will be given as an argument. If a program name is given, the module entry will be generated from the file name of the list file. The different commands load different HP assembler list file formats.

The debugging can be controlled by the following assembler comments:

; T32-ORG

To mark lines including ORG statements.

;T32-OFF

To switch of the source text debugging. The debugging must be switched of for lines containing data statements or definitions (i.e. lines which address column not containing a program address). Include files, which shall not be displayed, must be switched off also.

NOTE:

The source line numbers will not match in this case, as the relation to the original source is lost.

;T32-ON

Reactivation of the debugging function.

```
Data.LOAD.HP sps                ; load the module and symbols
sYmbol.LSTLOAD.HPASM \SPS1 sps1.lst ; load the source lines for
sYmbol.LSTLOAD.HPASM \SPS2 sps2.lst ; module 1
                                   ; load the source lines for
                                   ; module 2
```

See also

■ [sYmbol.LSTLOAD](#)

Format:

sYmbol.LSTLOAD.IAR <module>|<program> <file> [<base_address>]

Loading of a IAR assembler list file for source text debugging on assembler level. If the base address of the module doesn't fit, the base address will be given as an argument.

The first comment line of the file (beginning with **'***) must start in the **first column** of the source text! The debugging can be controlled by the following assembler comments:

*T32-ORG

To mark lines including ORG statements.

*T32-OFF

To switch of the source text debugging. The debugging must be switched of for lines containing data statements or definitions (i.e. lines which address column not containing a program address). Include files, which shall not be displayed, must be switched off also.

NOTE:

The source line numbers will not match in this case, as the relation to the original source is lost.

*T32-ON

Reactivation of the debugging function.

See also

■ [sYmbol.LSTLOAD](#)

Format: **sYmbol.LSTLOAD.INT68K** *<module>|<program> <file> [<base_address>]*

Loading of a Intermetrics assembler list file for source text debugging on assembler level. If the base address of the module doesn't fit, the base address will be given as an argument.

The debugging can be controlled by the following assembler comments:

*T32-ORG

To mark lines including ORG statements.

*T32-OFF

To switch of the source text debugging. The debugging must be switched of for lines containing data statements or definitions (i.e. lines which address column not containing a program address).

*T32-ON

Reactivation of the debugging function.

See also

■ [sYmbol.LSTLOAD](#)

Format: **sYmbol.LSTLOAD.INTEL** *<module>|<program> <file> [<base_address>]*

Loading of a INTEL assembler list file for source text debugging on assembler level. If the base address of the module doesn't fit, the base address will be given as an argument.

See also

■ [sYmbol.LSTLOAD](#)

Format: **sYmbol.LSTLOAD.INTEL2** *<module>|<program> <file> [<base_address>]*

Loading of a INTEL multi-segment assembler list file for source text debugging on assembler level. If the base address of the module doesn't fit, the base address will be given as an argument. This

See also

■ [sYmbol.LSTLOAD](#)

Format: **sYmbol.LSTLOAD.KEIL** *<module> <file> [<base>]*

Loading of an KEIL (8051) assembler list file for source text debugging on assembler level. The base address is optional, it is used when the address sections are unknown.

See also

■ [sYmbol.LSTLOAD](#)

Format: **sYmbol.LSTLOAD.MicroWare** *<module>|<program> <file> [<base>]*

Loading of a MICROWARE assembler list file for source text debugging on assembler level. If the base address of the module doesn't fit, the base address will be given as an argument. If a program name is given, the module entry will be generated from the file name of the list file.

Example:

```
Data.LOAD.ROF sps /MAP ; load the module and
sYmbol.LSTload.mw \\SPS sps1.lst 0x1000 ; symbols
; load the source
sYmbol.LSTload.MicroWare \\SPS sps2.lst 0x1200 ; lines for module 1
; load the source
; lines for module 2
```

See also

■ [sYmbol.LSTLOAD](#)

Format: **sYmbol.LSTLOAD.MRI68K** <module>|<program> <file> [<base_address>]

Loading of a MICROTEC assembler list file for source text debugging on assembler level. If the base address of the module doesn't fit, the base address will be given as an argument.

The first comment line of the file (beginning with ***)**) must start in the **first column** of the source text! The debugging can be controlled by the following assembler comments:

***T32-ORG**

To mark lines including ORG statements.

***T32-OFF**

To switch of the source text debugging. The debugging must be switched of for lines containing data statements or definitions (i.e. lines which address column not containing a program address).

***T32-ON**

Reactivation of the debugging function.

See also

■ [sYmbol.LSTLOAD](#)

sYmbol.LSTLOAD.OAK

Load OAK assembler source file

Format: **sYmbol.LSTLOAD.OAK** <module>|<program> <file> [<base>]

Loading of a OAK assembler list file for source text debugging on assembler level.

See also

■ [sYmbol.LSTLOAD](#)

The **sYmbol.MARKER** commands are intended for very advanced users only. The commands are used for fine-tuning the nesting function run-time analysis, typically in conjunction with the **Trace.STATistic.Func** command. Markers can be used to handle special cases for nested trace statistics. Please contact [Lauterbach support](#) before using the **sYmbol.MARKER** commands.

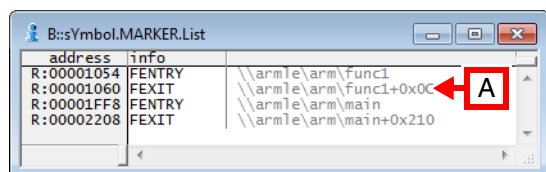
The **sYmbol.MARKER** commands let you create markers for symbols, display them in a list window, delete individual markers, and reset all markers. The following examples are just intended to illustrate these actions.

Example 1:

```
;Display the marker list window
sYmbol.MARKER.List

;Create function entry markers for the functions func1, main
sYmbol.MARKER.Create FENTRY func1
sYmbol.MARKER.Create FENTRY main

;Create function exit markers
sYmbol.MARKER.Create FEXIT sYmbol.EXIT(func1)
sYmbol.MARKER.Create FEXIT sYmbol.EXIT(main)
```



A Double-click a line to open a **List.auto** window, displaying the location of a marker. Note that the marker itself is not visible in the **List.auto** window.

Example 2:

```
;Delete an individual function entry marker
sYmbol.MARKER.Delete func1

;Delete an individual function exit marker
sYmbol.MARKER.Delete sYmbol.EXIT(main)

;Delete all markers
sYmbol.MARKER.RESet
```

See also

■ [sYmbol](#) ■ [sYmbol.MARKER.Create](#) ■ [sYmbol.MARKER.Delete](#) ■ [sYmbol.MARKER.List](#)
 ■ [sYmbol.MARKER.RESet](#) ■ [sYmbol.MARKER.TOUCH](#)

▲ 'Release Information' in 'Legacy Release History'

Format:	sYmbol.MARKER.Create <type> <instruction_address> sYmbol.CREATE.MARKER (deprecated) sYmbol.NEW.MARKER (deprecated)
<type>:	KENTRY KEXIT KBACKDOOR KBEGIN KEND KFROM KTO KLEAVE FENTRY KFENTRY IFENTRY ILEAVE FEXIT FEXITINIT FEXITDOUBLE FEXITCLEANUP FBACKDOOR KFBACKDOOR CLEANUP CLEANUP2 IGNORE IGNOREFROM IGNORETO CALL JUMP BYJUMP TASKSWITCH CORRELATE

Nesting function run-time analysis commands like **Trace.STATistic.Func** process the trace information to reconstruct the function call hierarchy. The focus is put on the transitions between functions. The following events are of interest:

- Function entries/exits
- Task switches
- Entries/exits of interrupt service routines
- Entries/exits of TRAP handlers

Optimization of the OS and the compiler as well as the technology used for the trace generation may disturb the reconstruction of the call hierarchy. As a result, nesting function run-time analysis fails. Markers are a mean to feed TRACE32 with additional information that help to pass blocking points in the analysis.

```
sYmbol.MARKER.Create KENTRY os_prologue      ; mark the address os_prologue
                                           ; as kernel entry point

sYmbol.MARKER.Create KEXIT os_epilogue        ; mark the address os_epilogue
                                           ; as kernel exit point

sYmbol.MARKER.List                           ; list all markers
```

The command **sYmbol.MARKER.List** provides a list of all created markers.

KENTRY/KEXIT/KBACKDOOR Marker

A detailed description of the usage of the markers is given at the command description of [Trace.STATistic.TASKKernel](#) and [Trace.STATistic.TASKFunc](#).

```
sYmbol.MARKER.Create KENTRY os_prologue      ; mark the address
                                              ; os_prologue
                                              ; as kernel entry point

sYmbol.MARKER.Create KEXIT os_epilogue        ; mark the address
                                              ; os_epilogue
                                              ; as kernel exit point

sYmbol.MARKER.List                          ; list all markers
```

KBEGIN/KEND Marker

Mark *<address>* as kernel event. These markers work the same way as **KENTRY** and **KEXIT** but are not nested, i.e. a **KEND** marker closes all previously opened **KBEGIN** markers.

FENTRY/FEXIT/FBACKDOOR/FEXITDOUBLE/FEXITINT Marker

FENTRY <i><address></i>	Mark <i><address></i> as function entry.
FEXIT <i><address></i>	Mark <i><address></i> as function exit.
FBACKDOOR <i><address></i>	Mark <i><address></i> as function entry. TRACE32 ignores this function entry in the trace evaluation if a prior function entry was detected.
FEXITDOUBLE <i><address></i>	Mark <i><address></i> as function exit where a function exits two function levels.
FEXITINT <i><address></i>	Mark <i><address></i> as interrupt exit/return event.

A detailed description of the usage of the markers is given at the command description of [Trace.STATistic.Func](#).

Purpose: Solve issues of trace export technology.

CORRELATE markers allows to assign read/write accesses to instructions.

If trace information is exported for all executed instructions but only for selected read/write accesses, an exact assignment of an read/write access to the load/store operation is not possible in most cases. The read/write access is displayed in the Trace Listing in red before the next exported instruction information (ptrace).

This behavior may disturb the reconstruction of the call hierarchy in the following case: A task switch occurs (write access to variable holding the task ID) and then a function is called, but in the Trace Listing the task switch is displayed after the function call. As a result the function is hooked into the wrong call tree.

More than one **CORRELATE** marker is possible as long as its address is unique between two ptrace packets.

In the example below, the write access to the task identifier (TASK.CONFIG(magic)) and all instructions are exported via a Nexus port (MPC5646).

B::Trace.List List.ADDRESS List.TASK Default								
Setup... Goto... Find... Chart Profile MIPS More Less								
record	run	address	cycle	data	symbol	ti.back		
-00005936		V:400012CA		ptrace	\\im02_bf1x\ossch\OSTaskForceDispatch+0x8A	1.400us		
		V:400012CA		se_cmpi	r3,0x0 ; r3,0			
		V:400012CC		se_beg	0x4000137E			
296		V:4000137E		mfspr	r7,spr273 ; r7,sprg1			
		V:40001382		se_stw	r7,0x8(r31) ; r7,8(r31)			
		V:40001384		mtspr	spr273,r31 ; sprg1,osOldRunning			
299		V:40001388		e_bl	0x40001048 ; OSTaskInternalDispatch			
203		V:40001048		e_stwu	r1,-0x40(r1) ; r1,-64(r1)			
		V:4000104C		se_mflr	r0			
		V:4000104E		e_stmw	r22,0x18(r1) ; r22,24(r1)			
		V:40001052		e_stw	r0,0x44(r1) ; r0,68(r1)			
205		V:40001056		e_lwz	r7,-0x7F68(r13) ; r7,osRunning_(r13)			
		V:4000105A		se_lhz	r30,0x1A(r7) ; r30,26(r7)			
		V:4000105C		e_stb	r30,-0x7F90(r13) ; r30,osOrtiRunning(r13)			
		V:40001060		e_rlwinm	r6,r30,0x0,0x18,0x1F ; r6,r30,0,24,31			
		V:40001064		e_rlwinm	r6,r6,0x1C,0x1A,0x1F ; r6,r6,28,26,31			
		V:40001068		e_ori	r3,r6,0x80 ; r3,r6,128			
		V:4000106C		e_bl	0x400005B8 ; OSSetPID0			
505		V:400005B8		msync				
506		V:400005BC		se_isync				
		V:400005BE		mtpid	r3 ; value			
		V:400005C2		msync				
		V:400005C6		se_isync				
		V:400005C8		seblr				
		TASK = TASK0						
-00005934		D:400059DC		wr-byte	09 \\im02_bf1x\Global\osOrtiRunning	1.060us		
-00005933		V:40001070		ptrace	\\im02_bf1x\ossch\OSTaskInternalDispatch+0x28	0.640us		

...

NEXUS.BTM ON

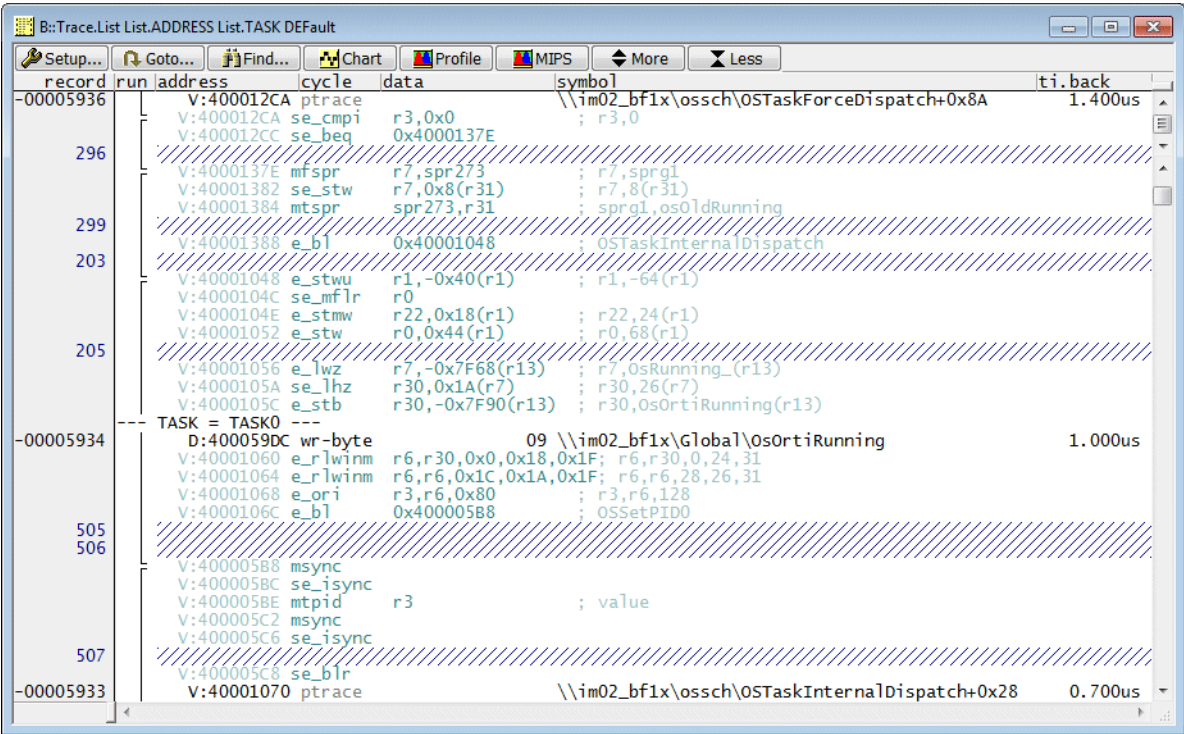
NEXUS.HTM ON

Break.Set TASK.CONFIG(magic) /Write /TraceData

...

For the reconstruction of the call hierarchy and an accurate timing the write access of the task ID has to be assigned exactly to the corresponding instruction. For this purpose a **CORRELATE** marker can be used.

```
sYmbol.MARKER.Create CORRELATE V:4000105C
```



IGNORE / IGNOREFROM / IGNORETO / IGNOREALLFUNC

IGNORE <address>	TRACE32 ignores all instructions with the specified <address>, for nesting function run-time analysis.
IGNOREFROM <start_address> IGNORETO <end_address>	TRACE32 ignores all instructions between <start_address> and <end_address>, for nesting function run-time analysis.
IGNOREALLFUNC	TRACE32 ignores all function entries and all function exits, for nesting function run-time analysis.

See also

- [sYmbol.MARKER](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format: **sYmbol.MARKER.Delete** <address>

Deletes the specified marker. To delete all markers, use [sYmbol.MARKER.RESet](#).

See also

■ [sYmbol.MARKER](#)

sYmbol.MARKER.List

Displays the marker list

Format: **sYmbol.MARKER.List** [<address>]
sYmbol.List.MARKER [<address>] (deprecated)

Opens the **sYmbol.MARKER.List** window, displaying the list of markers created with [sYmbol.MARKER.Create](#).

A detailed description of the usage of the markers is given at the command description of [Trace.STATistic.TASKKernel](#) and [Trace.STATistic.TASKFunc](#).

See also

■ [sYmbol.MARKER](#)

sYmbol.MARKER.RESet

Erase all markers

Format: **sYmbol.MARKER.RESet**

Removes all markers created with [sYmbol.MARKER.Create](#).

See also

■ [sYmbol.MARKER](#)

Format: sYmbol.MARKER.TOUCH [ON | OFF]

Default: ON

Enables/disables the re-processing of the function nesting analysis results after [sYmbol.MARKER.Create](#) commands. This command can be used in order to speed up the marker processing in TRACE32 in case a large number of markers is created.

Example:

```
sYmbol.MARKER.TOUCH OFF
;<group of sYmbol.MARKER.Create commands>
sYmbol.MARKER.TOUCH ON
```

See also

■ [sYmbol.MARKER](#)

Format: sYmbol.MATCH [Exact | Best | Choose]

Defines the behavior when a symbol is not unique.

- | | |
|---------------|--|
| Exact | Refuses any symbols that are not unique. This adjustment is useful for regression test and automatic batch scripts. |
| Best | Open browser window to choose symbol when there is no best match. This is the default. |
| Choose | Opens a symbol browser to choose from one of the symbols. This is useful for choosing between different overloaded methods in C++. |

See also

■ [sYmbol](#) ■ [Data.Find](#)

Format: sYmbol.MEMory

Displays a summary of memory used by the different components of the symbol table. The results are written to the **AREA A000** (display with **AREA** command).

See also

■ [sYmbol](#)

Format:

sYmbol.Modify

This command group allows to modify the symbol table or add additional information to existing symbols and types. See also [sYmbol.RELOCate](#).

See also

- [sYmbol.Modify.Access](#)
 - [sYmbol.Modify.AddressToRange](#)
 - [sYmbol.Modify.ATTRibute](#)
 - [sYmbol.Modify.NAME](#)
 - [sYmbol.Modify.RangeToAddress](#)
 - [sYmbol.Modify.SOURCE](#)
 - [sYmbol.Modify.StaticCOPY](#)
 - [sYmbol.Modify.TYPE](#)

▲ 'Release Information' in 'Legacy Release History'
- [sYmbol.Modify.ADDress](#)
 - [sYmbol.Modify.AlienFunction](#)
 - [sYmbol.Modify.CutFunction](#)
 - [sYmbol.Modify.NAMES](#)
 - [sYmbol.Modify.RangeToFunction](#)
 - [sYmbol.Modify.SplitFunction](#)
 - [sYmbol.Modify.StaticToStack](#)
 - [sYmbol](#)

sYmbol.Modify.Access

Modify access of symbols

Format:

sYmbol.Modify.Access <class>: [<symbol_path>|<range>]

Modifies the memory access class of symbols. The symbol path limits the modification to special symbols of a module or a program. If an address range is given, only the symbols in this range will be modified. The command is useful in combination with the automatic symbol relocation feature, when constants are placed in ROM.

Examples:

```
sYmbol.Modify.Access d:0x1000--0x1fff      ; all symbols in the range get
                                           ; the memory access class d:

sYmbol.Modify.Access p:                    ; all symbols in the 'const'
sYmbol.SECRANGE(const)                    ; section get the memory
                                           ; access class p:
```

See also

- [sYmbol.Modify.ADDress](#)
 - [sYmbol.Modify.AlienFunction](#)
 - [sYmbol.Modify](#)
 - [sYmbol.Modify.NAME](#)
 - [sYmbol.Modify.RangeToAddress](#)
 - [sYmbol.Modify.SOURCE](#)
 - [sYmbol.Modify.StaticCOPY](#)
 - [sYmbol.Modify.TYPE](#)
- [sYmbol.Modify.AddressToRange](#)
 - [sYmbol.Modify.ATTRibute](#)
 - [sYmbol.Modify.CutFunction](#)
 - [sYmbol.Modify.NAMES](#)
 - [sYmbol.Modify.RangeToFunction](#)
 - [sYmbol.Modify.SplitFunction](#)
 - [sYmbol.Modify.StaticToStack](#)

Format:

sYmbol.Modify.ADDress <symbol> <address>|<range>

Modifies the start address or address range of a symbol, module, program or function.

Example:

```
; assign new range to module
sYmbol.Modify.ADDress \mymodule 0x1000--0x1fff
```

See also

- [sYmbol.Modify.Access](#)
 - [sYmbol.Modify.AlienFunction](#)
 - [sYmbol.Modify](#)
 - [sYmbol.Modify.NAME](#)
 - [sYmbol.Modify.RangeToAddress](#)
 - [sYmbol.Modify.SOURCE](#)
 - [sYmbol.Modify.StaticCOPY](#)
 - [sYmbol.Modify.TYPE](#)
- [sYmbol.Modify.AddressToRange](#)
 - [sYmbol.Modify.ATTRibute](#)
 - [sYmbol.Modify.CutFunction](#)
 - [sYmbol.Modify.NAMES](#)
 - [sYmbol.Modify.RangeToFunction](#)
 - [sYmbol.Modify.SplitFunction](#)
 - [sYmbol.Modify.StaticToStack](#)

Format:

sYmbol.Modify.AddressToRange <symbol> |<address>

Extends a single address label into a symbol with an address range. The address range starts at the symbol address and ends at the address of the next symbol minus 1. This command is the opposite of [sYmbol.Modify.RangeToAddress](#).

Example:

```
sYmbol.Modify.AddressToRange mylabel
```

See also

- [sYmbol.Modify.Access](#)
 - [sYmbol.Modify.ADDress](#)
 - [sYmbol.Modify](#)
 - [sYmbol.Modify.SOURCE](#)

Format:

sYmbol.Modify.AlienFunction <symbol> |<range>

Disables frame information for a selected function.

See also

[sYmbol.Modify.Access](#)[sYmbol.Modify.ADDress](#)[sYmbol.Modify](#)[sYmbol.Modify.SOURCE](#)

sYmbol.Modify.ATTRibute

Modify memory attribute

Format:

sYmbol.Modify.ATTRibute <attribute> <symbol>|<range>

Modifies the attribute of the given memory range

Example:

```
; change the memory attribute for the range 0x400--0x4FF to DATA
sYmbol.Modify.ATTRibute DATA 0x400--0x4FF
```

See also

[sYmbol.Modify.Access](#)[sYmbol.Modify.ADDress](#)[sYmbol.Modify](#)[sYmbol.Modify.SOURCE](#)

sYmbol.Modify.CutFunction

Reduce function address information

Format:

sYmbol.Modify.CutFunction <range> | <address>

Reduces the address range information for the defined functions to a single address. This command is used only in very special cases.

Example:

```
sYmbol.List.Function
sYmbol.Modify.CutFunction 0x104c++0xff
```

See also

- [sYmbol.Modify](#)
- [sYmbol.Modify.Access](#)
- [sYmbol.Modify.ADDress](#)
- [sYmbol.Modify.SOURCE](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

sYmbol.Modify.NAME

Rename symbol

Format:

sYmbol.Modify.NAME <symbol_name> <new_name>

Renames symbol. If multiple symbols with the given name are available, an error “ambiguous symbol” is returned. Use in this case [sYmbol.Modify.NAMES](#).

Example:

```
; renames 'vtriplearray' to 'tt'
sYmbol.Modify.NAME vtriplearray tt
```

See also

- [sYmbol.Modify](#)
- [sYmbol.Modify.Access](#)
- [sYmbol.Modify.ADDress](#)
- [sYmbol.Modify.SOURCE](#)

sYmbol.Modify.NAMES

Rename symbols

Format:

sYmbol.Modify.NAMES <symbol_name> <new_name>

Renames all symbols having the name <symbol_name>, except for function locals.

If you need to rename a lot of symbols, consider using [sYmbol.DEOBFUSCATE](#) as it will be much faster.

See also

- [sYmbol.Modify](#)
- [sYmbol.Modify.Access](#)
- [sYmbol.Modify.ADDress](#)
- [sYmbol.Modify.SOURCE](#)
- [sYmbol.DEOBFUSCATE](#)

Format: **sYmbol.Modify.AddressToRange** <symbol> |<address>

Changes an symbol with an address range into a single address label. This command is the opposite of **sYmbol.Modify.AddressToRange**.

See also

[■ sYmbol.Modify](#)[■ sYmbol.Modify.Access](#)[■ sYmbol.Modify.ADDress](#)[■ sYmbol.Modify.SOURCE](#)

sYmbol.Modify.RangeToFunction

Modify address range into function

Format: **sYmbol.Modify.RangeToFunction** <symbol> |<range>

Modifies address range into function.

Example: assembly functions are often represented in the debugging information by the compiler as a single address label. As a consequence, these assembly functions won't be included in the function run-time analysis windows. The commands **sYmbol.Modify.AddressToRange** and **sYmbol.Modify.RangeToFunction** can be used together to change these single address labels into functions.

```
sYmbol.Modify.AddressToRange _divsi3  
sYmbol.Modify.RangeToFunction _divsi3
```

See also

[■ sYmbol.Modify](#)[■ sYmbol.Modify.Access](#)[■ sYmbol.Modify.ADDress](#)[■ sYmbol.Modify.SOURCE](#)

sYmbol.Modify.SOURCE

Define source file

Format: **sYmbol.Modify.SOURCE** <module> <file>

Defines the source file name for a given module. The command can be used when the names or directories of source files have been changed after compilation.

Example:

```
; use source file mod2.c for module 'mod1'
sYmbol.Modify.SOURCE \mod1 ..\src\mod2.c
```

See also

- [sYmbol.Modify.SplitFunction](#)
[sYmbol.Modify.StaticToStack](#)
[sYmbol.Modify.Access](#)
[sYmbol.Modify.AddressToRange](#)
[sYmbol.Modify.ATTRibute](#)
[sYmbol.Modify.NAME](#)
[sYmbol.Modify.RangeToAddress](#)
[sYmbol.Modify.TYPE](#)
- [sYmbol.Modify.StaticCOPY](#)
[sYmbol.Modify](#)
[sYmbol.Modify.ADDress](#)
[sYmbol.Modify.AlienFunction](#)
[sYmbol.Modify.CutFunction](#)
[sYmbol.Modify.NAMES](#)
[sYmbol.Modify.RangeToFunction](#)

sYmbol.Modify.SplitFunction

Split function

Format:

sYmbol.Modify.SplitFunction <label>

Makes two functions out of one function. Takes a label inside the function as a parameter. The second function starts at this label and has the name of the label.

See also

- [sYmbol.Modify.SOURCE](#)
- [sYmbol.Modify](#)
- [sYmbol.Modify.Access](#)
- [sYmbol.Modify.ADDress](#)

sYmbol.Modify.StaticCOPY

Create static copy of local stack variables

Format:

sYmbol.Modify.StaticCOPY <module> <address> <address> [<reg>]

Creates static copy of local stack variables.

See also

- [sYmbol.Modify.SOURCE](#)
- [sYmbol.Modify](#)
- [sYmbol.Modify.Access](#)
- [sYmbol.Modify.ADDress](#)

Format:

sYmbol.Modify.StaticToStack <reg> <address> [<symbol | address range>]

Changes static variables into global stack variables.

See also

[sYmbol.Modify.SOURCE](#)[sYmbol.Modify](#)[sYmbol.Modify.Access](#)[sYmbol.Modify.ADDress](#)

sYmbol.Modify.TYPE

Modify type of symbols

Format:

sYmbol.Modify.TYPE <symbol> <type>

Changes the symbol type.

See also

[sYmbol.Modify](#)[sYmbol.Modify.Access](#)[sYmbol.Modify.ADDress](#)[sYmbol.Modify.SOURCE](#)

▲ 'Release Information' in 'Legacy Release History'

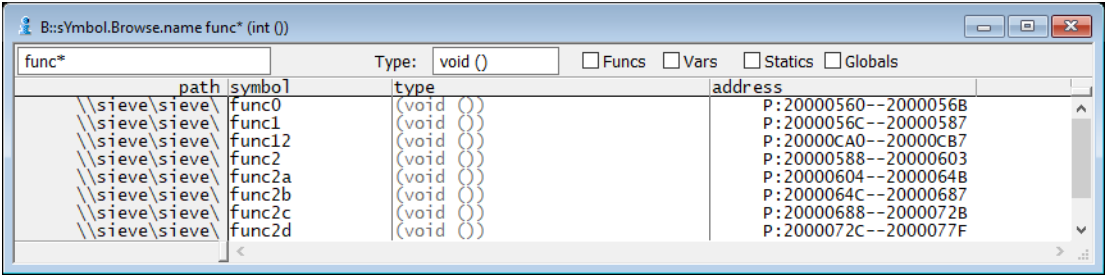
Format:

sYmbol.name [<name_pattern> [<type_pattern>] [/<option>]

<option>:

Click <cmd>
Delete

Displays symbols sorted alphabetically.



Lower and upper case characters are not distinguished. For mangled C++ symbols the search order is based on the function signatures. A complex search function is implemented to find symbol name very fast, if the complete name will be not known. The search patterns are:

- '*' Matches any string, empty strings too.
- '?' Matches any character, but not an empty character.
- ''' Can be used to input special characters like '*' or '?'

If the wildcard '*' is defined for the *type_pattern*, only symbols with HLL type information are extracted. The C++ demangler for the symbols is used, if the pattern contains the characters '(' or '.'.

Examples:

```
sYmbol                                ; displays all symbols

sYmbol *\*                            ; displays all local symbols

sYmbol \*\*                          ; displays all local symbols of global
                                   ; functions and module local symbols

sYmbol \mcc\*                        ; displays all symbols local to module
                                   ; 'mcc'

sYmbol func9\*                      ; displays all symbols local to
                                   ; function 'func9'

sYmbol i                            ; displays all symbols with the name 'i'

sYmbol \mcc\*\i                     ; displays all local symbols in module
                                   ; 'mcc' with the name 'i'
```

<code>sYmbol \m*\f*\i*</code>	<code>; displays all local symbols with the ; symbol name beginning with 'i' in all ; functions with function names ; beginning with 'f' in all modules ; beginning with 'm'</code>
<code>sYmbol * *</code>	<code>; displays all symbols with HLL type ; information</code>
<code>sYmbol * *ptr</code>	<code>; displays all symbols, which have an ; HLL type ; that ends with 'ptr', e.g. 'intptr'</code>
<code>sYmbol * char *</code>	<code>; displays all symbols, which have an ; HLL type ; that begins with the text 'char ', ; e.g. 'char *', 'char [10]', 'char &'</code>
<code>sYmbol * char ""</code>	<code>; displays all symbols with HLL type of ; 'char *'</code>
<code>sYmbol * """"</code>	<code>; displays all symbols with type names, ; that contain a '*'</code>
<code>sYmbol ops::operator*</code>	<code>; displays all operators defined for ; the C++ class 'ops'</code>
<code>sYmbol operator*</code>	<code>; will search for all symbols, ; beginning with the string 'operator' ; NOTE: the demangler is not active, so ; no operators of C++ classes are ; listed!</code>
<code>sYmbol *::operator*(int)</code>	<code>; display all operator of all classes, ; that have only one argument of type ; 'int'</code>

The **Click** option can define a command, that can be executed by a short click with the left mouse button. The characters '*' or '?' can be used as placeholder for the complete name of the symbol. Using the '*' will force the command to be executed without further interaction and without leaving the window. The character '?' will cause the cursor to leave the window and build a command line, that can be modified before entering. The option **Delete** deletes the window after the selection has been made.

Examples: See also [sYmbol.ForEach](#).

```
sYmbol * /Click "Break.Set"           ; will execute the command
                                         ; Break.Set <symbol>

sYmbol * /Click "Break.Set *          ; will execute the command
/Alpha"                               ; Break.Set <symbol> /Alpha

sYmbol * /Click "Var.View ?"          ; will build a command line
                                         ; Var.View <symbol>
                                         ; and leave the symbol window
```

The address based softkeys are available by pressing the left mouse button. Short clicking the left button can execute the command defined by the option **Click**. The default is the command [Data.List](#). Pressing one of the softkeys leaves the window, and builds a command line according to the softkey and symbol.

See also

■ [sYmbol](#)

sYmbol.NAMESPACES

Search symbol in C++ namespace

Format: **sYmbol.NAMESPACES** [*<namespace>*]

This command configures the debugger to search in the specified C++ namespaces for a debug symbol.

If the debug symbol cannot be found in the global namespace and the debug symbol is referred without scope operator (e.g. inside a `namespace {}` section or after a `using <namespace>` declaration, the debugger will search in the namespaces specified by this command.

Example:

```
sYmbol.NAMESPACES std                ; search debug symbols in namespace std if
                                         ; debug symbol not found in current
                                         ; context
```

See also

■ [sYmbol](#)

▲ ['Release Information' in 'Legacy Release History'](#)

Format: **sYmbol.NEW** <name> <address> | <range>

The **sYmbol.NEW** command group allows to create new symbols or modify existing user-defined symbols. The command **sYmbol.CREATE** has the same functionality, but executes faster when multiple symbols are created.

See also

- sYmbol.NEW.ATTRibute

■ sYmbol.NEW.MACRO

■ sYmbol.CREATE
- sYmbol.NEW.Function

■ sYmbol.NEW.Module

■ sYmbol.CREATE.RESet
- sYmbol.NEW.Label

■ sYmbol.NEW.Var

■ sYmbol.Delete
- sYmbol.NEW.LocalVar

■ sYmbol
- ▲ 'Release Information' in 'Legacy Release History'

sYmbol.NEW.ATTRibute

Create user-defined memory attribute

[\[Example\]](#)

Format: **sYmbol.NEW.ATTRibute** <attribute> <start_address | addressrange>

<attribute>: **DEFault**
 <architecture_specific_attributes>

Creates a user-defined memory attribute, e.g. for program code, data code, access width, etc. Memory attributes tell TRACE32 how to interpret memory content. If attributes are missing in the debug information of your symbol file, e.g. an ELF file, you can create the attributes with **sYmbol.NEW.ATTRibute**.

DEFault	Memory content is interpreted based on the current processor state.
<architecture_specific_attributes>	The softkeys below the TRACE32 command line display the available memory attributes. - For a selection of memory attributes for various architectures, see below. - For more information about available memory attributes, refer to the design manual of the respective architecture.

Selection of Memory Attributes for Various Architectures

Memory attributes for the ARM architecture:

ARM	Code of A32, ARM 32-bit instruction set
THUMB	Code of T32, Thumb/Thumb-2/ThumbEE instruction set

AARCH64	Code of A64, ARM 64-bit instruction set
DATA	Data

Memory attributes for the ARC architecture:

CODE	Program code
DATA.Byte	8-bit data
DATA.Word	16-bit data
DATA.Long	32-bit data

Memory attributes for the PowerPC architecture:

FLE	Code of standard PowerPC instruction set
VLE	Code of Variable Length Encoding
DATA	Data

Memory attributes for the TI-TMS320C55x architecture:

BYTE	Tells TRACE32 to interpret code in byte-pointer mode. Corresponds to SYStem.Option.ByteMode BYTE for the specified <i><address_range></i> .
WORD	Tells TRACE32 to interpret code in word-pointer mode.

Memory attributes for the Intel® x86 architecture:

USE16	16-bit mode
USE32	32-bit mode
USE64	64-bit mode

NOTE:	The Intel® Processor Trace does not include any information whether it is in 16, 32, or 64 bit mode. Using the above memory attributes, you can tell TRACE32 how to disassemble correctly.
--------------	--

Example:

```
;open window, displaying the memory attributes per address range
sYmbol.List.ATTRibute

;create an attribute for a data address range without DATA attribute
sYmbol.NEW.ATTRibute DATA D:10004138--10004193

;display all available information about an address including its attr.
sYmbol.INFO                D:10004138

;this code is now interpreted as data
List.Mix                   D:10004138
```

See also

- [sYmbol.NEW](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

sYmbol.NEW.Function

Create user-defined function

Format:

sYmbol.NEW.Function <name> <addressrange>

Creates a new function. The function has no parameters or local variables. It can only be used to define a range for a piece of code (e.g. for performance analysis).

Example:

```
sYmbol.NEW.Function myfunc mylabel1--(mylabel2-1)
```

See also

- [sYmbol.NEW](#)
- [sYmbol.CREATE.Function](#)

Format: **sYmbol.NEW.Label** <name> <address>

Creates a new label. A label is a symbol without type information that refers to a single memory location.

Example:

```
sYmbol.CREATE.Label mylab1 0x1000      ; creates "mylab1" at 1000
sYmbol.CREATE.Label mylab2 0x1010      ; creates "mylab2" at 1010
sYmbol.CREATE.Done                     ; make labels available to program

sYmbol.NEW.Label mylab3 0x1020         ; "mylab3" is available immediately
```

See also

- [sYmbol.NEW](#)
- [sYmbol.CREATE.Label](#)

Format:

sYmbol.NEW.LocalVar <function> <var> <address> | <addressrange> <type>

Creates a user-defined local variable in a user-defined function.

See also

■ [sYmbol.NEW](#)

Format:

sYmbol.NEW.MACRO <name> <contents>

Creates a new macro. The macro can be used like a C-preprocessor macro. Parameters can be supplied in the same way.

Example:

```
; creation and usage of macro MY_NEXT(<arg>)
sYmbol.NEW.MACRO MY_NEXT(p)  ((p)->next)
Var.View MY_NEXT(myvar)
```

See also

■ [sYmbol.NEW](#) ■ [sYmbol.CREATE.MACRO](#)
▲ 'Release Information' in 'Legacy Release History'

Format:

sYmbol.NEW.Module <name> <addressrange>

Defines a new module.

Example:

```
sYmbol.NEW.Module \\new 0x2000--0x2fff
sYmbol.Browse.Module
```

See also

- [sYmbol.NEW](#)
- [sYmbol.CREATE.Module](#)

sYmbol.NEW.Var

Create user-defined variable

Format:

sYmbol.NEW.Var <variable_name> <address> | <addressrange> <type>

Creates a user-defined variable.

Examples:

```
sYmbol.List.Type /Unnamed           ; list all types

sYmbol.NEW.Var my_char 0x1000 char   ; create variable

sYmbol.INFO      my_char             ; display all information
                                   ; about the created variable
```

```
sYmbol.List.Type /Unnamed

sYmbol.NEW.Var my_abc      D:0xa000  struct abc

sYmbol.INFO      my_abc      // structure my_abc of type abc


sYmbol.NEW.Var MyCharArray D:0x1200  char[50]

sYmbol.INFO      MyCharArray // character array with 50 elements


sYmbol.NEW.Var MyPtrArray  D:0x1400  unsigned short *[25]

sYmbol.INFO      MyPtrArray // unsigned short pointer array with 25 el.
```

See also

- [sYmbol.NEW](#)
- [sYmbol.CREATE.Var](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Using the **sYmbol.OVERLAY** command group, TRACE32 can be configured to debug targets that execute and switch between code overlays.

To enable overlay support, use **SYStem.Option.OVERLAY**.

Code overlays are characterized by utilizing the *same address range* in the target for executing *different code* at *different times*. This requires switching between different code segments in physical memory at run-time. As a result, multiple program symbols can refer to the same address range and may refer to program code that currently is **not** present in physical memory. This requires configuration of TRACE32.

See also

- [sYmbol.OVERLAY.AutoID](#)
- [sYmbol.OVERLAY.Create](#)
- [sYmbol.OVERLAY.DETECT](#)
- [sYmbol.OVERLAY.FRIEND](#)
- [sYmbol.OVERLAY.List](#)
- [sYmbol.OVERLAY.RESet](#)
- [sYmbol](#)

sYmbol.OVERLAY.AutoID

Automatically determine overlay IDs

Format:

sYmbol.OVERLAY.AutoID [OVS: | VM:]

Calculates a unique identifier for each overlay currently present in the system.

- OVS: (default)

Read the overlay code from the overlay storage area (OVS).
- VM:

To increase detection speed, TRACE32 can read the overlay code from the debugger VM:. This requires that the code was loaded to the execution area (**/CODESEC**) inside the segmented (**/OVERLAY**) debugger virtual memory (**/VM**) with a command like
Data.LOAD.Elf <file> /OVERLAY /CODESEC /NosYmbol /VM

To detect which overlay is currently active (i.e. present in the execution area), the debugger uses unique identifiers based on the overlays' contents. The command **sYmbol.OVERLAY.AutoID** calculates these identifiers for all currently declared overlay section. **Therefore before using it, all code overlay sections have to be known to TRACE32** (through debug information or by explicit declaration) and the **application code has to be present in memory**.

The unique identifier is often called "magic ID" (making reference to the arbitrary value) and is comprised of a "magic" *ID value* present on a corresponding *ID address*. It is found by comparing the contents of the overlay sections and searching for a pair that uniquely identifies each of them. By default the algorithm reads the overlays from the storage areas. To speed up the process, copy the program to the debugger VM: and use the option VM: (see there for details).

There are two ways for TRACE32 to know about the target program's overlays:

- In case of “Relocation-based Code Overlay Support”, TRACE32 reads information about overlay sections from the ELF file. This requires special build settings and using the option /overlay with [Data.LOAD.Elf <file> /OVERLAY](#)
- When using “File-based Code Overlay Support”, you have to manually declare all sections and source files (more generally: all DWARF-modules) before loading your ELF file. The declaration is done with the command [sYmbol.OVERLAY.Create](#).

NOTE:	To avoid problems with software breakpoints, sYmbol.OVERLAY.AutoID only uses ID addresses that do not correspond to HLL lines in the segment.
--------------	--

NOTE:	OVS stands for “Overlaid Storage area”. Besides being a parameter, OVS: is also a memory class specifier for accessing the overlay sections at their so-called <i>storage area</i> : the address it is stored at before being copied to an <i>execution area</i> (the address at which the overlay is executed).
--------------	--

See also

■ [sYmbol.OVERLAY](#)

Format:

sYmbol.OVERLAY.Create <execution_addr_range | overlay_segment_id>,
[<id_address>], [<id_value>], <elf_section_name>, [<dwarf_module>],
[<storage_address>]

Declares a code overlay section and its corresponding modules.

Typically this command is only used for “File-based Code Overlay Support” i.e. when the overlays are contained in different executable files (e.g. ELF files). In this case you have to **declare all sections and source files** (more generally: all DWARF-modules) **before loading the executable files** so that the load command (e.g. via [Data.LOAD.Elf <file> /OVERLAY](#)) can copy the object code to the corresponding OVS: memory.

In case of “Relocation-based Code Overlay Support”, manual declaration of the sections is not required, because they are declared automatically based on information contained in the ELF file when it is loaded using the option **/OVERLAY** ([Data.LOAD.Elf <file> /OVERLAY](#)). Despite this **sYmbol.OVERLAY.Create** can be useful e.g. in order to *define an overlay segment ID* for all code sections.

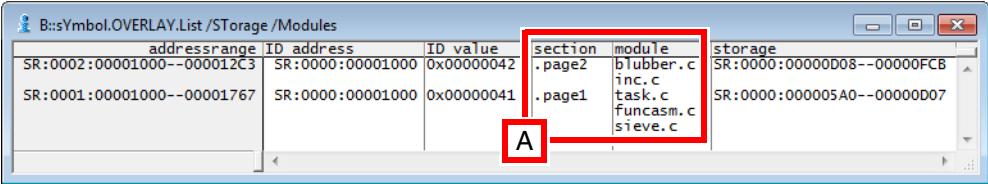
<execution_addr_range> or <overlay_segment_id>	Address range where the code overlay section gets executed. You have to specifies also one unique segment ID for all sections which belong to one overlay page (overlay pages overlap each other). e.g. P:0x42:0x1000-1fff In most cases it is enough just to specify the segment ID (here 0x42), since the debugger can normally find the execution addresses in the ELF section table by using the <elf_section_name>.
<id_address>, <id_value>	The (<id_address> x <id_value>) pair specifies a “magic” ID that uniquely identifies the overlay section when it is present at its execution address. For details regarding the ID see sYmbol.OVERLAY.AutoId which allows auto-detecting the ID.
<elf_section_name>	Unique name for the overlay code section as used in your ELF file. e.g. ".page1" The section names are normally defined in your linker script used for building your ELF file. In TRACE32 you can view the sections of a loaded ELF with command sYmbol.List.SECTION . From a system shell (cmd.exe / bshell) you can view the available section names e.g. with the GNU command <code>readelf -S <ElfFile></code> When using overlay sections from different ELF files you can prefix the section name with the ELF file name e.g. " <code>//myprog/.page1</code> " NOTE: By running the sYmbol.OVERLAY.Create command with the same arguments again except for a new <id_value>, you can assign more than one ID to the same <elf_section_name>. See example 4 .

<code><dwarf_module></code> (source file)	Required only for “File-based Code Overlay Support” to tell the debugger which source files belong to which overlay section (this can only be detected from the ELF when it contains relocation information). <code><dwarf_module></code> is usually the name of a source file (e.g. a C-/C++ file). You can omit this parameter, if your ELF file was linked with compiler support for code overlays e.g. “-Wl,--emit-relocs” (GCC) or “--emit-debug-overlay-section” (ARM RealView). NOTE: By running the sYmbol.OVERLAY.Create command with the same arguments again except for a new <code><dwarf_module></code>, you can assign more than one module to the same <code><elf_section_name></code>. See example 1.
<code><storage_address></code>	Specifies the start of the address range where the code section is stored before it gets copied to its execution memory space. You can usually omit this parameter, as the <code><storage_address></code> is normally auto-detected when loading you ELF file.

Examples

Example 1: File-based code overlay support (single-ELF)

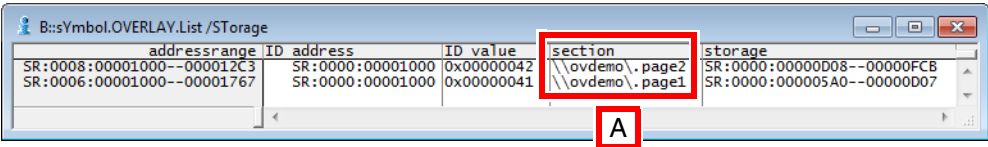
```
sYmbol.RESet
SYStem.Option.OVERLAY ON
sYmbol.OVERLAY.Create 1,,,".page1","task.c"
sYmbol.OVERLAY.Create 1,,,".page1","funcasm.c"
sYmbol.OVERLAY.Create 1,,,".page1","sieve.c"
sYmbol.OVERLAY.Create 2,,,".page2","blubber.c"
sYmbol.OVERLAY.Create 2,,,".page2","inc.c"
Data.LOAD.Elf ovdemo.elf /OVERLAY /NoClear /Include /NOFRAME
sYmbol.OVERLAY.AutoID
sYmbol.OVERLAY.List /STorage /Modules
```



A Modules per code overlay section

Example 2: Relocation-based code overlay support (single-ELF)

```
sYmbol.RESet
SYStem.Option.OVERLAY ON
Data.LOAD.Elf ovdemo.elf /OVERLAY /Include
sYmbol.OVERLAY.AutoID
sYmbol.OVERLAY.List /STorage
```

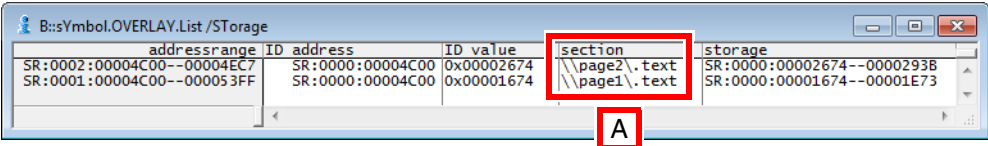


addressrange	ID	address	ID value	section	storage
SR:0008:00001000--000012C3		SR:0000:00001000	0x00000042	\\ovdemo\page2	SR:0000:00000D08--00000FCB
SR:0006:00001000--00001767		SR:0000:00001000	0x00000041	\\ovdemo\page1	SR:0000:000005A0--00000D07

A The file ovdemo.elf has two code overlay sections named .page1 and .page2

Example 3: Relocation-based multi-ELF code overlay support

```
sYmbol.RESet
SYStem.Option.OVERLAY ON
sYmbol.OVERLAY.Create 1,,,"\\page1\text"
sYmbol.OVERLAY.Create 2,,,"\\page2\text"
Data.LOAD.Elf page1.elf /NoClear /NoRegister /OVERLAY
Data.LOAD.Elf page2.elf /NoClear /NoRegister /OVERLAY
sYmbol.OVERLAY.AutoID
sYmbol.OVERLAY.List /STorage
```



addressrange	ID	address	ID value	section	storage
SR:0002:00004C00--00004EC7		SR:0000:00004C00	0x00002674	\\page2\text	SR:0000:00002674--0000293B
SR:0001:00004C00--000053FF		SR:0000:00004C00	0x00001674	\\page1\text	SR:0000:00001674--00001E73

A The code overlays in the two *.elf files (page1.elf and page2.elf) have the same section name: .text

Example 4: Relocation-based code overlay support (single-ELF) with more than one ID per ELF section

```
sYmbol.RESet
SYStem.Option.OVERLAY ON
sYmbol.OVERLAY.Create 1,D:0x1000,0x10,".page1"

;assign the ID 0x20 to the elf section named ".page2"
sYmbol.OVERLAY.Create 2,D:0x1000,0x20,".page2"

;additionally assign the ID 0x22 to the same elf section ".page2"
sYmbol.OVERLAY.Create 2,D:0x1000,0x22,".page2"

Data.LOAD.Elf ovdemo.elf /OVERLAY /Include /NoClear
sYmbol.OVERLAY.List /STorage
```

A

addressrange	ID	address	ID value	section	storage
SR:0002:00001000--000012C3	D:0000:00001000	0x20 0x22		.page2	SR:0000:00000D08--00000FCB
SR:0001:00001000--00001767	D:0000:00001000	0x00000010		.page1	SR:0000:000005A0--00000D07

A The code overlay section `.page2` has two IDs: `0x20` and `0x22`.

See also

■ [sYmbol.OVERLAY](#)

Format: sYmbol.OVERLAY.DETECT [ON | OFF]

Default: ON

Executing **sYmbol.OVERLAY.DETECT ON** performs an immediate update of the status (active vs. not active) of all registered code overlays. The detection mechanism uses the ID address and ID value shown in [sYmbol.OVERLAY.List](#).

The optional parameter controls the automatic update of the status of overlay pages when the target executes code that may have changed it.

To use the command, first enable overlays via **SYStem.Option.OVERLAY**.

See also

■ [sYmbol.OVERLAY](#)

sYmbol.OVERLAY.FRIEND

Declare a friend overlay segment

[build no. 46380 - DVD 08/2013]

Format: sYmbol.OVERLAY.FRIEND <original_overlay_id> <friend_overlay_id>

NOTE: Only relevant for decoding ARM ETM trace data in the context of overlays.

The command is relevant for tracing the switches between overlays using ARM ETM.

For an *original overlay segment*, the command declares (or deletes) the so-called “*friend*” *overlay segment*. The friend overlay segment (a better term would be *subsequent overlay*) is a segment that usually is executed *after* the original segment. Declaring a friend overlay gives the debugger a hint which allows to improve the accuracy of decoding trace data corresponding to the switch between the segments. See the box “background” on the following page for more information.

For each overlay one friend overlay can be declared. For deleting the friend overlay use the command with a friend ID of 0.

In addition to “normal” overlay segments, friend overlays can also be declared for the “common area” of all (“permanent”) non-overlay code by using the segment id 0.

The configuration of overlays segments and their respective friend overlays is displayed by [sYmbol.OVERLAY.List /FRiend..](#)

<code><original_overlay_id></code>	ID of the original overlay segment (16-bit integer) If <i>segment</i> is zero, a friend segment will be assigned for the global memory space.
<code><friend_overlay_id></code>	ID of the friend overlay segment i.e. typically the subsequent overlay segment (16-bit integer) If <i>friend</i> is zero, the friend will be deleted.

NOTE:

To have an effect, the command needs to be executed before activating RTS via [RTS.ON](#).

Background Information

If the trace decoder encounters an opcode on an address not contained in the current overlay segment this causes a problem. Therefore - when a friend overlay is declared - the trace decoder also checks the friend overlay for this address. If the address is found, the opcode is considered to be part of the friend overlay otherwise as part of the “common area” and thus as not belonging to an overlay.

The previous case can occur when the target switches to a new overlay segment. Switches between overlays are typically reported by an ownership trace message which is created by a special opcode writing to a dedicated register (ownership register).

If additional opcodes are executed in the context of the *old* overlay segment (e.g. opcodes residing on an address of the old overlay segment) the ownership message appears “too early” causing problems decoding the trace data of subsequent opcodes (e.g. RTS return-from-subroutine). If the opcode is executed in the context of the *new* overlay segment, the message is sent “too late” because the opcode was executed before the ownership trace message was created.

The special case where the subsequent overlay *does* include the address in question but contains a *different opcode* on the address cannot be handled correctly. It may cause a flow error which however in practice is rather infrequent.

See also

■ [sYmbol.OVERLAY](#)

Format:

sYmbol.OVERLAY.List [/Modules | /Storage | /FRriend]

Shows the declared code overlays and the corresponding symbol information set with [sYmbol.OVERLAY.Create](#) and/or [sYmbol.OVERLAY.AutoID](#). The code overlays currently present in memory (“active code overlays”) are highlighted.

Modules	Shows the DWARF modules related to the overlay section. This is only useful in case of “File-based Code Overlay Support “, since the column is empty in case of “Relocation-based Code Overlay Support”.
Storage	Shows the memory region from where the code overlays should be loaded.
FRriend	Shows the friends of each memory segment. See sYmbol.OVERLAY.FRIEND

See also

[sYmbol.OVERLAY](#)

Format:

sYmbol.OVERLAY.RESet

Clears the complete table of declared overlay sections.
The table of declared overlay sections can be shown with [sYmbol.OVERLAY.List](#). Entries can be added with [sYmbol.OVERLAY.Create](#).

See also

[sYmbol.OVERLAY](#)

The Greenhills compiler can add extra assembler instructions to the code, e.g. to function entries and function exits. This instrumentation code generates SFT software trace messages during program execution.

TRACE32 can extract the symbol information about the extra assembler instructions from the loaded application file. Using the **sYmbol.PATCH** command group, you can list the extracted symbol information, as well as enable and disable it.

For PRACTICE demo scripts (*.cmm), see `~/demo/rh850/etc/sft_trace/`

See also

- [sYmbol.PATCH.Disable](#)
- [sYmbol.PATCH.Enable](#)
- [sYmbol.PATCH.List](#)
- [sYmbol](#)

sYmbol.PATCH.Disable

Disable instrumentation code

Format: **sYmbol.PATCH.List** [*<address>* | *<range>*]

Disables instrumentation code at the specified *<address>* or within the specified address *<range>*. Executing the command without an argument disables all instrumentation codes.

The [sYmbol.PATCH.List](#) window displays an overview of all symbols in TRACE32 representing the enabled and disabled instrumentation codes.

Example: See [sYmbol.PATCH.List](#).

See also

- [sYmbol.PATCH](#)

sYmbol.PATCH.Enable

Enable instrumentation code

Format: **sYmbol.PATCH.Enable** [*<address>* | *<range>*]

Enables instrumentation code at the specified *<address>* or within the specified address *<range>*. Executing the command without an argument enables all instrumentation codes.

The [sYmbol.PATCH.List](#) window displays an overview of all symbols in TRACE32 representing the enabled and disabled instrumentation codes.

Example: See [sYmbol.PATCH.List](#).

See also

■ [sYmbol.PATCH](#)

sYmbol.PATCH.List

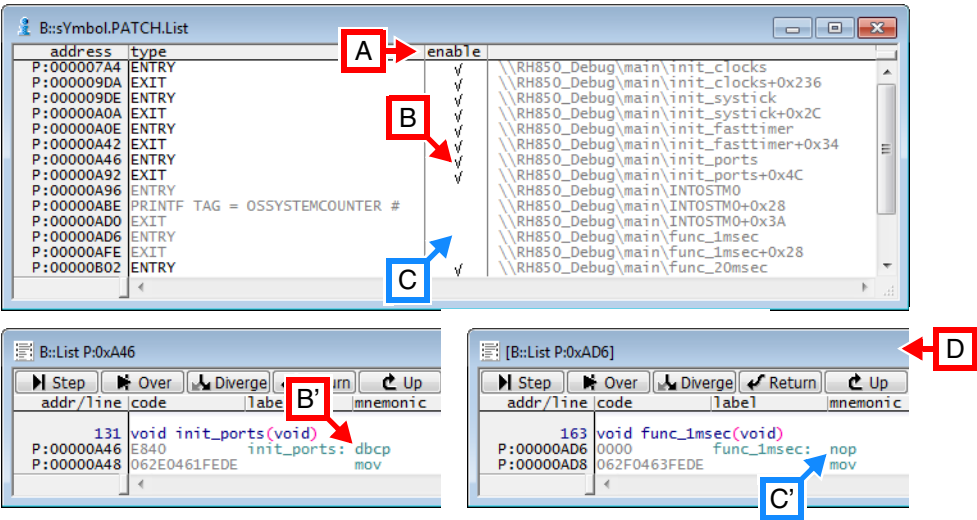
Display STF-symbol information

Format:

sYmbol.PATCH.List [<address>]

sYmbol.List.PATCH [<address>] (alias)

Lists the extracted symbol information of the instrumentation code in the **sYmbol.PATCH.List** window.



- A The **enable** column displays the status of the instrumentation codes. Clicking a cell in the **enable** column enables or disables the instrumentation codes:
- B A checkmark indicates that the original instrumentation code is active, see [B’].
- C No checkmark indicates that the original instrumentation code is patched, e.g. by NOP instructions, see [C’].
- D Double-clicking in the **sYmbol.PATCH.List** window opens a corresponding **List** window.

Example 1: This script deactivates function entries and exits within a specified range.

```
sYmbol.PATCH.List
;
sYmbol.PATCH.DISable INTOSTM0--sYmbol.END(func_1msec)
```

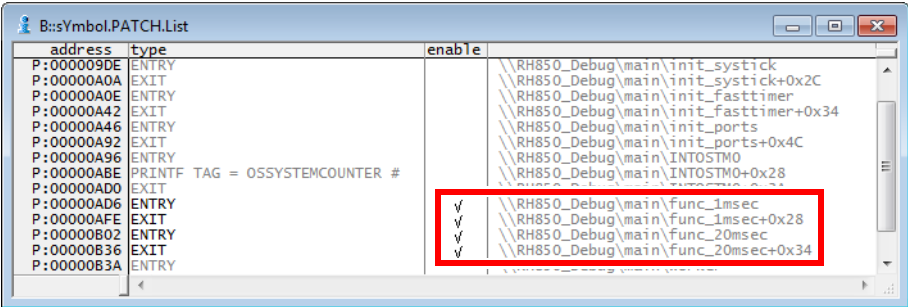
Example 2: This script uses the **sYmbol.ForEach** command to enable and disable symbol groups that match the desired name pattern.

```
sYmbol.PATCH.List

;1st step: disable all functions
sYmbol.ForEach "sYmbol.PATCH.DISable sYmbol.RANGE(""****")" ** /Function

;2nd step: enable the functions that match the name pattern, here *sec*
sYmbol.ForEach "sYmbol.PATCH.ENABLE sYmbol.RANGE(""****")" *sec* /Function
```

Result of example 2:



address	type	enable	
P:000009DE	ENTRY		\\RH850_Debug\\main\\init_systick
P:00000A0A	EXIT		\\RH850_Debug\\main\\init_systick+0x2C
P:00000A0E	ENTRY		\\RH850_Debug\\main\\init_fasttimer
P:00000A42	EXIT		\\RH850_Debug\\main\\init_fasttimer+0x34
P:00000A46	ENTRY		\\RH850_Debug\\main\\init_ports
P:00000A92	EXIT		\\RH850_Debug\\main\\init_ports+0x4C
P:00000A96	ENTRY		\\RH850_Debug\\main\\INTOSTM0
P:00000ABE	PRINTF TAG = OSYSTEMCOUNTER #		\\RH850_Debug\\main\\INTOSTM0+0x28
P:00000AD0	EXIT		\\RH850_Debug\\main\\func_1msec
P:00000AD6	ENTRY	✓	\\RH850_Debug\\main\\func_1msec
P:00000AFE	EXIT	✓	\\RH850_Debug\\main\\func_1msec+0x28
P:00000B02	ENTRY	✓	\\RH850_Debug\\main\\func_20msec
P:00000B36	EXIT	✓	\\RH850_Debug\\main\\func_20msec+0x34
P:00000B3A	ENTRY		

See also

■ [sYmbol.PATCH](#)

Format: **sYmbol.POINTER** [*<framepointer>*] [*<staticpointer>*]

Determination of frame pointer and static pointer registers. The frame pointer addresses the variables located on the stack, the static pointer addresses the static variables with position independent data only. During loading an HLL program the registers are automatically preset with the values according to the compilers.

See also

■ [sYmbol](#)

sYmbol.POSTFIX

Set symbol postfix

Format: **sYmbol.POSTFIX** [*<character>*]

Allows to define a postfix character. If for a symbol name used in a PRACTICE command no applicable symbol is found in the debug information, the postfix character is appended to the symbol name and the search is repeated.

The use of a postfix makes sense when using a compiler, which appends a special sign behind every symbol (for example MarkWilliams C).

See also

■ [sYmbol.PREFIX](#)

■ [sYmbol](#)

sYmbol.PREFIX

Set symbol prefix

Format: **sYmbol.PREFIX** [*<character>*]

A sign may be defined as a prefix. If during entry of a symbol no applicable drag-in can be found, the prefix will be appended in front of the symbol and the search will begin once more. The use of a prefix makes useful when using a compiler, which adds a special sign in front of every symbol (for example Microtec Pascal/C).

See also

■ [sYmbol.POSTFIX](#)

■ [sYmbol](#)

Format:

sYmbol.RELOCate <class>:<offset>] [<symbol_path>|<range>]

The command group **sYmbol.RELOCate** is outdated (excluding the command **sYmbol.RELOCate.shift**) and is nowadays only used for OS9-aware debugging (“**OS Awareness Manual OS-9**” (rtos_os9.pdf)).

The command **Data.LOAD Elf <file> /RELOC ...** provides greater flexibility when symbols need to be relocated.

See also

- [■ sYmbol.RELOCate.Auto](#)[■ sYmbol.RELOCate.Base](#)[■ sYmbol.RELOCate.List](#)[■ sYmbol.RELOCate.Magic](#)
- [■ sYmbol.RELOCate.Passive](#)[■ sYmbol.RELOCate.shift](#)[■ sYmbol](#)
- [▲ 'Release Information' in 'Legacy Release History'](#)

sYmbol.RELOCate.Auto

Control automatic relocation

Format:

sYmbol.RELOCate.Auto [ON | OFF]

The command **sYmbol.RELOCate.Auto** is outdated and is nowadays only used for OS9-aware debugging. Please refer to “**OS Awareness Manual OS-9**” (rtos_os9.pdf) for more information on this command.

See also

- [■ sYmbol.RELOCate](#)[■ sYmbol.RELOCate.Base](#)[■ sYmbol.RELOCate.List](#)[■ sYmbol.RELOCate.Magic](#)
- [■ sYmbol.RELOCate.Passive](#)[■ sYmbol.RELOCate.shift](#)

Format:

sYmbol.RELOCate.Base <class>:<base> [<symbol_path>|<range>]

The command **sYmbol.RELOCate.Base** is outdated and is nowadays only used for OS9-aware debugging. Please refer to “**OS Awareness Manual OS-9**” (rtos_os9.pdf) for more information on this command.

- See also
- sYmbol.RELOCate

■ sYmbol.RELOCate.Passive

■ sYmbol.RELOCate.Auto

■ sYmbol.RELOCate.shift

■ sYmbol.RELOCate.List

■ sYmbol.RELOCate.Magic

sYmbol.RELOCate.List

List relocation info

Format:

sYmbol.RELOCate.List

The command **sYmbol.RELOCate.List** is outdated and is nowadays only used for OS9-aware debugging. Please refer to “**OS Awareness Manual OS-9**” (rtos_os9.pdf) for more information on this command.

- See also
- sYmbol.RELOCate

■ sYmbol.RELOCate.Passive

■ sYmbol.RELOCate.Auto

■ sYmbol.RELOCate.shift

■ sYmbol.RELOCate.Base

■ sYmbol.RELOCate.Magic

sYmbol.RELOCate.Magic

Define program magic number

Format:

sYmbol.RELOCate.Magic <program_magic> [<symbol_path>|<range>]

The command **sYmbol.RELOCate.Magic** is outdated and is nowadays only used for OS9-aware debugging. Please refer to “**OS Awareness Manual OS-9**” (rtos_os9.pdf) for more information on this command.

- See also
- sYmbol.RELOCate

■ sYmbol.RELOCate.Passive

■ sYmbol.RELOCate.Auto

■ sYmbol.RELOCate.shift

■ sYmbol.RELOCate.Base

■ sYmbol.RELOCate.List

Format:

sYmbol.RELOCate.Passive <class>:<base>

The command **sYmbol.RELOCate.Passive** is outdated and nowadays only used for OS9-aware debugging. Please refer to “**OS Awareness Manual OS-9**” (rtos_os9.pdf) for more information on this command.

See also

■ sYmbol.RELOCate

■ sYmbol.RELOCate.Magic

■ sYmbol.RELOCate.Auto

■ sYmbol.RELOCate.shift

■ sYmbol.RELOCate.Base

■ sYmbol.RELOCate.List

sYmbol.RELOCate.shift

Relocate symbols

Format:

sYmbol.RELOCate.shift <class>:<offset> [<symbol_path>|<range>]

Manually relocates symbols. The symbol path limits the relocation to special symbols of a module or a program. If an address range is given, only the symbols in this range will be relocated. Relocation is always relative to the current address of the symbol. When the memory access classes of some symbols are wrong, they can be changed by the **sYmbol.Modify.Access** command.

Examples:

```
; relocate all program symbols by 12240H
sYmbol.RELOCate.shift P:0x12240
```

```
; relocate all data symbols of module 'main'
sYmbol.RELOCate.shift D:0x1000 \main
```

```
; relocate all data symbols in the given range
sYmbol.RELOCate.shift D:0x1000 0x40000--0x4ffff
```

See also

■ sYmbol.RELOCate

■ sYmbol.RELOCate.Magic

■ sYmbol.RELOCate.Auto

■ sYmbol.RELOCate.Passive

■ sYmbol.RELOCate.Base

■ sYmbol.RELOCate.List

▲ 'Release Information' in 'Legacy Release History'

Format:	sYmbol.RESet
---------	---------------------

All symbols and search paths for source files are cleared.

See also

- [sYmbol](#)

Format:

sYmbol.SourceBeautify [ON | OFF]

- ON
- Beautifies the indentations of HLL lines displayed in the [List](#) windows.
The source file itself is not touched.
- OFF
- Displays the HLL lines as formatted in the source file.

NOTE:

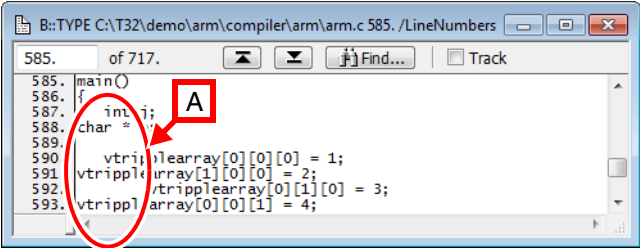
The command takes only effect if it is set to **ON** or **OFF** *before* executing the [Data.LOAD.<file_format>](#) command.

Example:

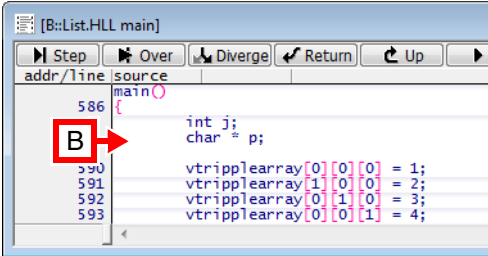
```
sYmbol.SourceBeautify ON

;load ELF file plus associated source file. For demo purposes, the *.c
;file contains some malformed HLL lines.
Data.LOAD.ELF armle.axf /StripPATH /LowerPATH

List.HLL main ;refer to [B]
```



A Source file (*.c) with indentation issues.



B Indentation issues fixed in TRACE32.

See also

- sYmbol

Format:	sYmbol.SourceCONVert <mode>
<mode>:	OFF EUC-JP

Converts the HLL source information from EUC (Extended UNIX Code) to JP for Windows.

See also

- [sYmbol](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format: **sYmbol.SourceLOAD** [*<module>*] [*<file>*]

By default an HLL source file is only loaded by TRACE32 when the contents of the HLL source file is required during debugging. The command **sYmbol.SourceLOAD** loads the HLL source file for the defined module on user request. The HLL source line information from the absolute object file must be loaded before using this command. If the command **sYmbol.SourceLOAD** is used without any parameter, all HLL source files are loaded.

The command **sYmbol.SourceLOAD** allows also the specification of a new HLL source file instead of the one listed in **sYmbol.List.SOURCE**. This can be useful, if the name or directory of the source file has been changed after compilation.

Examples:

```
; load all source files
Data.LOAD.COFF arm.abs
sYmbol.SourceLOAD
```

```
; load the source file for the module \\thumble\arm
Data.LOAD.COFF arm.abs
sYmbol.List.SOURCE
sYmbol.SourceLOAD \\thumble\arm
```

```
; load the source file NewArm.C for the module \\thumble\arm
Data.LOAD.COFF arm.abs
sYmbol.SourceLoad \\thumble\arm G:\ARM\etc\Test\NewArm.C
```

See also

■ [sYmbol](#)

Format:

sYmbol.SourcePATH [+ | - | -- | <directory>] ... (deprecated)

Adds or removes directories to the path used for searching the source files accessed by TRACE32. The current search order can be displayed by the command [sYmbol.SourcePATH.List](#). The list can be saved with the [STore](#) command and the **SPATH** item.

+	Use the '+' to add a directory to the path.
-	Use the '-' sign to remove a path from the list.
--	Two minus signs '--' clear the whole list like sYmbol.RESet .

Examples:

```
sYmbol.SourcePATH + c:\source\proj_1           ; add search directories
sYmbol.SourcePATH + c:\mike\source\proj_1
...
sYmbol.SourcePATH - c:\source\proj_1           ; remove search directory
```

See also

- [sYmbol.SourcePATH.Delete](#)
 - [sYmbol.SourcePATH.List](#)
 - [sYmbol.SourcePATH.Set](#)
 - [sYmbol.SourcePATH.SetCache](#)
 - [sYmbol.SourcePATH.SetCachedDirCache](#)
 - [sYmbol.SourcePATH.SetDir](#)
 - [sYmbol.SourcePATH.SetMasterDir](#)
 - [sYmbol.SourcePATH.SetRecurseDirCache](#)
 - [sYmbol.SourcePATH.Translate](#)
 - [sYmbol.SourcePATH.UP](#)
 - [sYmbol](#)
- [sYmbol.SourcePATH.DOWN](#)
 - [sYmbol.SourcePATH.RESet](#)
 - [sYmbol.SourcePATH.SetBaseDir](#)
 - [sYmbol.SourcePATH.SetCachedDir](#)
 - [sYmbol.SourcePATH.SetCachedDirIgnoreCache](#)
 - [sYmbol.SourcePATH.SetDynamicDir](#)
 - [sYmbol.SourcePATH.SetRecurseDir](#)
 - [sYmbol.SourcePATH.SetRecurseDirIgnoreCase](#)
 - [sYmbol.SourcePATH.TranslateSUBpath](#)
 - [sYmbol.SourcePATH.Verbose](#)

sYmbol.SourcePATH.Delete

Delete path from search list

Format:

sYmbol.SourcePATH.Delete <directory>

Removes the specified directory from the search path.

See also

- [sYmbol.SourcePATH](#)

Format:

sYmbol.SourcePATH.DOWN <directory>

Internal TRACE32 command. The specified directory becomes the last in the defined search order.

See also

■ [sYmbol.SourcePATH](#)

sYmbol.SourcePATH.List

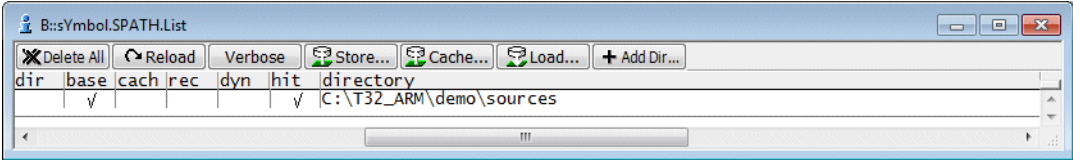
List source search paths

[\[Examples\]](#)

Format:

sYmbol.SourcePATH.List
sYmbol.List.SPATH (deprecated)

Lists defined search paths and their attributes.

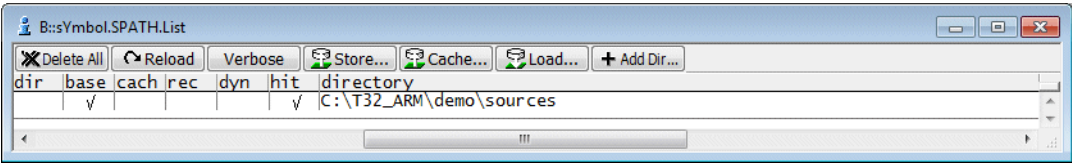


Description of Toolbar Buttons in the sYmbol.SourcePATH.List Window

Delete All	Reset all source path settings. (command sYmbol.SourcePATH.RESet)
Reload	Reload all loaded source files. (command sYmbol.SourceRELOAD)
Verbose	Enable/disable search details in the TRACE32 Message Area. (command sYmbol.SourcePATH.Verbose ON OFF)
Store...	Save source path search setting to <file>. (command STORe <file> SPATH)
Cache...	Save source path search setting plus list of all cached files to <file>. (Command STORe <file> SPATHCACHE)
Load...	Load source path settings from <file>. (command DO <file>)
AddDir	Add directory as base directory and as direct directory to search path list. (command sYmbol.SourcePATH <dir>)

Example for a base directory:

```
; Define directory as base for relative paths
sYmbol.SourcePATH.SetBaseDir C:\T32_ARM\demo\sources
```



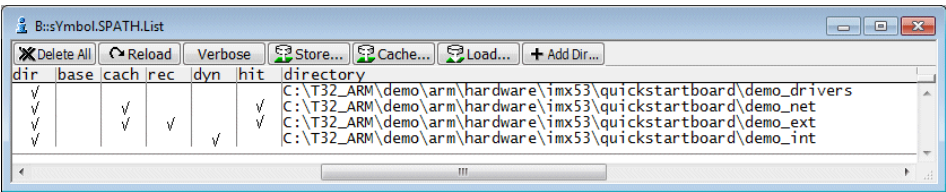
Example for direct directories:

```
; directory is direct search path
sYmbol.SourcePATH.SetDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_drivers

; directory is direct search path, all source files found in the defined
; directory are cached by TRACE32
sYmbol.SourcePATH.SetCachedDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_net

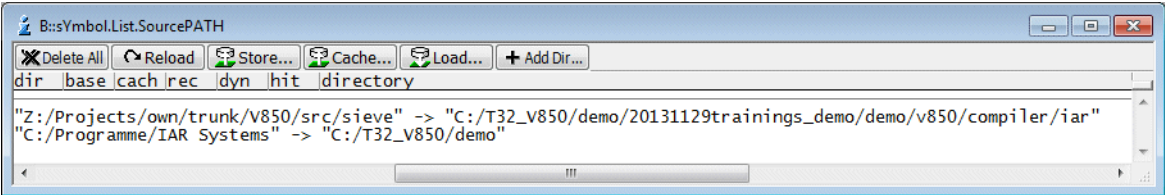
; directory is direct search path, the directory and all its
; sub-directories are used as search path
sYmbol.SourcePATH.SetRecurseDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_ext

; directory is a direct search path, if the source file was found in
; this directory, this directory will become the first to be searched
; in
sYmbol.SourcePATH.SetDynamicDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_int
```



Example for translated paths:

```
sYmbol.SourcePATH.Translate "Z:/Projects/own/trunk/V850/src/sieve"  
"C:/T32_V850/demo/20131129trainings_demo/demo/v850/compiler/iar"  
  
sYmbol.SourcePATH.Translate "C:/Programme/IAR Systems"  
"C:/T32_V850/demo"  
  
sYmbol.List.SourcePATH
```



See also

- [sYmbol.SourcePATH](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

sYmbol.SourcePATH.RESet

Reset search path configuration

Format:

sYmbol.SourcePATH.RESet

Resets the search path configuration.

See also

- [sYmbol.SourcePATH](#)

Format: **sYmbol.SourcePATH.Set** <directory>

This command combines the commands:

- [sYmbol.SourcePATH.SetDir](#)
- [sYmbol.SourcePATH.SetBaseDir](#)

See also

- [sYmbol.SourcePATH](#)

Format:

sYmbol.SourcePATH.SetBaseDir <directory>

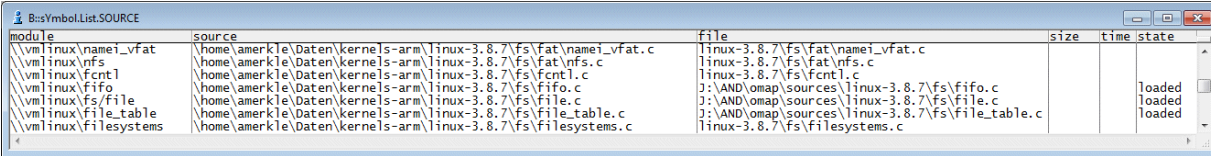
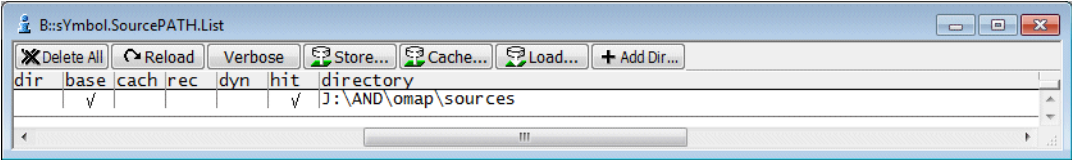
Defines directory as base for relative paths.

```
; load object file vmlinux and cut the following from the source paths:
; start of source path til end of "kernels-arm"
Data.LOAD.Elf ~~~~/vmlinux /StripPART "kernels-arm"

sYmbol.SourcePATH.SetBaseDir J:\AND\omap\sources

sYmbol.SourcePATH.List

sYmbol.List.SOURCE
```



source	Source path provided by executable.
file	Source path as modified by the Data.LOAD command or source path from which the file was loaded.

See also

- [sYmbol.SourcePATH](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format: sYmbol.SourcePATH.SetCache <file>

Internal TRACE32 software command, not of interest for TRACE32 users.

See also

- sYmbol.SourcePATH

sYmbol.SourcePATH.SetCachedDir

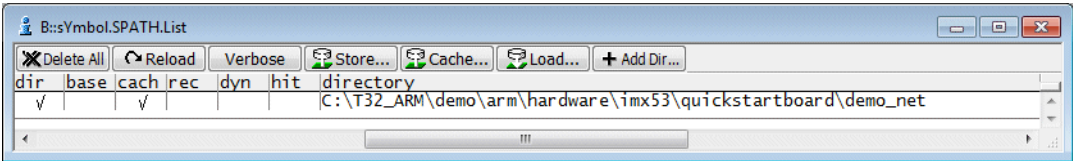
Cache direct search path directory

Format: sYmbol.SourcePATH.SetCachedDir {<directory>}

All source files found in the defined directory are cached by TRACE32.

Example:

```
sYmbol.SourcePATH.SetCachedDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_net
```



STOre CacheCon SPATHCACHE ; generate a script for fast recaching

See also

- sYmbol.SourcePATH

Format: **sYmbol.SourcePATH.SetCachedDirCache** <directory> ...

Internal TRACE32 software command, not of interest for TRACE32 users.

See also

■ [sYmbol.SourcePATH](#)

sYmbol.SourcePATH.SetCachedDirIgnoreCache Cache direct search path

Format: **sYmbol.SourcePATH.SetCachedDirIgnoreCase** {<directory>}

All source files found in the defined directory are cached by TRACE32 ignoring lowercase/uppercase.

See also

■ [sYmbol.SourcePATH](#)

Format:

sYmbol.SourcePATH.SetDir <directory>

The source files are directly searched in the defined directories.

Examples:

```
Data.LOAD.Elf armle.axf           ; load elf file

sYmbol.List.SOURCE               ; display path information for
                                ; source files
```

module	source	file	size	time	state
demo.sieve	C:\work_space\SVN\demo\demo\arm\hardware\tpc4357\sieve.c	C:\work_space\SVN\demo\demo\arm\hardware\tpc4357\sieve.c			error

Source path provided by Elf file

error indicates that the source file was not found under the provided path

```
; define directory as direct search path
sYmbol.SourcePATH.SetDir C:\T32_ARM\demo\arm\compiler\arm

sYmbol.SourcePATH.List           ; display search path defined by
                                ; user

sYmbol.List.SOURCE               ; display path information for
                                ; source files
```

B:sYmbol.SourcePATH.List						
[Delete All] [Reload] [Verbose] [Store...] [Cache...] [Load...] [+ Add Dir...]						
dir	base	cach	rec	dyn	hit	directory
✓					✓	C:\T32_ARM\demo\arm\compiler\arm

module	source	file	size	time	state
demo.sieve	C:\work_space\SVN\demo\demo\arm\hardware\tpc4357\sieve.c	C:\T32_ARM\demo\arm\compiler\arm\sieve.c			loaded

Source path from which the file was loaded

See also

- sYmbol.SourcePATH

Format: **sYmbol.SourcePATH.SetDynamicDir** {<directory>}

The search order defined by this command is dynamically changed. The directory in which the last searched source file was found becomes the first directory in which the debugger searches for the next source file.

Example:

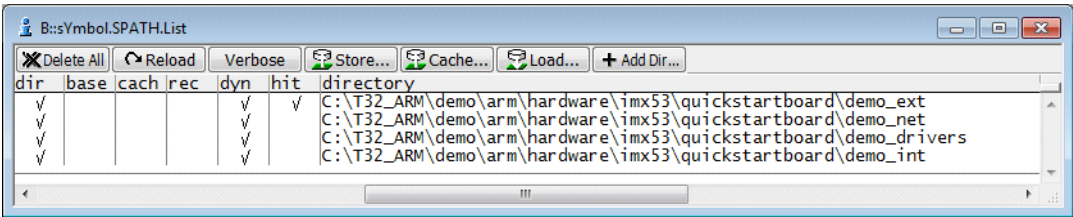
```
sYmbol.SourcePATH.SetDynamicDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_net

sYmbol.SourcePATH.SetDynamicDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_drivers

sYmbol.SourcePATH.SetDynamicDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_ext

sYmbol.SourcePATH.SetDynamicDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard\demo_int

sYmbol.SPATH.List
```



See also

- [sYmbol.SourcePATH](#)

Format: **sYmbol.SourcePATH.SetMasterDir** *<directory>*

If the command **STOre** *<file>* **SPATHCACHE** is used, all file names are only saved relative to the defined directory.

Example:

```
Data.LOAD G:\AND\compiler\xc\bt800ip.iee  
  
sYmbol.SourcePATH.SetCachedDir G:\AND\compiler\xc\System  
  
sYmbol.SourcePATH.SetMasterDir G:\AND\compiler  
  
STOre cache.cmm SPATHCACHE
```

```
// And Thu May 27 16:40:51 2004  
  
B::  
  
sYmbol.SourcePATH.SetCachedDirCache "xc\System"  
  
sYmbol.SourcePATH.SetCache "xc\System"  
(  
    "ad_cond.c"  
    "adsubs.c"  
    "bt_27.c"  
    "bt_27.h"  
    "bt_28.c"  
    "bt_flsh.c"  
)  
  
ENDDO
```

See also

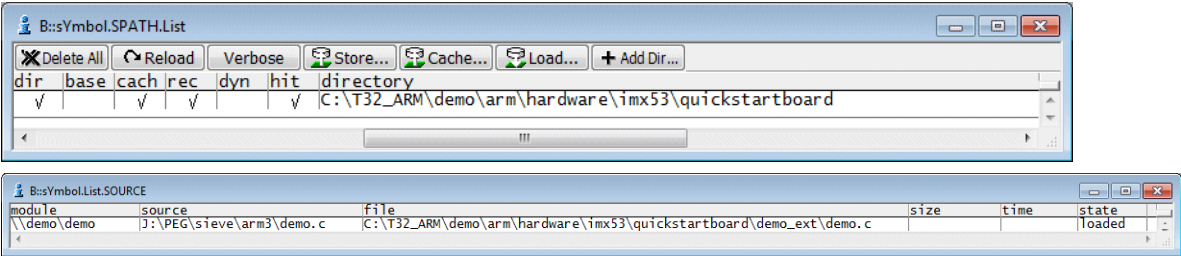
■ [sYmbol.SourcePATH](#)

Format: sYmbol.SourcePATH.SetRecurseDir {<directory>}

Use the defined directory and all subdirectories as search path.

Example:

```
sYmbol.SourcePATH.SetRecurseDir
C:\T32_ARM\demo\arm\hardware\imx53\quickstartboard
```



See also

■ [sYmbol.SourcePATH](#)

sYmbol.SourcePATH.SetRecurseDirCache

Internal use only

Format: sYmbol.SourcePATH.SetRecurseDirCache <directory>...

Internal TRACE32 software command, not of interest for TRACE32 users.

See also

■ [sYmbol.SourcePATH](#)

Format: **sYmbol.SourcePATH.SetRecurseDirIgnoreCase** {<directory>}

Use the defined directory and all subdirectories as search path. Lowercase/uppercase is ignored

See also

■ [sYmbol.SourcePATH](#)

sYmbol.SourcePATH.Translate

Replace part of the source path

Format: **sYmbol.SourcePATH.Translate** <original_path> <new_path>

Replaces <original_path> in source path with <new_path>. Use case: The code was built on a remote machine. Thus the base path must be replaced.

Example:

B:sYmbol.List.SOURCE

module	source	file
\\sieve\sieve	C:/Programme/IAR Systems/Embedded Workbench 6.5/v850/LIB/d185nn0.h	C:\Programme\
\\sieve\sieve	Z:/Projects/own/trunk/V850/src/sieve/sieve.c	Z:\Projects\o

file	size	time	state
C:\Programme\IAR Systems\Embedded Workbench 6.5\v850\LIB\d185nn0.h	13	50729A8	
Z:\Projects\own\trunk\V850\src\sieve\sieve.c	805	51DC0DF	

```
Data.LOAD.Elf sieve.elf

sYmbol.List.SOURCE

sYmbol.SourcePATH.Translate "Z:/Projects/own/trunk/V850/src/sieve"
"C:/T32_V850/demo/20131129trainings_demo/demo/v850/compiler/iar"

sYmbol.SourcePATH.Translate "C:/Programme/IAR Systems"
"C:/T32_V850/demo"

sYmbol.List.SourcePATH
```

B:sYmbol.List.SourcePATH

Delete All | Reload | Store... | Cache... | Load... | + Add Dir...

dir base | cach | rec | dyn | hit | directory

"Z:/Projects/own/trunk/V850/src/sieve" -> "C:/T32_V850/demo/20131129trainings_demo/demo/v850/compiler/iar"
"C:/Programme/IAR Systems" -> "C:/T32_V850/demo"

B:sYmbol.List.SOURCE

module	source	file
\\sieve\sieve	C:/Programme/IAR Systems/Embedded Workbench 6.5/v850/LIB/d185nn0.h	C:\Programme\
\\sieve\sieve	Z:/Projects/own/trunk/V850/src/sieve/sieve.c	C:\T32_V850\d

file	size	time	state
C:\Programme\IAR Systems\Embedded Workbench 6.5\v850\LIB\d185nn0.h	13	50729A8	
C:\T32_V850\demo\20131129trainings_demo\demo\v850\compiler\iar\sieve.c	805	51DC0DF	loaded

See also

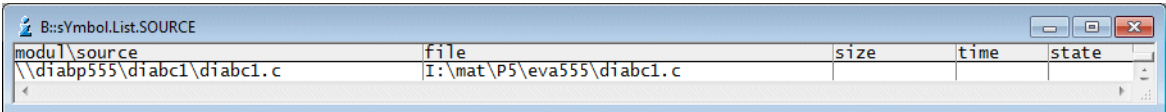
- [sYmbol.SourcePATH](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format:

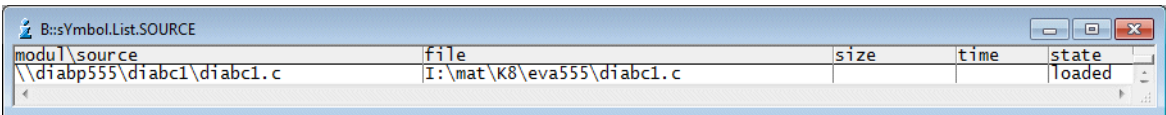
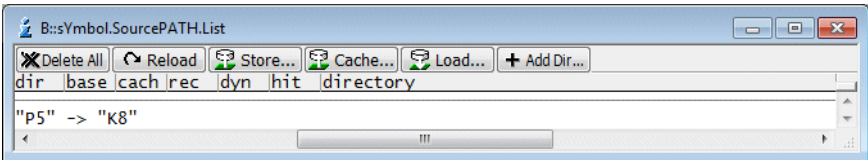
sYmbol.SourcePATH.TranslateSUBpath <original_sub_path>
<new_sub_path>

Replaces <original_sub_path> in source path with <new_sub_path>. Use case: Only a single folder is different.

Example:



```
sYmbol.SourcePATH.TranslateSUBpath "P5" "K8"
```



See also

- sYmbol.SourcePATH

Format:

sYmbol.SourcePATH.UP <directory>

Internal TRACE32 command. The defined directory becomes the first in the search path order.

See also

- sYmbol.SourcePATH

Format:

sYmbol.SourcePATH.Verbose ON | OFF

Default: OFF.

OFF	No details about the source file search are given.
ON	Details about the source file search are displayed in the TRACE32 message AREA.view window, if a source file is loaded by TRACE32.

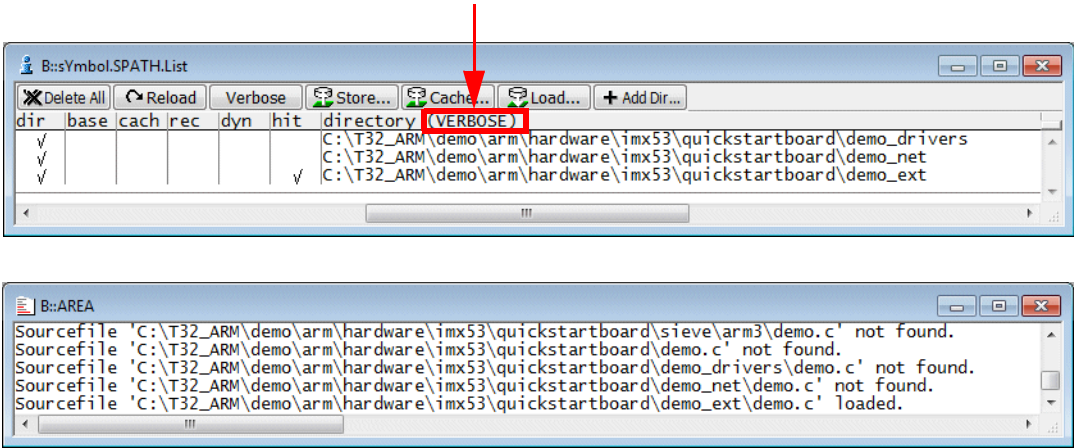
Example:

```
AREA.view                                ; open TRACE32 message AREA window
                                         ; to display source file search
                                         ; details

sYmbol.SourcePATH.Verbose ON

List                                     ; display source listing
```

Verbose ON is indicated in the **sYmbol.SourcePATH.List** window



See also

- [sYmbol.SourcePATH](#)

Format:

sYmbol.SourceRELOAD

Invalidates all loaded source files and marks them for reload. This can be useful when the source search path has been changed to reload the source modules that came from the old source path.

See also

■ [sYmbol](#)

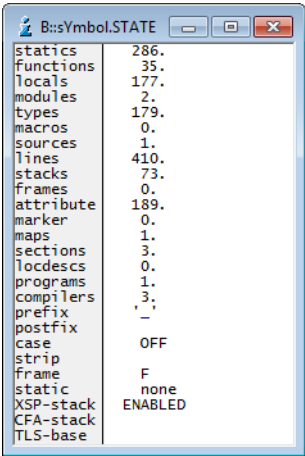
sYmbol.STATE

Display statistic

Format:

sYmbol.STATE

Displays the size of the symbol tables and the global settings. See also [sYmbol.MEMory](#).



See also

- [sYmbol](#)
- [sYmbol.Browse.sYmbol](#)
- [sYmbol.INFO](#)
- [sYmbol.List](#)
- [sYmbol.STATE\(\)](#)
- ▲ ['The Symbol Database' in 'Training Source Level Debugging'](#)

Format: sYmbol.STRIP [<length>]

Cuts symbols to the specified length. This option is useful, when the compiler has a limited symbol length.

Example:

```
sYmbol.STRIP 8.  
Data.List Main_Function                   ; will display 'Main_Fun'
```

See also

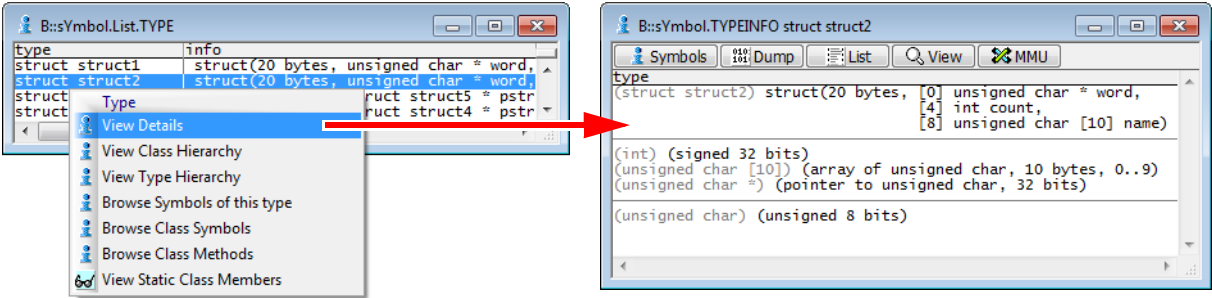
■ [sYmbol](#)

sYmbol.TYPEINFO

Display information about a specific data type

Format: sYmbol.TYPEINFO <data_type>

Displays information about a specific data type. Alternatively, right-click a data type in a [sYmbol.List.Type](#) window, and then select **View Details**, or double-click a data type in that window.

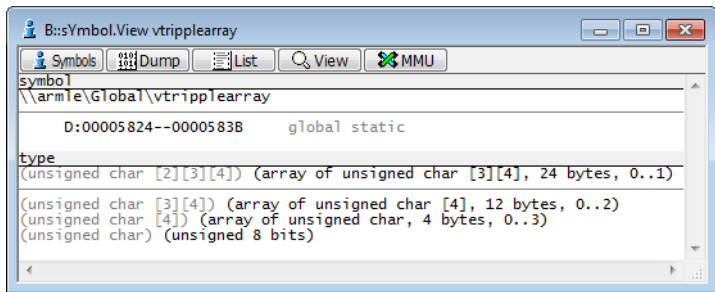


See also

■ [sYmbol](#)

Format: **sYmbol.View** <name> [/Track]

Same as command [sYmbol.Info](#).



See also

- [sYmbol](#)

SYnch Synchronization mechanisms between different TRACE32 systems

See also

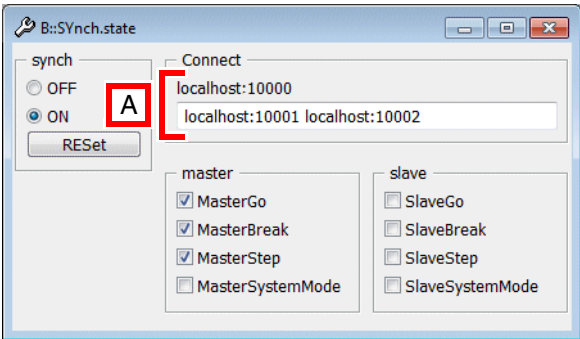
- | | | | |
|--|-------------------------------------|-----------------------------------|---|
| ■ SYnch.Connect | ■ SYnch.MasterBreak | ■ SYnch.MasterGo | ■ SYnch.MasterStep |
| ■ SYnch.MasterSystemMode | ■ SYnch.OFF | ■ SYnch.ON | ■ SYnch.RESet |
| ■ SYnch.SlaveBreak | ■ SYnch.SlaveGo | ■ SYnch.SlaveStep | ■ SYnch.SlaveSystemMode |
| ■ SYnch.state | ■ SYnch.XTrack | ■ TargetSystem | ■ InterCom |

Overview SYnch

For AMP [multicore debugging](#) and [multiprocessor debugging](#), two or more TRACE32 PowerView instances need to be started. The command group **SYnch** allows to establish a connection between different TRACE32 instances for the following purposes:

- To establish a start/stop synchronization between the cores/processors controlled by different TRACE32 instances.
- To allow concurrent assembler single steps between the cores/processors controlled by different TRACE32 instances.
- To allow synchronous system mode changes between the cores/processors controlled by different TRACE32 instances.
- To get a time synchronization between trace information in different TRACE32 instances. This requires that the trace information uses a common time base, e.g. global TRACE32 timestamp or global chip timestamps.

For configuration of the synchronization mechanisms, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **SYnch.state** window. See also prerequisite **[A]** below.



A Prerequisite: You have set up an InterCom system using the [InterCom](#) commands, which allows to exchange data between different TRACE32 systems.

Format 1:	SYnch.Connect [<i><instances></i> ...]
<i><instances></i> :	<i><intercom_name></i> ALL OTHERS <i><name_pattern></i> [<i><host></i> :] <i><port></i>
Format 2:	SYnch.Connect [<i><host></i> :] <i><port></i> ...

Format 1: Establishes connections via the **InterCom** system to other TRACE32 PowerView instances that are connected to same PowerDebug hardware module or the same MCI Server (PBI=MCISERVER in the config.t32 file).

Format 2: Establishes connections to other TRACE32 instances by using the **InterCom** system.

Format 1 and Format 2: **SYnch.ON** is automatically set when the command **SYnch.Connect** establish the connections.

(without parameters) Format 1, Format 2	If the command is used without parameters, it disconnects the TRACE32 PowerView instances.
<i><intercom_name></i> Format 1	InterCom name of a TRACE32 instance. Names can be assigned to TRACE32 instances with the InterCom.NAME command. The InterCom.execute command supports the wildcards * and ? in InterCom names.
<i><name_pattern></i> Format 1	The SYnch.Connect command supports the wildcards * and ? in InterCom names. For example: cluster1.*
ALL Format 1	All known TRACE32 instances.
OTHERS Format 1	ALL except SELF .
<i><host></i> : <i><port></i> Format 1, Format 2	Name of the host and the port number.

Example 1 for Format 2: This script line connects two instances based on their host names and port numbers.

```
SYnch.Connect localhost:20001 localhost:20002
```

Example 2 for Format 1 and Format 2: This script line disconnects the TRACE32 instances.

```
SYnch.Connect
```

Example 3 for Format 1: All instances are connected except the one where this command is executed.

```
SYnch.Connect OTHERS
```

Example 4 for Format 1: The command **SYnch.Connect OTHERS** is executed on all TRACE32 instances.

```
InterCom.execute ALL SYnch.Connect OTHERS
```

See also

- [SYnch](#)
- [SYnch.state](#)
- [InterCom.ENABLE](#)
- [InterCom.NAME](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format:	SYnch.MasterBreak [ON OFF]
---------	-------------------------------------

Default: OFF

SYnch.MasterBreak and **SYnch.SlaveBreak** can be freely programmed for the connected TRACE32 instances. But the break switch/cross trigger unit in the multicore chip might not provide resources to program all links. TRACE32 adjust the programming of the links to the available resources in this case.

ON	Invite other TRACE32 instances to stop synchronously. Cores in a multicore chip can be stopped instruction-accurate if the chip provides a break switch/cross trigger unit. Otherwise software-controlled stop synchronization is used. Software-controlled synchronization stop the cores/processors one by one with a minimum delay of 1.ms.
OFF	No invitation for a synchronous stop is broadcast to other TRACE32 instances.

See also

- [SYnch](#)
- [SYnch.state](#)

Format:SYnch.MasterGo [ON | OFF]

OFF (default)	No invitation for a synchronous start is broadcast to other TRACE32 instances.
ON	Invite other TRACE32 instances to start synchronously. Cores in a multicore chip can start synchronously if this is supported by the chip. Otherwise software-controlled start synchronization is used. software-controlled synchronization starts the cores/processors one by one with a minimum delay of 1.ms.

See also

- [SYnch](#)
- [SYnch.state](#)

SYnch.MasterStep

Invite other TRACE32 to Asm step synchronously

Format:SYnch.MasterStep [ON | OFF]

OFF (default)	No invitation for concurrent assembler single stepping is broadcast to other TRACE32 instances.
ON	Invite other TRACE32 instances to perform concurrent assembler single stepping. HLL single stepping is regarded as a synchronous start.

See also

- [SYnch](#)
- [SYnch.state](#)

Format:

SYnch.MasterSystemMode [ON | OFF]

OFF (default)	No invitation for synchronous mode changing is broadcast to other TRACE32 instances.
ON	Invite other TRACE32 instances to perform system mode changes synchronously. System mode changes are typically performed by one of the following commands: SYStem.Up , SYStem.Mode <mode> commands and SYStem.RESetTarget .

See also

■ [SYnch](#)

■ [SYnch.state](#)

SYnch.OFF

Disable connection mechanism

Format:

SYnch.OFF

Disables the software component that allows a TRACE32 instance to connect to other instances.

See also

■ [SYnch](#)

■ [SYnch.state](#)

SYnch.ON

Enable connection mechanism

Format:

SYnch.ON

Enables the software component that allows a TRACE32 instance to connect to other instances.

See also

■ [SYnch](#)

■ [SYnch.state](#)

Format:	SYnch.RESet
---------	--------------------

Resets the **SYnch** mechanism to its default settings.

See also

■ [SYnch](#)

■ [SYnch.state](#)

SYnch.SlaveBreak

Synchronize with stop in connected TRACE32

Format:	SYnch.SlaveBreak [ON OFF]
---------	------------------------------------

OFF	Don't synchronize with the stop of the program execution in a connected TRACE32 instance.
ON	Synchronize with the stop of the program execution in a connected TRACE32 instance. Cores in a multicore chip can be stopped instruction-accurate if the chip provides a break switch/cross trigger unit. Otherwise software-controlled stop synchronization is used. Software-controlled synchronization stop the cores/processors one by one with a minimum delay of 1.ms.

See also

■ [SYnch](#)

■ [SYnch.state](#)

Format:

SYnch.SlaveGo [ON | OFF]

OFF	Don't synchronize with the start of the program execution in a connected TRACE32 instance.
ON	Synchronize with the start of the program execution in a connected TRACE32 instance. Cores in a multicore chip can start synchronously if this is supported by the chip. Otherwise software-controlled start synchronization is used. software-controlled synchronization starts the cores/processors one by one with a minimum delay of 1.ms.

See also

- [SYnch](#)
- [SYnch.state](#)

Format:

SYnch.SlaveStep [ON | OFF]

OFF	Don't synchronize with assembler single steps in a connected TRACE32 instance.
ON	Synchronize with assembler single steps in a connected TRACE32 instance.

See also

- [SYnch](#)
- [SYnch.state](#)

Format:

SYnch.SlaveSystemMode [ON | OFF]

OFF	Don't synchronize with system mode changes in connected TRACE32 instances.
ON	Synchronize with system mode changes in connected TRACE32 instances.

See also

- [SYnch](#)
- [SYnch.state](#)

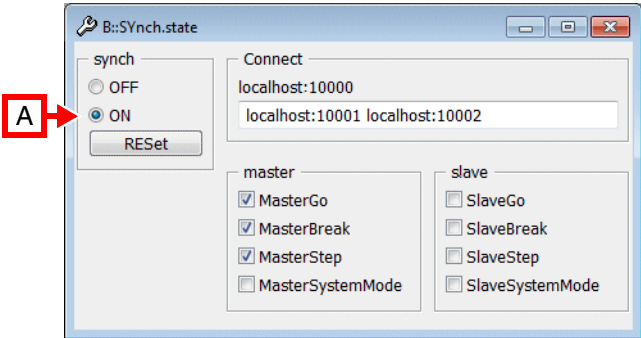
SYnch.state

Display current SYnch settings

Format:

SYnch.state

Displays the current setup of the **SYnch** mechanism.



A For descriptions of the commands in the **SYnch.state** window, please refer to the **SYnch.*** commands in this chapter. **Example:** For information about **ON**, see [SYnch.ON](#).

See also

- [SYnch](#)
- [SYnch.MasterStep](#)
- [SYnch.RESet](#)
- [SYnch.SlaveSystemMode](#)
- [SYnch.Connect](#)
- [SYnch.MasterSystemMode](#)
- [SYnch.SlaveBreak](#)
- [SYnch.XTrack](#)
- [SYnch.MasterBreak](#)
- [SYnch.OFF](#)
- [SYnch.SlaveGo](#)
- [SYnch.MasterGo](#)
- [SYnch.ON](#)
- [SYnch.SlaveStep](#)

Format 1:	SYnch.XTrack [<instances> ...] <trace>.XTrack <intercom_name> (deprecated)
<instances>:	<intercom_name> ALL OTHERS <name_pattern>
Format 2:	SYnch.XTrack [<host>:<port> ...

Establishes a time synchronization between trace information in different TRACE32 instances via the **InterCom** system. This requires that the trace information uses a common time base e.g. global TRACE32 time stamps or global chip timestamps.

Format 1: Establishes a time synchronization to other TRACE32 PowerView instances that are connected to same PowerDebug hardware module or the same MCI Server (PBI=MCISERVER in the config.t32 file).

Format 2: Establishes a time synchronization to other TRACE32 PowerView instances in general.

(without parameters) Format 1, Format 2	If the command is used without parameters, it disconnects the TRACE32 PowerView instances.
<intercom_name> Format 1	InterCom name of a TRACE32 instance. Names can be assigned to TRACE32 instances with the InterCom.NAME command. The InterCom.execute command supports the wildcards * and ? in InterCom names.
<name_pattern> Format 1	The SYnch.Connect command supports of the wildcards * and ? in InterCom names. For example: cluster1.*
ALL Format 1	All known TRACE32 instances.
OTHERS Format 1	ALL except SELF .
<host>:<port> Format 2	Name of the host and the port number.

Example 1 for Format 1:

```
; all instances are connected to each other
InterCom.execute ALL SYnch.XTrack OTHERS

; trace commands in one TRACE32 instance
Trace.List /Track
Trace.Chart.sYmbol /ZoomTrack
```

Example 2 for Format 2:

```
; connect the instance where SYnch.XTrack is executed with the other two
; instances 'localhost:20001' and 'localhost:20002'
SYnch.XTrack localhost:20001 localhost:20002

; trace commands in one TRACE32 instance
Trace.List /Track
Trace.Chart.sYmbol /ZoomTrack
```

See also

■ [SYnch](#)

■ [SYnch.state](#)

The **SYStem** commands are used for setting the operating modes of the system. In addition, they are used to define all those parameters which remain valid after stopping the emulation. The subcommands of the **SYStem** command group are highly target-dependent; for details, check the [Processor Architecture Manual](#) for your target system.

In general, the configuration commands (e.g. [SYStem.Option](#)) should be used before the emulation system is activated with the **SYStem.Mode** or **SYStem.Up** command. Changing configuration options while the system is up may cause unpredictable behavior.

Format:

SYStem.BdmClock *<rate>* (deprecated)
Use [SYStem.JtagClock](#) instead.

<rate>:

2 | 4 | 8 | 16 | *<fixed>*

<fixed>:

1000. ... 5000000.

Either the divided CPU frequency is used as the BDM clock or a fixed clock rate. The fixed clock rate must be used if the operation frequency is very slow or if the clock line is not available. The default is a fixed rate of 1 MHz.

See also

- [SYStem.state](#)

Format: **SYStem.BREAKTIMEOUT** *<time>*

The **Break.direct** command performs a break request and waits for 3.s for the target to stop, otherwise an “emulation running” error is reported. The command **SYStem.BREAKTIMEOUT** can be use to specify a different timeout.

See also

■ [SYStem.state](#)

Format:

SYStem.CADICommand *<command>*

Sends the *<command>* as string to the target. *<command>* is accepted without ' '. The command is send via CADI function 'CADIXfaceBypass()'. If an answer is provided, it will be displayed in [AREA.view](#) beginning with "Answer: ".

See also

- [SYStem.state](#)

Virtual targets only: CADI

The **SYSystem.CADlconfig** command group is used to define CADI-specific setups for debugging and tracing.

See also

- [SYSystem.CADlconfig.ExecSwOnly](#)
 - [SYSystem.CADlconfig.SpecRegDefine](#)
 - [SYSystem.CADlconfig.Traceconfig](#)
 - [SYSystem.IRISconfig](#)
- [SYSystem.CADlconfig.RemoteServer](#)
 - [SYSystem.CADlconfig.SpecRegsOnly](#)
 - [SYSystem.CADlconfig.TraceCore](#)
 - [SYSystem.state](#)

SYSystem.CADlconfig.ExecSwOnly

Filter on executing software capability

Format:

SYSystem.CADlconfig.ExecSwOnly ON | OFF

Default = ON.

ON	TRACE32 connects just to CADI instances which can execute software (CADI internal flag .executesSoftware is set).
OFF	TRACE32 connects to any core or instance via CADI, regardless its capability of executing software.

See also

- [SYSystem.CADlconfig](#)

SYSystem.CADlconfig.RemoteServer

Define connection to CADI server

Virtual targets only: CADI

Format:

SYSystem.CADlconfig.RemoteServer [<ip> <port>]

Informs TRACE32 how to connect to the CADI server for debugging purposes. If this command is omitted from your start-up script, then TRACE32 assumes that the virtual target (including the CADI server) and TRACE32 are running on the same machine.

Without arguments: Resets the connection to a configuration where TRACE32 and the virtual target are assumed to be running on the same machine (`localhost`).

With arguments: Defines a connection to the CADI server on a remote computer. Ensure that your CADI server allows remote connections.

<ip>	IP address or host name of the remote computer where the virtual target is running.
<port>	Parameter Type: Decimal value . TCP/IP port of the CADI server.

Figure 1 of 2: The red line illustrates the connection that is defined as a remote connection using the **SYStem.CADIconfig.RemoteServer** command.

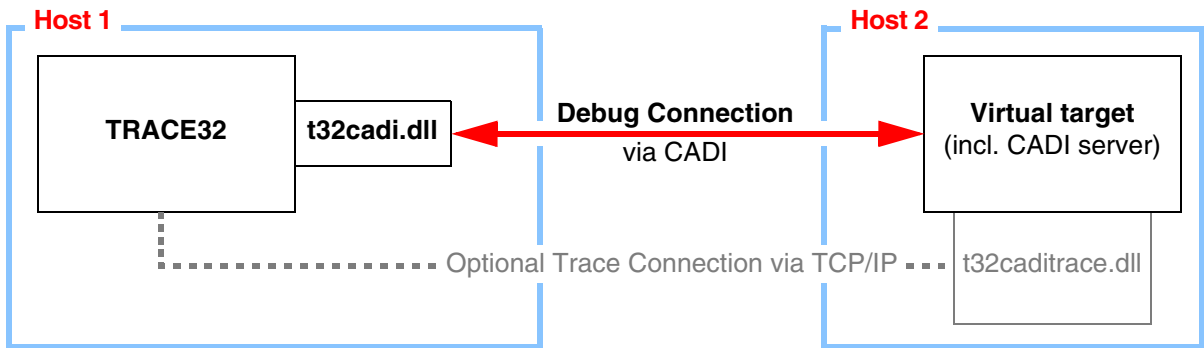
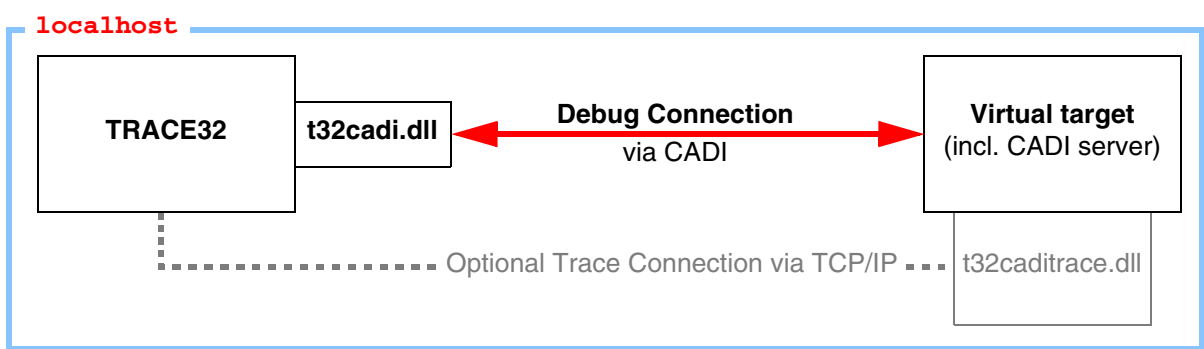


Figure 2 of 2: The red line illustrates the localhost connection that is defined using the **SYStem.CADIconfig.RemoteServer** command.



Examples:

```
SYStem.CADIconfig.RemoteServer  
  
SYStem.CADIconfig.RemoteServer 192.168.178.2 7002.  
  
SYStem.CADIconfig.RemoteServer RmtPC 7000.
```

See also

- [SYStem.CADIconfig](#)
- [SYStem.CADIconfig.RemoteServer\(\)](#)

Format:	SYSystem.CADlconfig.SpecRegDefine {<offset> <Reg name>}
---------	--

With **SYSystem.CADlconfig.SpecRegDefine** a generic read and write access to registers which are not part of the regular CPU architecture register sets and its peripherals can be defined. The special defined register is searched from the complete CADI provided register list. **SYSystem.CADlconfig.SpecRegDefine** without any parameter deletes the complete special register definition (=default). A maximum of six registers can be defined.

<offset>	Address offset in hex format to access the register named with <Reg name>. The offset is add to the CADI base address for special registers defines (= DBG:0C0000000). Maximum offset value is 0x0FFFFFF, higher values are masked with 0x0FFFFFF.
<Reg name>	Name of the special defined register as case sensitive string. Maximum string length is 27 characters.

See also

- [SYSystem.CADlconfig](#)

SYSystem.CADlconfig.SpecRegsOnly

Use only special defined register set

Format:	SYSystem.CADlconfig.SpecRegsOnly ON OFF
---------	--

Default = OFF.

ON	TRACE32 searches only for special defined registers (see SYSystem.CADlconfig.SpecRegDefine)
OFF	TRACE32 searches for regular register sets defined by SYSystem.CPU selection and for special defined registers.

See also

- [SYSystem.CADlconfig](#)

SYStem.CADlconfig.Traceconfig

Define network settings to CADI trace

Virtual targets only: CADI

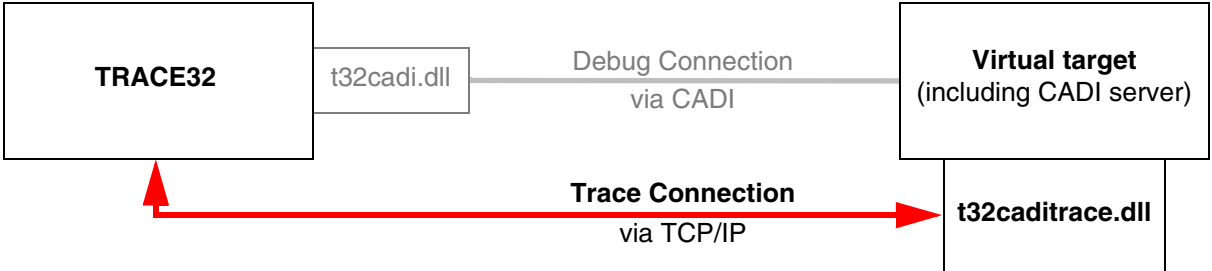
Format:SYStem.CADlconfig.Traceconfig [*<ip>* *<port>*]

Without arguments: Defines a connection to the trace plug-in based on the default values. The file name of the trace plug-in is t32caditrace.dll / t32caditrace.so.

With arguments: Defines a user-defined connection to the trace plug-in.

<i><ip></i> (default: 127.0.0.1)	IP address of the host machine where the virtual target is running. IPv4 only. Host names are <i>not</i> allowed.
<i><port></i> (default: 21000.)	Parameter Type: Decimal value . TCP/IP port of the trace plug-in.

The red line illustrates the connection that is defined using the **SYStem.CADlconfig.Traceconfig** command.



See also

- [SYStem.CADlconfig](#)
- [SYStem.CADlconfig.Traceconfig\(\)](#)
- ['Introduction' in 'Virtual Targets User's Guide'](#)

SYStem.CADlconfig.TraceCore

Define core for CADI trace

Format:SYStem.CADlconfig.TraceCore *<Corenumber>*

Defines the core for which the CADI trace feature will be active in the case of SMP core setup. CADI trace allows just one core to be traced.

See also

- [SYStem.CADlconfig](#)

Format:

SYStem.CONFIG <parameter>

SYStem.MultiCore (deprecated)

The commands of the **SYStem.CONFIG** command group describe the target configuration to the debugger.

<parameter>

The supported parameters highly depend on the target processor architecture. Please refer to the description of the **SYStem.CONFIG** command in your [Processor Architecture Manual](#) for more information.

See also

- SYStem.CONFIG.STM

■ SYStem.state

▲ 'Release Information' in 'Legacy Release History'

Format 1:	SYStem.CONFIG.CORE <core thread> <chip> SYStem.MultiCore.CORE (deprecated)
Format 2: Virtual targets only	SYStem.CONFIG.CORE <string> <value>

If the target provides a joint debug interface for several cores/chips it is necessary to inform the TRACE32 instance which core/chip it controls for debugging. This means, **SYStem.CONFIG.CORE** tells TRACE32 to which specified <core> and <chip> it has to connect when the **SYStem.Up** command is executed.

Most architectures have 1 <chip> with several <cores>. Architectures with 2 or more <chips> are relatively rare.

Next:

- Description of [Format 1 for Physical Targets](#) including example.
- Description of [Format 2 for Virtual Targets](#) including examples.

Format 1 for Physical Targets

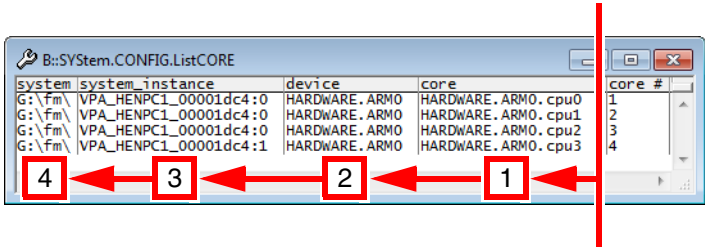
<core>	Specify the core within the chip that is controlled by the TRACE32 instance. The command SYStem.CONFIG.CORE <core> is mainly used, if the cores within a chip are not daisy-chained, or daisy-chained and interrelated. The <core> index is also used by the command CORE.NUMber as start value to set up an SMP system. See also Example for <core> and <chip>.
<chip>	Specify which cores belong to the same chip if several chips are daisy-chained. This allows the debugger to coordinate chip-wide resources e.g. chip reset, cross trigger matrix, shared trace port etc. Key for this coordination is that TRACE32 is aware of the chip.

<string>	Unique string identifying the core to which TRACE32 connects in a virtual target. The syntax rules for <string> differ for the MCD and the CADI interface: - See MCD interface - See CADI interface See also example for <string> in an SMP system. This simple example applies to the MCD and the CADI interface.
<value>	Core number identifying the core to which TRACE32 connects in a virtual target. The same <value> is displayed in the core # column of the SYStem.CONFIG.ListCORE window. The example for <value> applies to the MCD and the CADI interface.

MCD Interface - Syntax Rules for <string> - Format 2 of SYStem.CONFIG.CORE

To specify *unique strings identifying cores*, observe the following rules for <string> and take the screenshot below into account for a better understanding of these rules:

- The <string> is case-sensitive.
- The <string> starts at the first column **to the left of the core #** column:



```

;                                col.1   col.2           col.3           col.4
SYStem.CONFIG.CORE "<core>|<device>|<system_instance>|<system>"
```

- In the <string>, each column of the **SYStem.CONFIG.ListCORE** window is represented by a | character (vertical line). To ignore one or more columns, type one | per column to be ignored:

```

SYStem.CONFIG.CORE "||4:1" ;ignore columns 1 and 2, and connect
                           ;to the core having this unique
                           ;sub-string '4:1' in column 3
```

- You can specify (a) an entire row or (b) the contents of one or more cells (cell = intersection of column and row) or (c) just a sub-string of a cell:

```
; (a) entire row - the ellipsis ... is used for space economy
SYStem.CONFIG.CORE "HARDWARE.ARM0.cpu3 | HARDWARE.ARM0 | ... | ..."

; (b) content of one cell
SYStem.CONFIG.CORE "HARDWARE.ARM0.cpu3 "

; (c) sub-string from cell
SYStem.CONFIG.CORE "cpu3 "
```

- Wildcards like * and ? are **not** supported.
- For SMP systems: To connect to two or more cores, separate their unique strings or sub-strings with a **comma**:

```
; connect to cpu1 and cpu2
SYStem.CONFIG.CORE "cpu1,cpu2 "

; let's take the sub-strings '4:0' and '4:1' from column 3 into
; account
SYStem.CONFIG.CORE "cpu0 | | 4:0,cpu3 | | 4:1 "
```

To specify *unique strings identifying cores*, observe the following rules for <string> and take the screenshot below into account for a better understanding of these rules:

- The <string> is case-sensitive.
- The <string> always contains the **instance** or a part of it, and may contain the **Simulation ID**:

Simulation ID	simulation	core	instance	core #
7000	System Generator:FVP_VE_Cortex_A15x4	FVP_VE_Cortex_A15x4	cluster.cpu0	1
7000	System Generator:FVP_VE_Cortex_A15x4	ARM_Cortex-A15	cluster.cpu1	2
7000	System Generator:FVP_VE_Cortex_A15x4	ARM_Cortex-A15	cluster.cpu2	3
7000	System Generator:FVP_VE_Cortex_A15x4	ARM_Cortex-A15	cluster.cpu3	4
7000	System Generator:FVP_VE_Cortex_A15x4	ARM_Cortex-A15	cluster.l2_cache	5
7000	System Generator:FVP_VE_Cortex_A15x4	PVCache	cluster.l2_cache	6
7000	System Generator:FVP_VE_Cortex_A15x4	PVCache	cluster.cpu0.l1dcache	7
7000	System Generator:FVP_VE_Cortex_A15x4	PVCache	cluster.cpu0.l1icache	8
7000	System Generator:FVP_VE_Cortex_A15x4	PVCache	cluster.cpu1.l1dcache	9

```
;connect to a core by just specifying the instance
SYStem.CONFIG.CORE "cluster.cpu0"
```

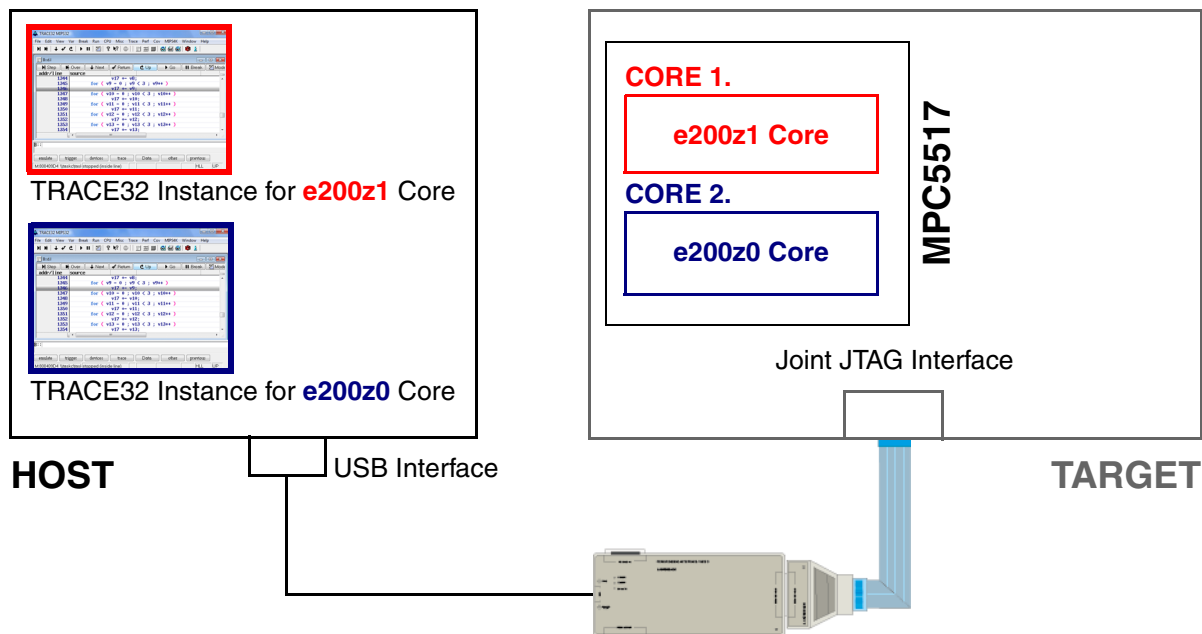
```
;connect to a core by specifying the Simulation ID and the instance
SYStem.CONFIG.CORE "7000|cluster.cpu0"
```

- Information from columns other than **Simulation ID** and **instance** cannot be used to define a connection to a core.
- Wildcards like * and ? are **not** supported.
- For SMP systems: To connect to two or more cores, separate their unique strings or sub-strings with a **comma**:

```
;connect to cpu0 and cpu3, see instance column
SYStem.CONFIG.CORE "cpu0,cpu3"
```

```
;let's take the Simulation ID into account
SYStem.CONFIG.CORE "7000|cpu0,7000|cpu3"
```

Example for the MPC5517 (one chip with 2 cores)



Start-up commands for the **e200z1** core:

```
SYStem.RESet                ; reset all SYStem settings

SYStem.CPU MPC5517          ; select MPC5517 as target
                             ; processor

;                            <core> <chip> ; assign the TRACE32 instance to
SYStem.CONFIG.CORE         1.    1.    ; core 1 (e200z1), chip 1

SYStem.Up                  ; establish the communication
                             ; between the debugger and the
                             ; e200z1 core
```

Start-up commands for the **e200z0** core:

```
SYStem.RESet                ; reset all SYStem settings

SYStem.CPU MPC5517          ; select MPC5517 as target
                             ; processor

;                            <core> <chip> ; assign the TRACE32 instance to
SYStem.CONFIG.CORE         2.    1.    ; core 2 (e200z0), chip 1

SYStem.Mode.Attach         ; establish the communication
                             ; between the debugger and the
                             ; e200z1 core
```

Example for <string> - Format 2 of SYStem.CONFIG.CORE

This simple example applies to the MCD and the CADI interface in an SMP system. Steps that are specific to SMP systems are flagged with '(SMP system)' in the comments:

```
SYStem.DOWN

SYStem.CONFIG.ListCORE           ;list the cores of a virtual target in
                                ;the SYStem.CONFIG.ListCORE window.

SYStem.CPU CortexA9MPCore        ;select a multicore CPU (SMP system)

SYStem.CONFIG.CoreNumber 2       ;set up the number of cores you want
CORE.NUMber 2                   ;TRACE32 to connect to (SMP system)

SYStem.CONFIG.CORE "cpu1,cpu2"   ;connect to cpu1 and cpu2 (SMP system)

                                ;NOTE: In the 'Cores' pull-down list,
                                ;cpu1 becomes core 0 and
                                ;cpu2 becomes core 1.
                                ;The 'Cores' pull-down list is accessible
                                ;via the TRACE32 state line.

SYStem.Up
```

Example for <value> - Format 2 of SYStem.CONFIG.CORE

This example applies to the MCD and CADI Interface.

```
SYStem.DOWN

SYStem.CONFIG.ListCORE ;list the cores of a virtual target in the
                        ;SYStem.CONFIG.ListCORE window

                        ;the core <values> are displayed in the
                        ;'core #' column. TRACE32 connects to the core
                        ;having the specified <value>

;                        <value>
SYStem.CONFIG.CORE    2.

SYStem.Up
```

See also

■ [SYStem.CONFIG.ListCORE](#) ■ [CORE.ASSIGN](#)

Format: **SYStem.CONFIG.CoreNumber** *<number>*

Sets up the number of hardware threads that are available inside the chip. The access to the particular hardware threads can be configured by architecture specific **SYStem.CONFIG** commands described in your [Processor Architecture Manual](#). The defined hardware threads can be used by the **CORE** commands to set up an SMP system.

An error message is displayed below the command line if the command is used for single-core CPUs.

See also

❏ [CONFIGNUMBER\(\)](#)

Format:	SYStem.CONFIG.DEBUGPORT <dp_fi> <dp_emu> <dp_hw>
<dp_fi>:	Unknown CSWP0 IntelUSB0 SNEAKPEEK0 XCP0
<dp_emu>:	GTL0 GTL1 GTL2 GTL3 GTL4 InfineonDAS0 VerilogTransactor0 Unknown
<dp_hw> :	DebugCable0 DebugCableA DebugCableB

Debugging via a functional interface of the host computer (<dp_fi> parameters)

	Interface	Protocol
Unknown	(default)	
CSWP0	USB	Arm CoreSight Wire Protocol
IntelUSB0	USB	Intel DCI Protocol
SNEAKPEEK0	TCP/IP	MIPI SneakPeek Protocol
XCP0	TCP/IP	ASAM MCD-1 XCP Protocol

Debugging via a functional interface is only supported for certain architectures. Please refer to the Lauterbach home page for details.

Please refer to [“TRACE32 Debug Back-Ends”](#) (backend_overview.pdf) for setup details.

Debugging a RTL simulation or emulation (<dp_emu> parameters)

	Communication via
Unknown	(default)
GTL<n>	Generic Transactor Library designed by Lauterbach, e.g. GTL0
InfineonDAS0	Infineon DAS Server
VerilogTransactor0	Verilog Transactors for testing the low-level communication, not suitable for debugging

Debugging a RTL simulation or emulation is only supported for certain architectures. Please refer to the Lauterbach homepage for details.

Please refer to “[TRACE32 Debug Back-Ends](#)” (backend_overview.pdf) for setup details.

Debugging via TRACE32 hardware (<dp_hw> parameter)

The command **SYStem.CONFIG.DEBUGPORT** <dp_hw> is only needed for a PowerDebug plus CombiProbe configuration and that only if two Whisker cables are connected to the CombiProbe.

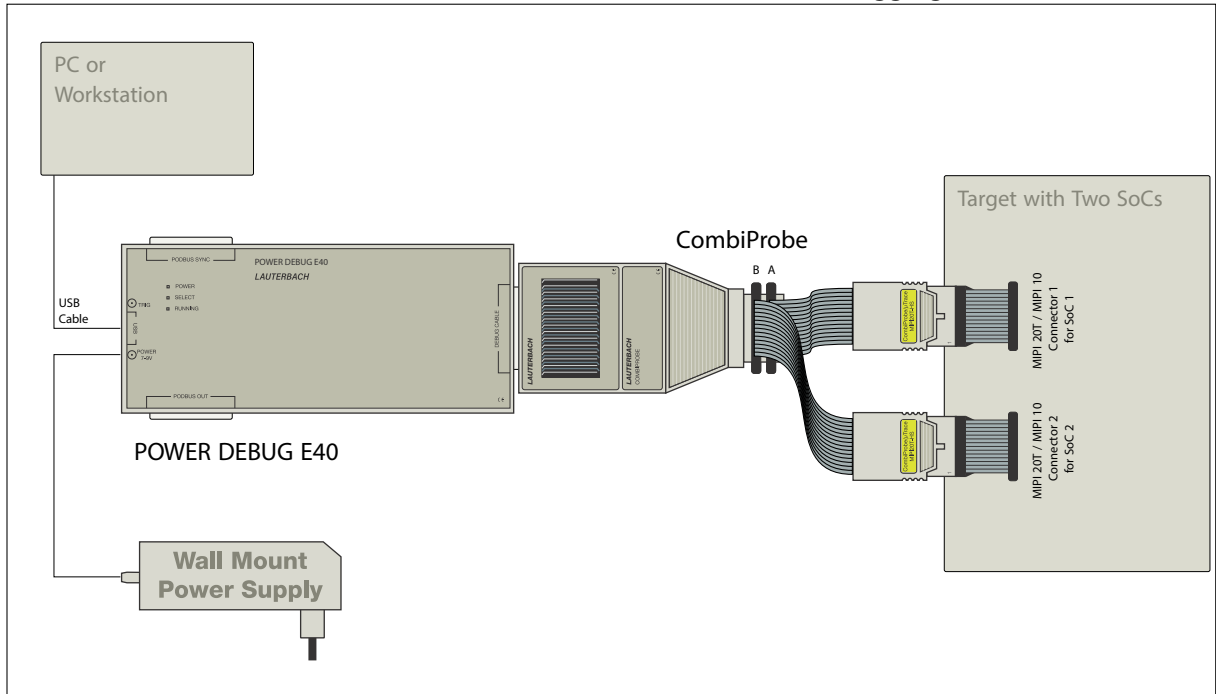
If two whiskers are connected to the CombiProbe, the following must essentially be observed:

You need to ensure that each whisker has a master PowerView instance. In single debug port scenarios, the first PowerView instance started is automatically a master. In a dual debug port scenario, you must specify the master PowerView instance yourself for the second whisker using the **SYStem.CONFIG.Slave OFF** command. Details about the command can be found in your [Processor Architecture Manual](#) manual.

Please be aware that **SYStem.RESet** also resets the **SYStem.CONFIG.DEBUGPORT** and **SYStem.CONFIG.Slave** settings. So if you want to use one of the Lauterbach demo scripts for the second whisker, you probably have to adapt the script.

There are two dual-whisker setups:

1. CombiProbe with two connected MIP20T-HS whiskers for debugging via two connectors.



If SoC 1 is debugged via the first debug connector (cable A whisker),

```
SYStem.CONFIG.DEBUGPORT DebugCableA
```

must be set in each TRACE32 instance that controls cores of this SoC.

Analogously, if SoC 2 is debugged via the second debug connector (cable B whisker)

```
SYStem.CONFIG.DEBUGPORT DebugCableB
```

must be set in each TRACE32 instance that controls cores of this SoC.

Debugging via two debug connectors is also possible using two MIP134 whiskers and a CombiProbe 2. However if you are still using a CombiProbe, it is recommended to contact Lauterbach support in advance.

2. CombiProbe with two connected AUTO26 whiskers for dsPIC33 dual core debugging.

A dual core dsPIC33 needs a CombiProbe with two AUTO26 whiskers for debugging, because each core has its own debug connector. In the required AMP setup, the following setting must be made in the TRACE32 instance controlling the first core (cable A whisker):

```
SYStem.CONFIG.DEBUGPORT DebugCableA
```

Analogously, the following must be set in the TRACE32 instance controlling the second core (cable B whisker):

```
SYStem.CONFIG.DEBUGPORT DebugCableB
```

There exist ready-to-run dual-whisker scripts from Lauterbach for this core architecture.

SYStem.CONFIG.DEBUGTIMESCALE

Extend debug driver timeouts

Format:

SYStem.CONFIG.DEBUGTIMESCALE <multiplier>

Extends any timing behavior of the debug driver by the passed <multiplier>.

The timing behavior should be adapted in case the debugger is connected to an emulator that runs with a much smaller clock compared to a silicon target. The original timeout settings may cause timeouts or bus errors in this scenario.

<multiplier>	The <multiplier> can take only values out of the power-of-two series e.g. 1, 2, 4, 8, 16 etc. A high <multiplier> can cause the software to hang for an extended period of time in case the debug driver waits for a certain condition.
--------------	---

Format:

SYStem.CONFIG.ELA <sub_cmd>

<sub_cmd>:

Base <address> | RESET

Configures the ELA CoreSight module, which provides visibility to on-chip signals. After configuration, the **ELA** command group is available.

Base <address>	Informs the debugger about the start address of the register block of the component. And this way it notifies the existence of the component.
RESET	Undoes the configuration for this component. This does not cause a physical reset for the component on the chip.

See also

- [ELA](#)

SYStem.CONFIG.ListCORE

Display the cores of a virtual target

Virtual targets only

Format:

SYStem.CONFIG.ListCORE

Retrieves the list of cores from a virtual target and displays the result as a snapshot in the **SYStem.CONFIG.ListCORE** window. To re-read the list from the virtual target, re-open the window.

The following screenshot shows an example of a core list provided by a virtual target that is connected to TRACE32 via the CADI interface:

A

Simulation ID	simulation	core	instance	core #
7000	System Generator:FVP_VE_Cortex_A15x4	FVP_VE_Cortex_A15x4	cluster.cpu0	1
7000	System Generator:FVP_VE_Cortex_A15x4	ARM_Cortex-A15	cluster.cpu1	2
7000	System Generator:FVP_VE_Cortex_A15x4	ARM_Cortex-A15	cluster.cpu2	3
7000	System Generator:FVP_VE_Cortex_A15x4	ARM_Cortex-A15	cluster.cpu3	4
7000	System Generator:FVP_VE_Cortex_A15x4	ARM_Cortex-A15	cluster.cpu3	5
7000	System Generator:FVP_VE_Cortex_A15x4	PVCache	cluster.t2_cache	6
7000	System Generator:FVP_VE_Cortex_A15x4	PVCache	cluster.cpu0.l1dcache	7
7000	System Generator:FVP_VE_Cortex_A15x4	PVCache	cluster.cpu0.l1icache	8
7000	System Generator:FVP_VE_Cortex_A15x4	PVCache	cluster.cpu1.l1dcache	9

B

The following screenshot shows an example of a core list provided by a virtual target that is connected to TRACE32 via the MCD interface:

A

system	system_instance	device	core	core #
G:\fm\	VPA_HENPC1_00001dc4:0	HARDWARE.ARM0	HARDWARE.ARM0.cpu0	1
G:\fm\	VPA_HENPC1_00001dc4:0	HARDWARE.ARM0	HARDWARE.ARM0.cpu1	2
G:\fm\	VPA_HENPC1_00001dc4:0	HARDWARE.ARM0	HARDWARE.ARM0.cpu2	3
G:\fm\	VPA_HENPC1_00001dc4:0	HARDWARE.ARM0	HARDWARE.ARM0.cpu3	4

B

- A
- Column headers in the
- SYStem.CONFIG.ListCORE**
- window correspond to the column headers of the used interface.
- B
- The
- core #**
- column displays the sequence of cores provided by the interface.

See also

-
- [SYStem.CONFIG.ListSimulation](#)
-
- [SYStem.CONFIG.CORE](#)
-
- [SYStem.CONFIG.ListCORE\(\)](#)
-
- [SYStem.CONFIG.ListSIM\(\)](#)
- ▲
- 'Connecting to Virtual Targets' in 'Virtual Targets User's Guide'

SYStem.CONFIG.ListSIMulation

Display the simulations of a virtual target

Virtual targets only: CADI

Format:SYStem.CONFIG.ListSIMulation

Retrieves the list of simulations from a virtual target and displays the result as a snapshot in the **SYStem.CONFIG.ListSIMulation** window. To re-read the list from the virtual target, re-open the window.

A

sim #	Simulation ID	name	CADISimulation2	description
1	7000	System Generator:FVP_VE_Cortex_A15x4	yes v0	pid = 5056, pwd =
2	7001	System Generator:FVP_VE_Cortex_A15x4	yes v0	pid = 5920, pwd =

- A
- The
- sim #**
- column displays the sequence of simulations provided by the interface.

See also

-
- [SYStem.CONFIG.ListCORE](#)
-
- [SYStem.CONFIG.ListCORE\(\)](#)
-
- [SYStem.CONFIG.ListSIM\(\)](#)

SYStem.CONFIG.MULTITAP

Select type of JTAG multi-TAP network

Format:

SYStem.CONFIG.MULTITAP <sub_cmd>

<sub_cmd>:

NONE

PrimaryTAP <irlength> <irvalue> <drlength> <drenable> <drdisable>

Some SoCs with a JTAG interface need special JTAG sequences before the core can be accessed. E.g. The JTAG-TAP of the core has to be dynamically added to the JTAG daisy chain of the SoC.

NONE	Disables any special multi-TAP handling (default).
PrimaryTAP	Before accessing the core via JTAG, the debugger writes <irvalue> to the JTAG IR shift-register and then <drenable> to the JTAG DR shift-register. After accessing the core via JTAG, the debugger writes <irvalue> to the JTAG IR shift-register and then <drdisable> to the JTAG DR shift-register.

See also

- [SYStem.CONFIG.MULTITAP.JtagSequence](#)

Format:

SYStem.CONFIG.MULTITAP.JtagSEQUence.<sub_cmd>

<sub_cmd>:

on

Attach <seq_name> | default

SElect <seq_name> | none

DeSElect <seq_name> | none

Some SoCs with a JTAG interface need special JTAG sequences before the core can be accessed. For example, the JTAG-TAP of some ARC cores has to be dynamically added to the JTAG daisy chain of an SoC.

With this command you can select complex JTAG sequences which should be executed by the debugger whenever it switches between the debugged cores.

For some CPUs (selected with [SYStem.CPU](#)), TRACE32 already provides pre-defined JTAG sequences. For others, you can create JTAG sequences for your individual needs with [JTAG.SEquence.Create](#).

The debugger assumes that every JTAG sequence starts and ends in the state of the JTAG state machine which was set with [SYStem.CONFIG.TAPState](#).

<seq_name>	Name of a JTAG sequence created with JTAG.SEquence .
Attach	Choose a JTAG sequence which is executed during the attach to the core via SYStem.Up or SYStem.Attach . If SYStem.CONFIG SLAVE is set to OFF , the selected JTAG sequence should reset the JTAG TAP (e.g. by going to the tests-logic-reset state).
SElect	Specifies the JTAG sequence to be executed for core A when the debugger switches from any other core to core A (in a multicore setup). <other> core ==switch==> core A
DeSElect	Specifies the JTAG sequence to be executed for core A when the debugger switches from core A to any other core (in a multicore setup). core A ==switch==> <other> core
default	By choosing Attach default , the debugger will perform a default action, which is: Resetting the core by going to the tests-logic-reset state (if SYStem.CONFIG SLAVE is set to OFF) and then starting the JTAG sequence, which has been assigned to SElect .
none	No JTAG sequence will be executed.
on	Enable the multitap JTAG sequences.

Example: This script snippet illustrates the use of JTAG sequences in PRACTICE scripts (*.cmm), i.e. where and when you should define and enable a JTAG sequence, and when it is executed automatically.

```
SYStem.CPU <cpu_type>
...

;define a JTAG sequence
JTAG.SEquence.Create myAttach
JTAG.SEquence.Add    myAttach  <your_code>
...

;enable the multitap JTAG sequences
SYStem.CONFIG.MULTITAP.JtagSEquence.on
SYStem.CONFIG.MULTITAP.JtagSEquence.Attach myAttach
...

;the JTAG sequence 'myAttach' is executed automatically on SYStem.Up
SYStem.Up
...
```

See also

■ [SYStem.CONFIG.MULTITAP](#) ■ [JTAG.SEquence](#)

Format:	SYStem.CONFIG.state [/<tab>] SYStem.MultiCore.view (deprecated)
---------	--

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

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

<tab>	Opens the SYStem.CONFIG.state window on the specified tab: • DebugPort • Jtag • etc. The number of tabs and commands on the tabs are CPU specific. For architecture-specific information about the tabs, refer to the Processor Architecture Manuals [▲] listed in the See also block below.
-------	--

Example:

```
SYStem.CONFIG.state /Jtag      ;open the window on the Jtag tab
```

Format:	SYStem.CONFIG.TRACEPORT <index> <sub_cmd>
<index>:	1 2 ...
<sub_cmd>:	TraceSource <source> Name <name> Type <type> RESET

<index>	Index number of the trace port.
TraceSource <source>	Declares the trace <source>, e.g. TPIU, ETR, ETM. These trace <sources> are only available if: • You have already added them as components in the SYStem.CONFIG.state /Components window. • You have programmed them with the appropriate commands, see example below. For more information, refer to the SYStem.CONFIG description in your Processor Architecture Manuals.
Name <name>	Assigns a user-defined <name> to a trace port. Unique names are useful for AMP debugging because they allow you to differentiate trace ports with identical types across multiple TRACE32 instances.
Type <type>	Declares the trace port type, e.g. AURORA or PCIE .
RESET	Removes the trace port declaration.

Example:

```
; declare system trace components
SYStem.CONFIG DTMCONFIG ON
SYStem.CONFIG.ETM.BASE APB:0x8000E000
SYStem.CONFIG.FUNNEL1.BASE APB:0x80004000
SYStem.CONFIG.FUNNEL1.ATBSOURCE ETM 0 DTM 2
SYStem.CONFIG.ETF1.BASE APB:0x8000C000
SYStem.CONFIG.ETF1.ATBSOURCE FUNNEL1
SYStem.CONFIG.TPIU.BASE APB:0x80003000
SYStem.CONFIG.TPIU.ATBSource ETF1
SYStem.CONFIG.TRACEPORT1.Type.AURORA
SYStem.CONFIG.TRACEPORT1.TraceSource TPIU
```

See also

- [TRACEPORT](#)

Format: **SYStem.CONFIG.TRANSACTORPIPENAME** <file>

Defines the pipe name used to communicate with the Verilog Transactor in the RTL Simulation.

Example: Enter the transactor pipe name in TRACE32 PowerView.

```
SYStem.CONFIG.DEBUGPORT VerilogTransactor0
SYStem.CONFIG.TRANSACTORPIPENAME "/tmp/t32verilog_actuator_user"
```

Linux: Define environment variable in the context of the RTL simulation and Verilog Transactor in the shell.

```
> export T32VERILOGTRANSACTORPIPE=/tmp/t32verilog_actuator_user
```

SYStem.CONFIG.USB

USB configuration

Intel® x86

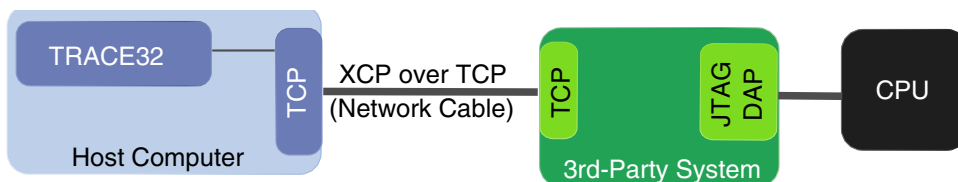
Using the **SYStem.CONFIG.USB** command group, you can configure a TRACE32 system for debugging via the USB protocol.

For more information, see [“Debugging via USB User’s Guide”](#) (usbdebug_user.pdf).

SYStem.CONFIG.XCP

XCP specific settings

The **SYStem.CONFIG.XCP** command group allows to set up and configure debugging over XCP.



The command group is available after **XCP0** has been selected as debug port.

```
;optional step: open the SYStem.CONFIG.state dialog showing the DebugPort
;tab
SYStem.CONFIG.state /DebugPort

;selecting the XCP back-end activates the SYStem.CONFIG XCP commands
SYStem.CONFIG.DEBUGPORT XCP0
```

For more information, see “[XCP Debug Back-End](#)” (backend_xcp.pdf).

See also

■ [SYStem.CONFIG](#)

Format: **SYStem.CPU** <cpu>

Tells TRACE32 the exact CPU type used on your target. This is required to get the matching **PER** file and other CPU specific settings (e.g. predefined settings for on-chip FLASH). Asterisks (*) can be used as wildcard characters to list all CPUs of an architecture or just the ones matching the filter criterion.

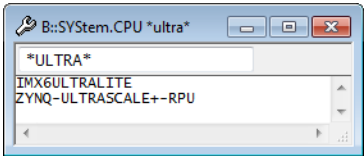
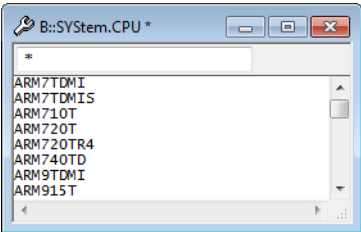
Examples:

```
SYStem.CPU ARM940T ;select the CPU type ARM940T
;...               your code
SYStem.Up          ;start the debugger
```

NOTE: **SYStem.CPU** used together with an asterisk in a PRACTICE script (*.cmm) causes the script to stop, and the **SYStem.CPU** window is displayed until you have made a selection.

```
SYStem.CPU *       ;list the CPUs of an architecture
```

```
SYStem.CPU *ultra* ;list the CPUs of an architecture matching the
                   ;filter criterion
```



See also

- [SYStem.state](#)
- [PER.view](#)
- [CPUFAMILY\(\)](#)
- [SYStem.CPU\(\)](#)
- ['Release Information' in 'Legacy Release History'](#)

Format:

SYStem.CpuAccess *<sub_cmd>* (deprecated)

<sub_cmd>:

Enable (deprecated)
Use **SYStem.MemAccess StopAndGo** instead.

Denied (deprecated)
There is no need to use a successor command (default setting).

Nonstop (deprecated)
Use **SYStem.CpuBreak Denied** instead.

Default: Denied.

Configures how memory access is handled during run-time.

- Enable**

Allow intrusive run-time memory access.
- Denied**

Lock intrusive run-time memory access.
- Nonstop**

Lock all features of the debugger that affect the run-time behavior.

See also

■ [SYStem.state](#)

Format:	SYSystem.CpuBreak [<i><mode></i>]
<i><mode></i> :	Enable Denied

Default: Enable.

Enable	Allows stopping the target.
Denied	Denies stopping the target. This includes manual stops and stop breakpoints. However, short stops, such as spot breakpoints, may still be allowed. SYSystem.CpuBreak Denied can be used to protect a target system which does not tolerate that the program execution is stopped for an extended period of time; for example, a motor controller which could damage the motor if the motor control software is stopped. For more information, see SYSystem.CpuSpot, SYSystem.MemAccess.

See also

- [SYSystem.state](#)

Format:	SYSystem.CpuSpot [<i><mode></i>]
<i><mode></i> :	Enable Denied Target SINGLE

Default: Enable.

Spotting is an intrusive way to transfer data periodically or on certain events from the target system to the debugger. As a result, the program is not running in real-time anymore. For more information, see [SYSystem.CpuBreak](#) and [SYSystem.MemAccess](#).

Enable	Allows spotting the target.
Denied	Denies spotting the target. Stopping the target may still be allowed.
Target	Allows spotting the target controlled by the target. This allows target-stopped FDX and TERM communication. All other spots are denied.
SINGLE	Allows single spots triggered by a command. This includes spotting for changing the breakpoint configuration and the SNOOPPer.PC command. This setting also allows target-stopped FDX and TERM communication. All other spots are denied.

See also

- [SYSystem.state](#)

SYSystem.DCI

DCI configuration

Intel® x86

The Intel® Direct Connect Interface (DCI) allows debugging and tracing of Intel® targets using the USB3 port of the target system.

The [SYSystem.DCI](#) command group allows to configure target properties as well as TRACE32 hardware dedicated for the use with DCI.

For more information, see “[Debugging via Intel® DCI User’s Guide](#)” (dci_intel_user.pdf).

See also

- [SYSystem](#)

Format:	SYStem.DETECT <type>
<type>:	state BASECPU CPU DaisyChain CoreSightTrace DAP IDCode JtagClock PortSHaRing

state	Opens the System Detection Wizard which allows a step-by-step investigation of the entire JTAG chain. It displays all found devices and suggests a CPU and the corresponding multi-core settings for each found device. The user can choose the desired core, any valid multi-core configuration will be applied.
BASECPU	Detects and selects the appropriate CPU and multi-core configuration for the first device in the JTAG chain. CPU detection is only based on the JTAG ID code, no further ID registers are read from the device.
CPU	Detects and selects the appropriate CPU and multi-core configuration for the first device in the JTAG chain. The command will investigate the entire device for an exact detection. If supported for your architecture, you can use SYStem.CPU AUTO instead.
DaisyChain	Scans your JTAG chain and prints details to the AREA window. See example .
CoreSightTrace	Tries to detect the ARM CoreSight Trace topology and prints details to the AREA window.
DAP	Identifies the types of access ports of the DAP. In addition, the SYStem.DETECT.DAP command inspects the ROM tables, if available, to discover any CoreSight components and their access addresses. The result is displayed in the form of a list in the SYStem.DETECT.DAP window.
SHOWChain	Scans the JTAG chain and show details in a separate window. In this window (SYStem.DETECT SHOWChain) you can double-click on a CPU core to set the values IRPOST, IRPRE, DRPOST and DRPRE in window SYSTEM.CONFIG state /Jtag accordingly.

IDCode Detects the ID codes of all JTAG-TAP controllers in the JTAG chain and stores them internally, i.e. the result is **not** printed to the **AREA** window. In order to access the result use the following functions:

- **IDCODENUMBER()** returns the number of detected TAP controllers.
- **IDCODE()** returns the IDCODE of the n-th TAP controller

Example: `PRINT IDCODE(0)` prints `0x100034B1`, if the first core in the JTAG chain is an ARC700.

JtagClock Determines the maximum JTAG Frequency by polling the BYPASS register. This only reflects the quality of the electrical connection and the speed the BYPASS register path. The function **EVAL()** retrieves the result of the command in Hertz. This command may heavily confuse all devices attached to the JTAG chain.

PortSHaRing Determines if the debug port is shared with a 3rd-party tool. In a PRACTICE script, the result can be obtained by the **PORTSHARING()** function.

NOTE:	The availability of the SYStem.DETECT types are dependent on the architecture and the debug port protocol, so they may not be available for all architectures and configurations.
--------------	--

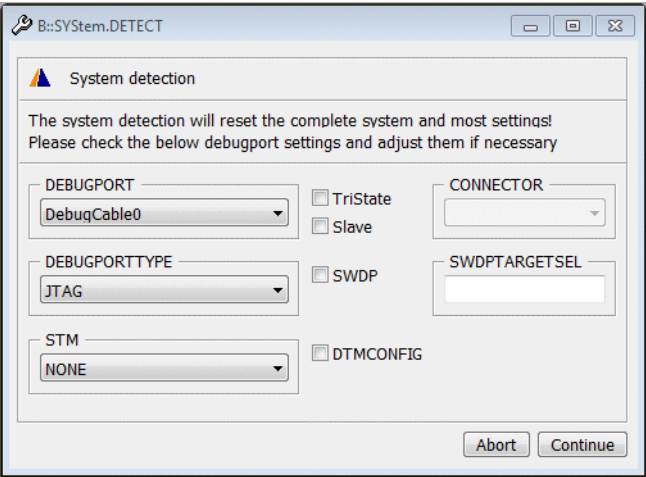
NOTE:	SYStem.DETECT may apply a Power-on Reset to your system and always switches the debug system to Down-State (see SYStem.Mode Down).
--------------	---

The System Detection Wizard

The System Detection Wizard is still under construction. It currently supports only the following architectures: ARM/Cortex, MIPS, TriCore.

Example for the ARM architecture:

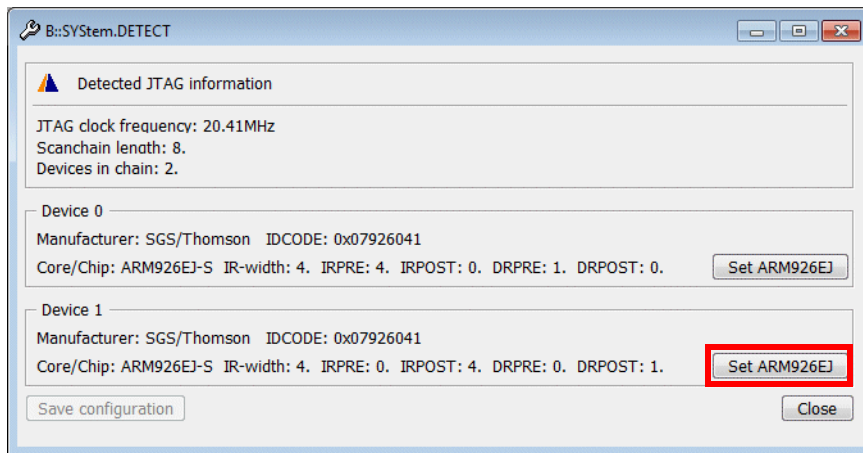
```
SYStem.DETECT state                                ; open a System Detection Wizard
```



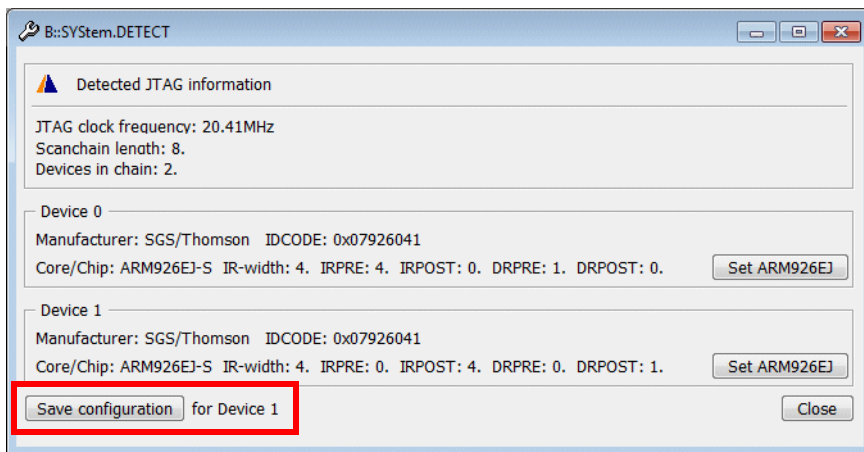
Before TRACE32 can check your target configuration it needs to know how the TRACE32 debugger is connected to the target (here for example via JTAG).

Click **Continue** to confirm that the connection details are valid.

TRACE32 runs the system detection and display the results.



If several cores are detected, you can choose which core should be controlled by the current TRACE32 instance by pushing the **Set <core>** button. Here for example “Device 1: Set ARM9EJ”.

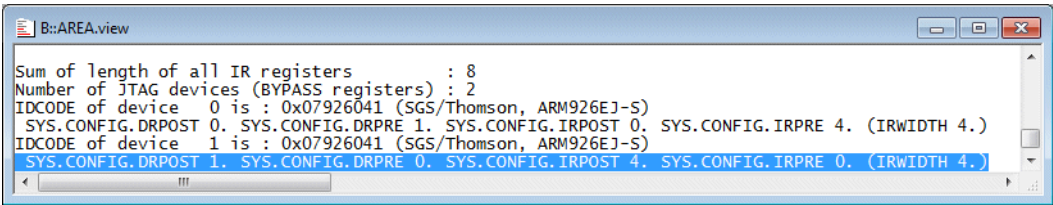


The **Save Configuration** button allows you to store the chosen settings to a file.

Daisy-Chain Detection via the TRACE32 AREA Window

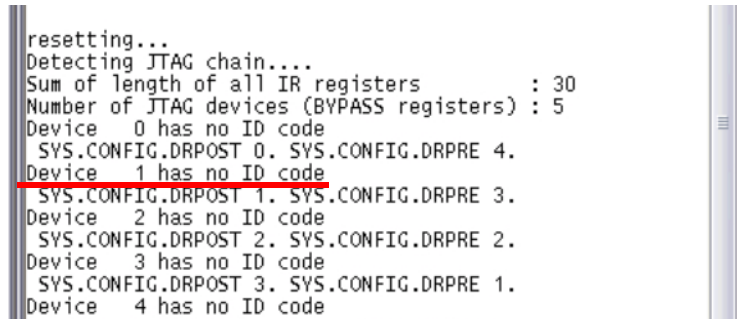
```
AREA.view ; open a TRACE32 AREA window

SYStem.DETECT.DaisyChain ; run TRACE32 daisy-chain detection
```



The result of the daisy-chain detection is displayed in the **AREA** window. TRACE32 also displays the commands that are required for a correct daisy-chain setup (see blue bar). Just copy these commands to your script, separate them by space and you are done.

Please be aware that TRACE32 can only generate the commands for the daisy-chain setup if it knows the ID codes for the cores on your target (see picture below).



See also

- [SYStem.state](#)
- [IDCODE\(\)](#)

SYStem.DLLCommand

Custom DLL connection to target

Debugger MIPS, V24 monitor with DLL

Format:

SYStem.DLLCommand

See also

- [SYStem.state](#)

The **SYStem.InfineonDAS** command group allows to configure the back-end for DAS. The command group is available after **InfineonDAS0** has been selected as debug port.

```
;optional step: open the SYStem.CONFIG.state dialog showing the DebugPort
;tab
SYStem.CONFIG.state /DebugPort

;selecting the DAS back-end activates the SYStem.InfineonDAS commands
SYStem.CONFIG.DEBUGPORT InfineonDAS0
```

For more information, see [“Debugging via Infineon DAS Server”](#) (backend_das.pdf).

See also

■ [SYStem](#)

Virtual targets only: IRIS

The **SYStem.IRISconfig** command group is used to define IRIS-specific setups for debugging and tracing.

See also

- [SYStem.CADIconfig](#)
- [SYStem.state](#)

SYStem.IRISconfig.RemoteServer

Define connection to IRIS server

Virtual targets only: IRIS

Format:

SYStem.IRISconfig.RemoteServer [*<ip>* *<port>*]

Defines connection parameters to IRIS server.

Format: **SYStem.JtagClock** <frequency>

Selects the JTAG port frequency (TCK) used by the debugger to communicate with the processor. The frequency affects e.g. the download speed. It could be required to reduce the JTAG frequency if there are buffers, additional loads or high capacities on the JTAG lines or if VTREF is very low. A very high frequency will not work on all systems and will result in an erroneous data transfer. Therefore we recommend to use the default setting if possible.

See also

- [SYStem.state](#)
- [SYStem.JtagClock\(\)](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

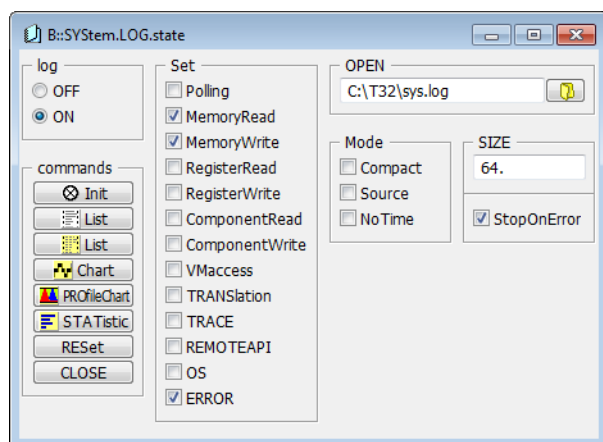
Using the **SYStem.LOG** command group, you can record the read and write accesses TRACE32 performs to the target hardware. For example, the **SYStem.LOG** command group can be used to diagnose why errors like “debug port fail”, “bus error”, etc. occurred. By default, logging stops after an error has occurred (see [SYStem.LOG.StopOnError](#)).

The read and write accesses can be displayed in the [SYStem.LOG.List](#) window. In addition, they can be recorded in a system log file with an unlimited file size. The log entries are recorded in plain text format, and the read and write accesses are converted to [Data.Set](#) commands. This way it is possible to re-run the system log in a TRACE32 Instruction Set Simulator.

The system log records all TRACE32 debugger accesses to the target. Examples of accesses that cannot be logged include:

- Accesses via JTAG API
- Accesses initiated by the [TERM](#) command
- Accesses initiated by the [SNOOPer](#) command

For configuring a system log, use the TRACE32 command line, a PRACTICE script (*.cmm), or the [SYStem.LOG.state](#) window:



Example:

```
SYStem.LOG.state           ;optional: open the configuration window
SYStem.LOG.RESet          ;use default configuration of system log
SYStem.LOG.OPEN ~~\sys.log ;open a system log file for writing
SYStem.LOG.List           ;display the accesses to the target

;log the read and write accesses to the target
List.auto
Step.single

SYStem.LOG.CLOSE          ;close the system log file for writing
```

See also

- | | | | |
|--|------------------------------------|-----------------------------------|------------------------------------|
| ■ SYStem.LOG.CLEAR | ■ SYStem.LOG.CLOSE | ■ SYStem.LOG.Init | ■ SYStem.LOG.List |
| ■ SYStem.LOG.Mode | ■ SYStem.LOG.OFF | ■ SYStem.LOG.ON | ■ SYStem.LOG.OPEN |
| ■ SYStem.LOG.RESet | ■ SYStem.LOG.Set | ■ SYStem.LOG.SIZE | ■ SYStem.LOG.state |
| ■ SYStem.LOG.StopOnError | ■ SYStem.state | ■ SLTrace | ■ LOG |

SYStem.LOG.CLEAR

Clear the ‘SYStem.LOG.List’ window

Format:

SYStem.LOG.CLEAR

Clears and immediately re-populates the list in the [SYStem.LOG.List](#) window with new system log entries - same as if you are running the commands [SYStem.LOG.Init](#) and [SYStem.LOG.ON](#) in rapid succession.

SYStem.LOG.CLEAR is the command behind the **Clear** button in the [SYStem.LOG.List](#) window.

See also

- | | | |
|------------------------------|-----------------------------------|------------------------------------|
| ■ SYStem.LOG | ■ SYStem.LOG.Init | ■ SYStem.LOG.state |
|------------------------------|-----------------------------------|------------------------------------|

Format:	SYStem.LOG.CLOSE
---------	------------------

Closes the active system log file. Any further read and write accesses are not recorded in the system log file. You can now open the file in an **EDIT** or **TYPE** window or in another application.

Example: Using the **DO** command, you can also re-run the system log in a TRACE32 Instruction Set Simulator. You do **not** need to rename the file extension to **cmm**, the extension for PRACTICE scripts; you can keep the file extension **log**.

```
...

SYStem.LOG.CLOSE                ;close the system log file for writing

PEDIT ~~\sys.log                ;open log file as a PRACTICE script
```

See also

[SYStem.LOG](#)[SYStem.LOG.OPEN](#)[SYStem.LOG.state](#)

SYStem.LOG.Init

Clear the "SYStem.LOG.List" window

Format:	SYStem.LOG.Init
---------	-----------------

Clears the system log entries displayed in the **SYStem.LOG.List** window. The **SYStem.LOG.Init** command has **no** impact on the system log file.

Since the **SYStem.LOG.List** window itself continues to log the target, the list in the window may be immediately re-populated after clearing.

To prevent the list in the **SYStem.LOG.List** window from being re-populated, run these commands:

```
SYStem.LOG.OFF
SYStem.LOG.Init
```

See also

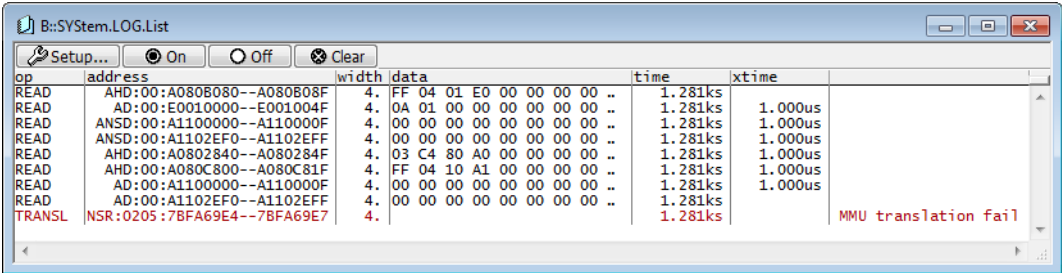
[SYStem.LOG](#)[SYStem.LOG.CLEAR](#)[SYStem.LOG.state](#)

Format:

SYSystem.LOG.List

Data.LOG (deprecated)

Displays all types of target accesses made by TRACE32.



Description of Toolbar Buttons in the SYSystem.LOG.List Window

Setup	Opens the SYSystem.LOG.state window.
On	Starts/resumes logging - same as SYSystem.LOG.ON .
Off	Pauses logging - same as SYSystem.LOG.OFF .
Clear	Clears and immediately re-populates the list with new system log entries - same as if you are running the commands SYSystem.LOG.Init and SYSystem.LOG.ON in rapid succession.

Description of Columns in the SYSystem.LOG.List Window

op	Type of target access.
address	Access class and address where TRACE32 has accessed the target.
width	The bus width used by TRACE32. Example: The value 4 . indicates that TRACE32 performs 32-bit read or write accesses.
data	Data written or read. Only the first 8 bytes are displayed in the SYSystem.LOG.List window.
time	Absolute timestamps in relation to ZERO .
xtime	Execution time.

See also

- [SYSystem.LOG](#)
- [SYSystem.LOG.state](#)
- ▲ 'Release Information' in 'Legacy Release History'

Format:	SYStem.LOG.Mode <logging_mode>		
<logging_mode>:	Compact	ON OFF	
	Source	ON OFF	
	NoTime	ON OFF	

Sets the degree of detail with which the read and write accesses are recorded in the system log file.

Compact (default: OFF)	OFF: Access addresses and data are included in the system log file. ON: Access addresses are logged, but data is not logged.
Source (default: OFF)	OFF: The source of accesses such as ETM, HTM, etc. is not logged. ON: Information about which component has accessed the target is included in the system log file.
NoTime (default: OFF)	OFF: Timing information is included in the system log file; see time and xtime columns of the SYStem.LOG.List window. ON: Timing information is not logged. NoTime ON is useful, for example, if you want to compare two versions of a system log file.

See also

- [SYStem.LOG](#)
- [SYStem.LOG.state](#)

Format:

SYStem.LOG.OFF

Temporarily deactivates logging, i.e. the read and write accesses are no longer logged. However, the system log file remains operational. Logging can be resumed with [SYStem.LOG.ON](#).

Example:

```
SYStem.LOG.OPEN  ~~\sys.log  ;open a system log file for writing
;...                ;log read and write accesses
;...
SYStem.LOG.OFF    ;temporarily deactivate logging
;...                ;accesses are no longer logged
SYStem.LOG.ON     ;resume logging
;...                ;accesses are logged again
;...
SYStem.LOG.CLOSE  ;close system log file and terminate logging
```

See also

- [SYStem.LOG.ON](#)
- [SYStem.LOG](#)
- [SYStem.LOG.state](#)

Format:

SYStem.LOG.ON

Logs all read and write accesses you have selected with [SYStem.LOG.Set](#). The **SYStem.LOG.ON** command can be used after the system log has been temporarily deactivated with the command [SYStem.LOG.OFF](#).

See also

- [SYStem.LOG.OFF](#)
- [SYStem.LOG](#)
- [SYStem.LOG.state](#)

Format:	SYStem.LOG.OPEN <i><file></i>
---------	--

Generates a new system log file for logging read and write accesses and opens it for writing. The number of logged read and write accesses is unlimited. If a file with the same name already exists, it will be overwritten.

<i><file></i>	The default extension for <i><file></i> is *.log .
---------------------	---

Example:

```
SYStem.LOG.OPEN  ~~\sys.log      ;open a system log file
; ...
; ...
SYStem.LOG.CLOSE                               ;close file and terminate logging
```

The path prefix `~~` expands to the TRACE32 system directory, by default `C:\t32`.

See also

- [SYStem.LOG](#)
- [SYStem.LOG.CLOSE](#)
- [SYStem.LOG.state](#)

SYStem.LOG.RESet

Reset configuration of system log to defaults

Format:	SYStem.LOG.RESet
---------	-------------------------

Resets all commands of the **SYStem.LOG** command group to their defaults. You can view the result in the [SYStem.LOG.state](#) window.

See also

- [SYStem.LOG](#)
- [SYStem.LOG.state](#)

Format:

SYStem.LOG.Set <setting>

<setting>:

Polling ON | OFF
MemoryRead ON | OFF
MemoryWrite ON | OFF
RegisterRead ON | OFF
RegisterWrite ON | OFF
ComponentRead ON | OFF
ComponentWrite ON | OFF
VMaccess ON | OFF
TRANSLation ON | OFF
TRACE ON | OFF
REMOTEAPI ON | OFF
OS ON | OFF
ERROR ON | OFF

Allows you to select the TRACE32 accesses you want to record in [SYStem.LOG.List](#).

<setting> OFF	This <setting> is omitted from the system log file.
<setting> ON	This <setting> is included in the system log file.
ComponentRead	Read accesses to a debug component.
ComponentWrite	Write accesses.
ERROR	- ON: All errors are included in the system log file. - OFF: Logs only errors of the read and write accesses that are set to ON.
MemoryRead	Memory read accesses.
MemoryWrite	Memory write accesses.
OS	Accesses to the operating system (OS).
Polling	Polling of the CPU. The polling mode can be set with SYStem.POLLING .
RegisterRead	Register read accesses.
RegisterWrite	Register write accesses.
REMOTEAPI	Accesses via the TRACE32 Remote API. See also “API for Remote Control and JTAG Access in C” (api_remote_c.pdf).
TRACE	Accesses to the trace data streaming.

TRANSlation	• ON: Display valid address translations and translation failures in the system log file. Currently only the logical address is displayed. • OFF: Include only translation failures in the system log file.
VMaccess	If the option is enabled, any read or write access to addresses with the access class VM: or AVM: will be recorded in the SYStem.LOG.List window.

See also

- [SYStem.LOG](#)
- [SYStem.LOG.state](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

SYStem.LOG.SIZE

Define number of lines in the 'SYStem.LOG.List' window

Format:	SYStem.LOG.SIZE <i><lines></i>
---------	---

Default: 64.

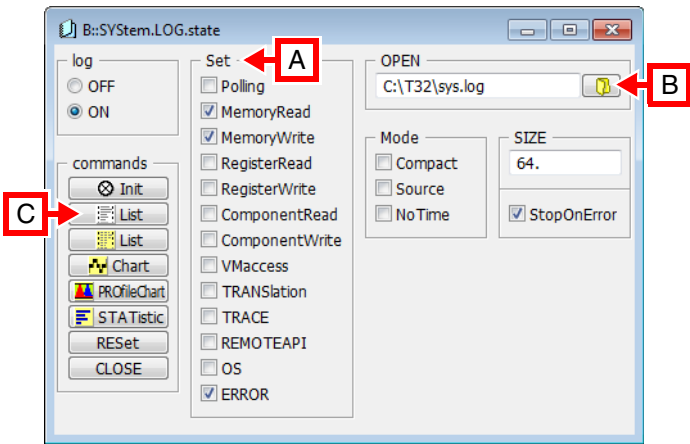
Defines the number of lines displayed in the **SYStem.LOG.List** window. The displayed lines reflect the most recent read and write accesses to the target hardware. The *<lines>* setting does **not** affect the file size.

See also

- [SYStem.LOG](#)
- [SYStem.LOG.state](#)

Format: **SYStem.LOG.state**

Opens the **SYStem.LOG.state** window, where you can configure a system log for recording read and write accesses to the target hardware.



- A** Only the information of selected options is displayed in the **SYStem.LOG.List** window and recorded in the system log file.
For descriptions of the individual options, see **SYStem.LOG.Set**.
- B To open a system log file, do one of the following:**
- Click the **folder** icon and navigate to the file you want to use.
 - Type path and file name into the **OPEN** text box. Then press **Enter**.
The TRACE32 **message line** displays that the file is now open for recording log entries.
- To close the system log file:**
- Clear the content from the **OPEN** text box. Then press **Enter**.
The TRACE32 message line displays that the file is now closed.
- C** For descriptions of the commands in the **SYStem.LOG.state** window, refer to the **SYStem.LOG.*** commands in this section.
Example: For information about the **List** button, see **SYStem.LOG.List**.

See also

- | | | | |
|--|------------------------------------|------------------------------------|-----------------------------------|
| ■ SYStem.LOG | ■ SYStem.LOG.CLEAR | ■ SYStem.LOG.CLOSE | ■ SYStem.LOG.Init |
| ■ SYStem.LOG.List | ■ SYStem.LOG.Mode | ■ SYStem.LOG.OFF | ■ SYStem.LOG.ON |
| ■ SYStem.LOG.OPEN | ■ SYStem.LOG.RESet | ■ SYStem.LOG.Set | ■ SYStem.LOG.SIZE |
| ■ SYStem.LOG.StopOnError | ■ SLTrace | | |

Format:SYStem.LOG.StopOnError ON | OFF

Defines the logging behavior after an error has occurred.

ON (default)	TRACE32 automatically runs the command SYStem.LOG.OFF to stop logging after an error, such as a bus error, has occurred.
OFF	Logging continues after an error has occurred.

NOTE:

The system log file remains open for writing.

- To continue logging, click **Start** in the [SYStem.LOG.List](#) window or run [SYStem.LOG.ON](#).
- To close the system log file, run [SYStem.LOG.CLOSE](#).

See also

- [SYStem.LOG](#)
- [SYStem.LOG.state](#)

SYStem.MCDCommand

Send command to MCD server

Virtual targets only: MCD

Format:SYStem.MCDCommand <command>

SYStem.MCDCommand <command> sends via the MCD API call **mcd_execute_command_f()** the <command> as ASCII character string to the MCD server. It is not necessary to enclose the command with quotation marks. All quotations marks within <command> will be sent as such to the MCD server. The answer from the MCD server on this command can have a maximum of 100 characters and can be read out by [SYStem.MCDCommand.ResultString\(\)](#).

See also

- [SYStem.state](#)

Virtual targets only: MCD

Format: **SYStem.MCDconfig** *<srv.cfg>*

SYStem.MCDCommand *<srv.cfg>* will send *<srv.cfg>* as ASCII character string within the parameter *config_string* of **mcd_open_server_f()** towards the MCD server. It is not necessary to enclose *<srv.cfg>* with quotation marks. All quotations marks in *<srv.cfg>* will be sent to the MCD server.

See also

■ [SYStem.state](#)

Format:	SYStem.MemAccess <i><mode></i>
<i><mode></i> :	Enable Denied StopAndGo <i><cpu_specific></i> <i><interface_specific></i>

The debugger can read and write the target memory while the CPU is executing the program.

Enable CPU (deprecated)	Is used to activate the memory access while the CPU is running on the TRACE32 Instruction Set Simulator and on debuggers which do not have a fixed name for the memory access method.
Denied	Memory access during program execution to target is disabled.
StopAndGo	Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed.
<i><cpu_specific></i>	Depending on the target processor, there may exist means to access processor / target memory while the processor is running. Examples are DAP, AHB or AXI for ARM processors, NEXUS or SAP for Power Architecture, and so on. Refer to processor architecture manuals for details.
<i><interface_specific></i>	Depending on the debug interface in use, there may exist means to access processor / target memory while the processor is running. Example: XCP.

See also

- [SYStem.state](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

Format:	SYSystem.Mode [<i><mode></i>] SYSystem.<mode> (as an alternative)
<i><mode></i> :	StandBy Down Attach NoDebug Go Up Prepare

Configures how the debugger connects to the target and how the target is handled. Please refer to the description of this command in your [Processor Architecture Manual](#) for more information.

See also

- [SYSystem.state](#)
- ▲ 'Release Information' in 'Legacy Release History'

Format:	SYSystem.Option <i><option></i> [ON OFF]
---------	---

The *<options>* of **SYSystem.Option** are used to control special features of the debugger or to configure the target. It is recommended to execute the **SYSystem.Option** commands **before** the emulation is activated by a **SYSystem.Up** or **SYSystem.Mode** command.

<i><option></i>	Mostly architecture specific. For descriptions of the options, refer to the Processor Architecture Manuals .
-----------------------	--

NOTE:	Some of the commands toggle between the options ON and OFF if they are invoked without parameters.
--------------	--

See also

- [SYSystem.Option.IMASKASM](#)
 - [SYSystem.Option.MACHINESPACES](#)
 - [SYSystem.Option.ZoneSPACES](#)
 - ▲ 'Release Information' in 'Legacy Release History'
- [SYSystem.Option.IMASKHLL](#)
 - [SYSystem.Option.MMUSPACES](#)
 - [SYSystem.state](#)

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step.

See also

■ [SYStem.Option](#)

SYStem.Option.IMASKHLL

Disable interrupts while HLL single stepping

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

Default: OFF.

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

See also

■ [SYStem.Option](#)

SYStem.Option.MACHINESPACES

Address extension for guest OSes

ARM and Intel® x86

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

Default: OFF

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

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

If **SYStem.Option.MACHINESPACES** is enabled, addresses are extended with an identifier called machine ID. The [machine ID](#) clearly specifies to which host or guest machine the address belongs.

NOTE:

For architecture-specific information about the command, refer to the [Processor Architecture Manual](#).

See also

■ [SYStem.Option](#)

SYStem.Option.MMUSPACES

Separate address spaces by space IDs

Format:

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

Default: OFF.

Enables the use of [space IDs](#) for logical addresses to support **multiple** address spaces.

For an explanation of the TRACE32 concept of [address spaces](#) ([zone spaces](#), [MMU spaces](#), and [machine spaces](#)), see [“TRACE32 Concepts”](#) (trace32_concepts.pdf).

NOTE:

SYStem.Option.MMUSPACES should not be set to **ON** if only one translation table is used on the target.

If a debug session requires space IDs, you must observe the following sequence of steps:

1. Activate **SYStem.Option.MMUSPACES**.

2. Load the symbols with [Data.LOAD](#).

Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

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

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

See also

■ [SYStem.Option](#)

SYStem.Option.ZoneSPACES

Enable symbol management for zones

Format:

SYStem.Option.ZoneSPACES [ON | OFF]

Default: OFF. Not supported for all core architectures.

For CPUs which have several operation modes with individual MMU translations and register sets, the **SYStem.Option ZoneSPACES** command allows to load separate symbol sets for these CPU modes. Within TRACE32, such CPU modes are referred to as zones. Addresses and symbols belonging to a certain CPU mode are identified by their access class specifier.

- OFF
- TRACE32 does not separate symbols by access class.
- ON
- Separate symbol sets can be loaded for each zone, even with overlapping address ranges. Loaded symbols are specific to one of the CPU zones.

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

NOTE:

For architecture-specific information about the command, refer to the [Processor Architecture Manual](#).

See also

■ [SYStem.Option](#)

Format:	SYStem.PAUSE <time> [/<option>]
<option>:	Target Host

Pauses the execution of any operation of the debugger including semihosting, status polling, and APIs. The command cannot be interrupted or canceled. The passed option only takes effect if a back-end with virtual connection to an RTL emulation/simulation is used, e.g. GTL.

<time>	• The pause time of Host is typically measured in milliseconds, e.g. 100ms • The pause time of Target is typically measured in microseconds, e.g. 10us • Without Target or Host: The used time base depends on SYStem.VirtualTiming.PauseinTargetTime.
Host	Host clock is used as time base.
Target	Emulation clock is used as time base.

See also

- [SYStem.state](#)
- [WAIT](#)

Format:	SYStem.POLLING <run_mode> <stopped_mode>
<run_mode>:	CONTInuous DEFAult FAST OFF SLOW
<stopped_mode>:	DEFAult OFF SIGnals

When the CPU is running, the debug driver can poll the CPUs state in the background to speed up operations where a fast break detection is preferred. Features that can be improved by a fast break detection are:

- CPU break from PodBus Trigger, if the CPU has no dedicated break in line
- Any communication that is based to spot break points e.g. TERM, FDX
- Break triggered actions like **Data.EPILOG**, **Data.TIMER**
- Precision of the runtime counter if the CPU has no dedicated break in line

If the poll rate is high the system is more sensitive to disturbances.

CONTInuous	Polling with maximum frequency.
FAST	Polling with interval of 1 ms.
OFF	No polling at all, to prevent from disturbance.
SLOW	Normal polling with interval of SETUP.UpdateRATE , increase RTCK time-out to hide short power down/sleep states of the CPU.
DEFAult	Polling with interval of SETUP.UpdateRATE .

NOTE:	The minium polling interval for timing measurement tasks, e.g. RunTime , is 1 ms, any lower settings are ignored.
--------------	--

When the CPU is stopped, the debug driver can poll the CPUs state in the background to observe for exceptional events caused by watch-dogs or other CPUs.

For transactor based solutions those polling can slow down the debug session, therefore it's possible to turn it off.

SIGnals	Polling of signals as RESET only
OFF	No polling at all, to prevent from disturbance.
DEFault	Polling with interval of SETUP.UpdateRATE .

See also

- [SYStem.state](#)
- [SETUP.UpdateRATE](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

SYStem.PORT

Configure external communication interface

Format:	SYStem.PORT <i><mode></i>
<i><mode></i> :	COM <i><x></i> <i><settings></i> <i><ip></i> : <i><port></i>

This command is used to configure an external communication interface:

- Between TRACE32 and a monitor program running on the target.
- Between TRACE32 and a debug agent running on the target.

Both serial and TCP/IP are supported.

<i><x></i>	COM port number, e.g. COM1, COM2
<i><settings></i>	The <i><settings></i> for the communication interfaces depend on the host operating system.

Examples:

SYStem.PORT COM1 baud=9600	; configure COM1 as external ; communication interface
SYStem.PORT 10.1.2.99:2345	; configure TCP/IP as external ; communication interface

See also

- [SYStem.state](#)

Format:	SYStem.RESet
---------	---------------------

Resets all debug system settings (such as SYStem.CPU, SYStem.JtagClock) to their default values. After this switches to the **SYStem.Mode Down** state.

See also

■ [SYStem.state](#)

■ [RESet](#)

▲ ['Release Information' in 'Legacy Release History'](#)

Format:	SYStem.RESetOut
---------	------------------------

Triggers the CPU RESET command, which initializes the peripherals. This command is not available on all probes.

Example:

```
SYStem.RESetOut           ; Initialize peripherals
```

See also

■ [SYStem.state](#)

▲ ['Release Information' in 'Legacy Release History'](#)

Format:	SYStem.RESetTarget
---------	---------------------------

A target reset is performed and then released. On most targets, **SYStem.RESetTarget** is similar to **SYStem.Up** and [Register.RESet](#). On virtual platforms usually activates a target platform reset.

See also

■ [SYStem.state](#)

▲ ['Release Information' in 'Legacy Release History'](#)

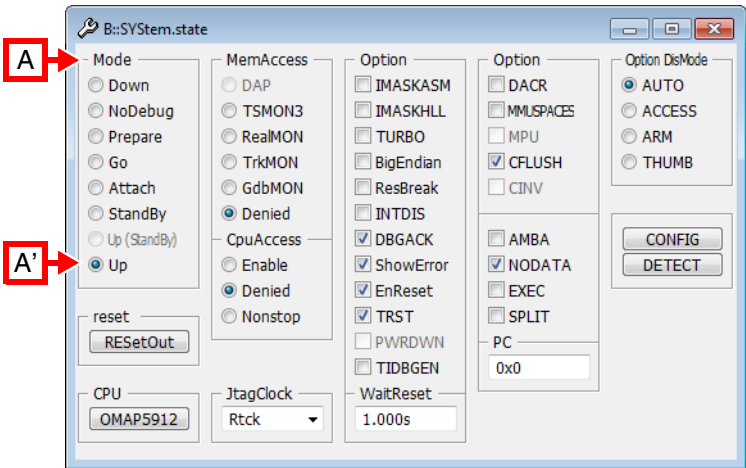
Format:

SYStem.state

Opens a **SYStem.state** window displaying all probe setup parameters. You can also open the **SYStem.state** window by double-clicking the **System** field in the [state line](#) of the [TRACE32 main window](#).

All modes can be selected and altered by clicking the appropriate buttons, options, etc. within the **SYStem.state** window.

The **SYStem.state** window is highly target-dependent; example of a **SYStem.state** window:



A For descriptions of the architecture-specific commands in the **SYStem.state** window, please refer to the **SYStem.*** commands in your [Processor Architecture Manual](#).

Example: For information about **Up**, see **SYStem.Mode Up**.

See also

- SYStem.BdmClock

■ SYStem.CONFIG

■ SYStem.CpuSpot

■ SYStem.DLLCommand

■ SYStem.JtagClock

■ SYStem.MemAccess

■ SYStem.POLLING

■ SYStem.RESetTarget

■ SYStem.VirtualTiming

□ SYStem.CPU()

■ SYStem.BREAKTIMEOUT

■ SYStem.CPU

■ SYStem.CSWP

■ SYStem.GTL

■ SYStem.LOG

■ SYStem.Mode

■ SYStem.PORT

■ SYStem.SNEAKPEEK

□ CPUIS()

□ SYStem.Mode()

■ SYStem.CADICCommand

■ SYStem.CpuAccess

■ SYStem.DCI

■ SYStem.InfineonDAS

■ SYStem.MCDCommand

■ SYStem.Option

■ SYStem.RESet

■ SYStem.TARGET

□ hardware.ICD()

□ SYStem.Up()

■ SYStem.CADIconfig

■ SYStem.CpuBreak

■ SYStem.DETECT

■ SYStem.IRISconfig

■ SYStem.MCDconfig

■ SYStem.PAUSE

■ SYStem.RESetOut

■ SYStem.TCFconfig

□ INTERFACE.SIM()

Format:

SYStem.TARGET *<target>*

<target>:

<ip_address> | *<ip_name>*

Defines the OSE target IP address or name to debug. If no target is specified, TRACE32 uses a broadcast message to find OSE targets and uses the first target found.

See also

- [SYStem.state](#)

Format:	SYStem.VirtualTiming
---------	-----------------------------

The commands of the **SYStem.VirtualTiming** command group are used to modify time timing behavior of the debugger when virtual debug interfaces are used, e.g. the TRACE32 Verilog Actuator or Generic Transactor Library.

These virtual debug interfaces connect the debugger to simulations with a virtual time or extra slow emulators. In these scenarios, PowerView commands can take a very long time to execute. Therefore, the interaction with the target is done by PRACTICE scripts mainly, but the default timing of the debugger software targets more a stutter-free behavior of the windows.

To modify these timings, the **SYStem.VirtualTiming** commands can be used. The debugger uses timeouts to cancel polling for expected results in the debug registers of the core and for reading back hardware acknowledge signals of the target as the RTCK signal.

The commands **SYStem.VirtualTiming.TimeinTargetTime** and **SYStem.VirtualTiming.PauseinTargetTime** allow to couple the timing to the emulation/simulation timing. This allows to stall the whole simulation environment without the internal timeouts expiring, but also can lead to longer execution time of the debugger.

The command **SYStem.VirtualTiming.TimeScale** can be used to reduce or increase the internal timeouts in general. In case the simulation does not respond fast enough, the timeouts need to be extended. In case the debugger polls for an error state too long to keep the user interface responsive, the timeouts can be reduced.

See also

- [SYStem.VirtualTiming.HardwareTimeout](#)
 - [SYStem.VirtualTiming.InternalClock](#)
 - [SYStem.VirtualTiming.MaxTimeout](#)
 - [SYStem.VirtualTiming.PauseinTargetTime](#)
 - [SYStem.VirtualTiming.PollingPause](#)
 - [SYStem.VirtualTiming.TimeScale](#)
 - ▲ 'Timing Adaption' in 'Debugging via Infineon DAS Server'
 - ▲ 'Timing Adaption' in 'GTL Debug Back-End'
 - ▲ 'Timing Adaption' in 'Verilog Debug Back-End'
- [SYStem.VirtualTiming.HardwareTimeoutScale](#)
 - [SYStem.VirtualTiming.MaxPause](#)
 - [SYStem.VirtualTiming.OperationPause](#)
 - [SYStem.VirtualTiming.PauseScale](#)
 - [SYStem.VirtualTiming.TimeinTargetTime](#)
 - [SYStem.state](#)

Format:

SYStem.VirtualTiming.HardwareTimeout ON | OFF

The debugger has a timeout that handle the maximum time to wait for an hardware signal. The timeout can be disabled in case it doesn't matter for the debug scenario in case error messages “no RTCK” or “subcore communication timeout” occur. In case the hardware timeout is necessary for the debug scenario the debugger can enter an endless loop, in that cases the hardware timeout should be extended by [SYStem.VirtualTiming.HardwareTimeoutScale](#) or the target should respond earlier.

- ON

Hardware timeout is active and used to cancel hardware operations.
- OFF

Hardware timeout is disabled.

Example:

```
SYStem.VirtualTiming.PauseinTargetTime OFF ; use host time
```

See also

- [SYStem.VirtualTiming](#)
- ▲ 'Timing Adaption' in 'Debugging via Infineon DAS Server'
- ▲ 'Timing Adaption' in 'GTL Debug Back-End'
- ▲ 'Timing Adaption' in 'Verilog Debug Back-End'

[\[Example\]](#)

Format:

SYStem.VirtualTiming.HardwareTimeoutScale

Scales the timeout that leads to the message “subcore communication timeout” for software-only tools. The command is implemented for the back-end GTL.

Low-level operations can have a total communication timeout to prevent the system from hanging in case of an error. In software-only tools, this timeout can malfunction. Either the timeout appears too late and the system seem to hang, or the timeout appears too early and the operation fails with the message “subcore communication timeout”.

Example:

```
;shrink the standard timeout by factor 10 to display the timeout earlier
SYStem.VirtualTiming.HardwareTimeoutScale 0.1
```

See also

■ [SYStem.VirtualTiming](#)

▲ ['Timing Adaption' in 'Debugging via Infineon DAS Server'](#)

▲ ['Timing Adaption' in 'GTL Debug Back-End'](#)

▲ ['Timing Adaption' in 'Verilog Debug Back-End'](#)

SYStem.VirtualTiming.InternalClock

Base for artificial time calculation

Format: **SYStem.VirtualTiming.InternalClock** *<frequency>*

Overrides the TRACE32 debug clock setting for internal timing calculation in case the interface's debug clock implementation is wrong in a software-only solution. A clock of 0Hz (default) will not override the debug clock setting. The command is implemented for the back-end GTL.

Example:

```
; TRACE32 transfers 10Mhz setting to the interface
SYStem.JtagClock 10Mhz

; but use 100kHz for internal timing calculations
SYStem.VirtualTiming.InternalClock 100kHz
```

See also

■ [SYStem.VirtualTiming](#)

▲ ['Timing Adaption' in 'Debugging via Infineon DAS Server'](#)

▲ ['Timing Adaption' in 'GTL Debug Back-End'](#)

▲ ['Timing Adaption' in 'Verilog Debug Back-End'](#)

Format: **SYStem.VirtualTiming.MaxPause** *<time>*

<time>: **10ns ... 40000ms**

The debugger software contains statements to wait time, in order to give the target time to respond. The command is used to set up the maximum time for those wait statements.

Example:

```
SYStem.VirtualTiming.MaxPause 1s ; set maximum to 1 second
```

See also

■ SYStem.VirtualTiming

- ▲ 'Timing Adaption' in 'Debugging via Infineon DAS Server'
- ▲ 'Timing Adaption' in 'GTL Debug Back-End'
- ▲ 'Timing Adaption' in 'Verilog Debug Back-End'

SYStem.VirtualTiming.MaxTimeout

Override time-outs

Format: **SYStem.VirtualTiming.MaxTimeout** *<time>*

<time>: **<x>ns ... <y>ms**

The debugger software contains sections where a status of the target is polled for a certain time until a conditions is met in order to finish an operation. The command is used to set up the maximum time that is used for those sections.

Example:

```
SYStem.VirtualTiming.MaxTimeout 10s ; set maximum to 10 seconds
```

See also

■ SYStem.VirtualTiming

- ▲ 'Timing Adaption' in 'Debugging via Infineon DAS Server'
- ▲ 'Timing Adaption' in 'GTL Debug Back-End'
- ▲ 'Timing Adaption' in 'Verilog Debug Back-End'

Format:	SYStem.VirtualTiming.OperationPause <i><time></i>
<i><time></i> :	<i><x></i> ns ... <i><y></i> ms

The debug driver issue pause statements after each action e.g. shift or bus access to give the emulation time to compute. Operation pauses will slow down the debugger and prevent from pre-bundling operations.

Example:

```
SYStem.VirtualTiming.OperationPause 100ns ; enable pause of 100ns
```

See also

- [SYStem.VirtualTiming](#)
- ▲ ['Timing Adaption' in 'Debugging via Infineon DAS Server'](#)
- ▲ ['Timing Adaption' in 'GTL Debug Back-End'](#)
- ▲ ['Timing Adaption' in 'Verilog Debug Back-End'](#)

Format:	SYStem.VirtualTiming.PauseinTargetTime ON OFF
---------	---

The debugger software contains statements to wait time, in order to give the target time to respond. The command specifies if the time shall elapse in virtual simulator time (ON) or real host time (OFF). When set to ON the debugger behaves as with a real target, but become as slow as the simulation.

- ON** Pause time elapses in virtual simulation time.
- OFF** Pause time elapses in real host time.

Example:

```
SYStem.VirtualTiming.PauseinTargetTime OFF ; use host time
```

See also

- [SYStem.VirtualTiming](#)
- ▲ ['Timing Adaption' in 'Debugging via Infineon DAS Server'](#)
- ▲ ['Timing Adaption' in 'GTL Debug Back-End'](#)
- ▲ ['Timing Adaption' in 'Verilog Debug Back-End'](#)

Format: **SYStem.VirtualTiming.PauseScale** *<factor>*

The debugger software contains statements to wait time in order to give the target time to respond. The command scales these time in order to wait a shorter or longer time.

Example:

```
SYStem.VirtualTiming.PauseScale 10. ; 10 times longer pauses
```

See also

■ [SYStem.VirtualTiming](#)

▲ ['Timing Adaption' in 'Debugging via Infineon DAS Server'](#)

▲ ['Timing Adaption' in 'GTL Debug Back-End'](#)

▲ ['Timing Adaption' in 'Verilog Debug Back-End'](#)

SYStem.VirtualTiming.PollingPause

Advance emulation time when polling

Format: **SYStem.VirtualTiming.PollingPause** *<time>*

The command is used for systems that advance the simulation time only when the debugger executes actions on it. Usually the debugger is executing JTAG shifts all the time, but this is not efficient to let the emulation time grow. Executing a "pause" statement at the emulation is more efficient, therefore the command inserts "pause" statements of a certain duration when the emulation time needs to be increased, e.g. when the core is running and the debugger waits until a breakpoint has been hit.

A smaller polling pause will speed up the debugger, but slow down the execution of the target program in the emulation. A bigger polling pause will make the debugger less responsive, but let that emulation run faster.

Example:

```
SYStem.VirtualTiming.PollingPause 10us ; insert 10us pauses
```

See also

■ [SYStem.VirtualTiming](#)

Format:

SYStem.VirtualTiming.TimeinTargetTime ON | OFF

The command specifies whether the timeout time shall elapse in virtual simulator time (**ON**) or real host time (**OFF**). When set to **ON** the debugger behaves as with a real target, but become as slow as the simulation.

- ON

Use virtual target time to elapse time-outs
- OFF

Use real host time to elapse time-outs

Example:

```
SYStem.VirtualTiming.TimeinTargetTime ON ; time-outs elapse time-outs
```

See also

- [SYStem.VirtualTiming](#)
- ▲ ['Timing Adaption' in 'Debugging via Infineon DAS Server'](#)
- ▲ ['Timing Adaption' in 'GTL Debug Back-End'](#)
- ▲ ['Timing Adaption' in 'Verilog Debug Back-End'](#)

Format: **SYStem.VirtualTiming.TimeScale** *<factor>*

The command scales pauses and time-outs in general.

Example 1:

```
;in this example, time-outs and pauses are 10 times longer  
SYStem.VirtualTiming.TimeScale 10.0
```

Example 2:

```
;in this example, time-outs and pauses are 100 times shorter  
SYStem.VirtualTiming.TimeScale 0.01  
  
;in case the error "RTCK fail" or "subcore communication timeout"  
;occur, extend hardware timeouts by factor 10  
SYStem.VirtualTiming.HardwareTimeoutScale 10.0
```

See also

■ [SYStem.VirtualTiming](#)

▲ ['Timing Adaption' in 'Debugging via Infineon DAS Server'](#)

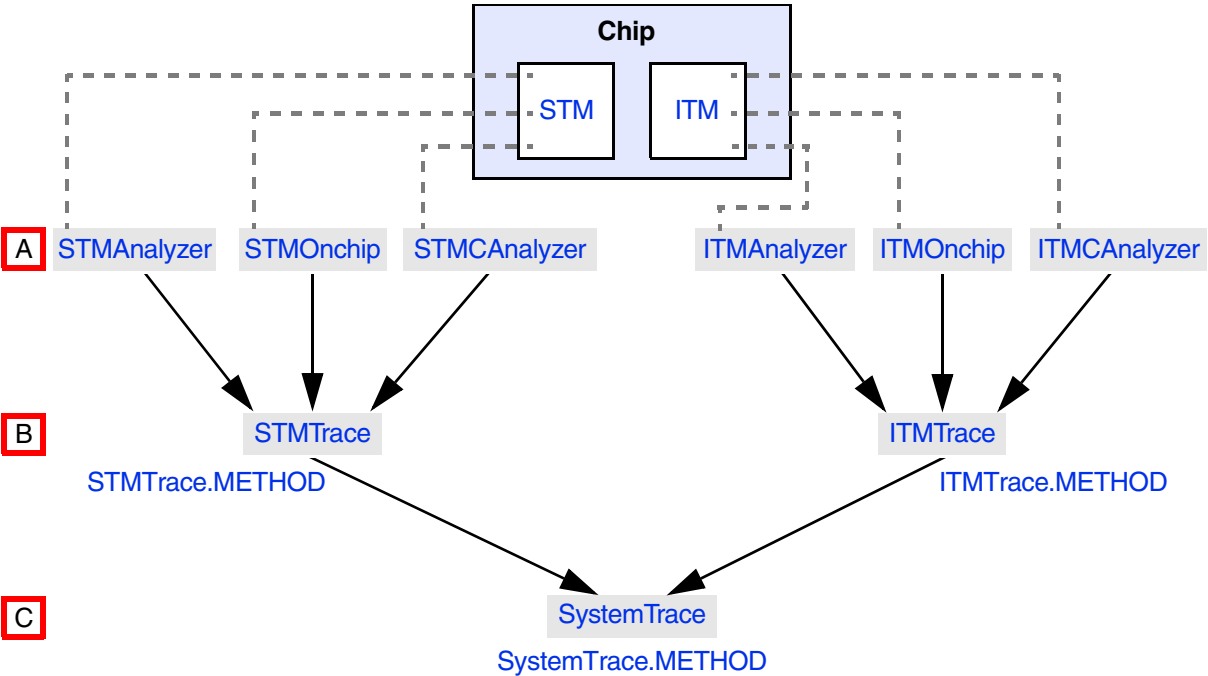
▲ ['Timing Adaption' in 'GTL Debug Back-End'](#)

▲ ['Timing Adaption' in 'Verilog Debug Back-End'](#)

Format: **SystemTrace.<sub_cmd>**

Using the **SystemTrace** command group, you can configure the trace recording as well as analyze and display trace data emitted either by the trace source STM or ITM.

The figure illustrates that there are three ways [A to C] to handle instrumented code from the STM or ITM:



- A The six command groups are distinctive ways to handle STM or ITM trace data. Prior to that you need to set the trace method with **STMTrace.METHOD** or **ITMTrace.METHOD**.
- B Alternatively, the generic replacement command groups **STMTrace** and **ITMTrace** can be used to handle STM or ITM trace data. Prior to that you need to set the trace method with **STMTrace.METHOD** or **ITMTrace.METHOD**.
- C The command group **SystemTrace** lets you handle STM or ITM trace data independently of the trace protocol and trace method in the [SystemTrace.List](#) window. This requires that the trace method has been set with the command **SystemTrace.METHOD**.

Examples for [A] and [B] can be found in sections “[System Trace User’s Guide](#)” (trace_stm.pdf) and “[Overview ITM<trace>](#)” in General Commands Reference Guide I, page 87 (general_ref_i.pdf).

An example for [C] can be found [below](#).

<code><sub_cmd></code>	For descriptions of the subcommands, please refer to the general <code><trace></code> command descriptions in “General Commands Reference Guide T” (general_ref_t.pdf). Example: For a description of SystemTrace.List refer to <code><trace>.List</code>
------------------------------	---

Example 1:

```
SystemTrace.state           ;optional step: open the window in which the
                             ;trace recording is configured.
SystemTrace.METHOD Onchip  ;select the trace method Onchip for
; <configuration>           ;recording system trace data.

STM.state                   ;optional step: open the window in which
                             ;the trace source STM is configured.

STM.ON                      ;switch the trace source STM on.
; <configuration>

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

SystemTrace.List            ;display the system trace data from the STM.
```

NOTE:	The trace method selection for the SystemTrace command group corresponds to the trace method selection for the Trace command group. This becomes obvious when you compare the examples 1 and 2. For background information, see “Types of Replacements for <trace>” in General Commands Reference Guide T, page 121 (general_ref_t.pdf).
--------------	---

Example 2:

```
Trace.state                ;optional step: open the window in which the
                           ;trace recording is configured.
Trace.METHOD Analyzer    ;select the trace method Analyzer for
;<configuration>          ;recording instruction trace data.

ETM.state                 ;optional step: open the window in which
                           ;the trace source ETM is configured.

ETM.ON                    ;switch the trace source ETM on.
;<configuration>

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

Trace.List                 ;display the instruction trace data
                           ;from the ETM.
```

See also

- [STM](#)
- [ITM](#)
- [Trace.METHOD](#)
- ▲ ['Release Information' in 'Legacy Release History'](#)

SystemTrace.state

Open system-trace configuration window

Format: **SystemTrace.state**

Opens the **SystemTrace.state** window, displaying all probe setup parameters.