LAUTERBACH
DEVELOPMENT TOOLS

General Function Reference

Release 09.2023

MANUAL

General Function Reference

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents .....

PRACTICE Script Language .....

TRACE32 Functions .....

General Function Reference **1**

History **24**

In This Document **26**

How This Document is Organized 26

Difference between Functions and Commands in TRACE32 27

Purpose of Functions 28

Example 1: Return Status of the Target 28

Example 2: Return Status of a TRACE32 Tool 28

Example 3: Return the Version Number 28

Example 4: Convert a String 28

How to Use Functions 29

Example 1: In PRACTICE Scripts 29

Example 2: As Parameters in Commands 29

Example 3: Together with the Output Commands PRINT and Data.Print 30

Example 4: Address Function and Their Access Class Specifiers 30

Which Return Values of Functions can be Printed? 31

Related Documents 31

ACCESS Functions **32**

In This Section 32

ACCESS.isGUEST() TRUE if access class belongs to guest 32

ACCESS.isHYPERVISOR() TRUE if access class belongs to hypervisor 33

ADDRESS Functions **34**

In This Section 34

ADDRESS.ACCESS() Access class as ordinal number 34

ADDRESS.ACCESS.CMP() Compare access classes 34

ADDRESS.ACCESS.CMPSTRICT() Compare access classes, strict 35

ADDRESS.EXPANDACCESS() Fully qualified access class 35

ADDRESS.INSTR.LEN() Length of instruction 36

ADDRESS.isDATA() Check if memory class refers to data 36

ADDRESS.isGUEST() TRUE if address is guest address 37

ADDRESS.isHYPERVISOR() TRUE if address is hypervisor address 37
ADDRESS.isINTERMEDIATE() Check if intermediate address 38
ADDRESS.isNONSECURE() TRUE if non-secure (TrustZone) access 38
ADDRESS.isNONSECUREEX() TRUE if non-secure access 39
ADDRESS.MACHINEID() Extract machine ID 39
ADDRESS.MAU() Minimal addressable unit size (MAU) 40
ADDRESS.OFFSET() Address without class 41
ADDRESS.isONCHIP() TRUE if on-chip address area 41
ADDRESS.isPHYSICAL() TRUE if physical address 41
ADDRESS.isPROGRAM() TRUE if program address 42
ADDRESS.isSECURE() TRUE if secure (TrustZone) access 42
ADDRESS.isSECUREEX() TRUE if secure access 43
ADDRESS.RANGE.BEGIN() Lowest address value of address range 44
ADDRESS.RANGE.END() Highest address value of address range 44
ADDRESS.RANGE.SIZE() Size of address range 45
ADDRESS.SEGMENT() Segment of an address 46
ADDRESS.STRACCESS() Access class of an address 46

Analyzer Functions 47

In This Section 48
Analyzer() Check if Analyzer command group is available 48
Analyzer.CONFIG.<powertrace>() Check if specified PowerTrace connected 49
Analyzer.COUNTER.EVENT() Get value of trigger program event counter 49
Analyzer.COUNTER.TIME() Get value of trigger program time counter 50
Analyzer.DSEL() For internal usage only 50
Analyzer.FIRST() Get record number of first trace record 50
Analyzer.FLOW.ERRORS() Get number of flow errors / hard errors 50
Analyzer.FLOW.FIFOFULL() Get number of FIFO overflows 51
Analyzer.FOCUS.EYE() Check quality of data eye 53
Analyzer.ISCHANNELUP() Check if serial link is established 54
Analyzer.MAXSIZE() Get max. size of trace buffer in records 55
Analyzer.MODE() Get Analyzer recording mode 55
Analyzer.MODE.FLOW() Check if Analyzer operates as flowtrace 55
Analyzer.PCIE.CONFIG() Value of register field from PCIe configuration 56
Analyzer.PCIE.ISCONFIGURED() TRUE if prerequisites are fulfilled 57
Analyzer.PCIE.Register() Value of 32-bit register from PCIe configuration 57
Analyzer.PROBEREVISION() Get revision of StarCore NEXUS probe 58
Analyzer.RECORDS() Get number of used trace records 58
Analyzer.RECORD.ADDRESS() Get address recorded in trace record 58
Analyzer.RECORD.DATA() Get data recorded in trace record 59
Analyzer.RECORD.OFFSET() Get address in trace record as number 59
Analyzer.RECORD.TIME() Get timestamp of trace record 60
Analyzer.REF() Get record number of reference record 61
Analyzer.SIZE() Get current trace buffer size in records 61

Analyzer.STATE() Get state of Analyzer 61
Analyzer.THRESHOLD() Get threshold voltage of parallel preprocessor 62
Analyzer.TraceCONNECT() Name of trace sink of the SoC 62
Analyzer.TRACK.RECORD() Get record number matching search 63
Analyzer.TRIGGER.TIME() Time of trigger point in trace 63

ARM Function 65
ARMARCHVERSION() ARM architecture version of CPU 65

Advanced Register Trace (ART) Functions 66
In This Section 66
ART.FIRST() Get record number of first trace record 66
ART.MAXSIZE() Get max. size of trace buffer in records 67
ART.MODE() Get ART recording mode 67
ART.RECORD.ADDRESS() Get address recorded in trace record 67
ART.RECORD.OFFSET() Get address in trace record as number 68
ART.RECORD.TIME() Get timestamp of trace record 68
ART.RECORDS() Get number of used trace records 68
ART.REF() Get record number of reference record 68
ART.SIZE() Get current trace buffer size in records 69
ART.STATE() Get state of ART trace 69
ART.TRACK.RECORD() Get record number matching search 69

AUTOFOCUS Functions 70
In This Section 70
AUTOFOCUS() TRUE if AutoFocus preprocessor attached 70
AUTOFOCUS.OK() TRUE if command execution successful 70
AUTOFOCUS.FREQUENCY() Frequency of trace-port clock 70

AVX Functions 71
In This Section 71
AVX() Content of AVX register 71
AVX512() Content of AVX512 register 72

Break Functions 73
In This Section 73
Break.Alpha.EXIST() TRUE if Alpha breakpoint exists 73
Break.Beta.EXIST() TRUE if Beta breakpoint exist 73
Break.Charly.EXIST() TRUE if Charly breakpoint exists 74
Break.Program.EXIST() TRUE if enabled program breakpoint exists 74
Break.ReadWrite.EXIST() TRUE if enabled data address breakpoint exists 74

BMC Functions (Benchmark Counter) 75
In This Section 75
BMC.CLOCK() Frequency of core clock 75
BMC.COUNTER() Value of a benchmark counter 75
BMC.COUNTER.BYNAME() Value of a benchmark counter 76

BMC.COUNTER.CORE() Value of a benchmark counter 76
BMC.COUNTER.BYNAME.CORE() Value of a benchmark counter 77
BMC.OVERFLOW() TRUE if benchmark counter overflow 77
BMC.OVERFLOW.BYNAME() TRUE if benchmark counter overflow 77
BMC.OVERFLOW.CORE() TRUE if benchmark counter overflow 78
BMC.OVERFLOW.BYNAME.CORE() TRUE if benchmark counter overflow 78

Boundary Scan Description Language (BSDL) Functions 79
In This Section 79
BSDL.CHECK.BYPASS() Chain bypass test 79
BSDL.CHECK.FLASHCONF() Flash configuration test 79
BSDL.CHECK.IDCODE() Chain IDCODE test 79
BSDL.GetDRBit() Data register bit 80
BSDL.GetPortLevel() Port level value 80

CABLE Functions 81
In This Section 81
CABLE.GalvanicISOlation() Cable has galvanic isolation 81
CABLE.GalvanicISOlation.FIRMWARE() Adapter firmware version 81
CABLE.GalvanicISOlation.SERIAL() Serial number of adapter 81
CABLE.NAME() Name of debug cable 82
CABLE.SERIAL() Serial number of debug cable 82
CABLE.TWOWIRE() TRUE if two-wire debugging supported 82

CACHE Functions 83
In This Section 83
CACHE.DC.DIRTY() Dirty-flag of L1 Data Cache Line 83
CACHE.DC.DIRTYMASK() Dirty-flag mask of L1 Data Cache Line 84
CACHE.DC.LRU() LRU information of L1 Data Cache Line 84
CACHE.DC.TAG() Address Tag of L1 Data Cache Line 84
CACHE.DC.VALID() Valid-flag of L1 Data Cache Line 85
CACHE.DC.VALIDMASK() Valid-flag mask of L1 Data Cache Line 85
CACHE.IC.DIRTY() Dirty-flag of L1 Unified Cache Line 86
CACHE.IC.DIRTYMASK() Dirty-flag mask of L1 Unified Cache Line 86
CACHE.IC.LRU() LRU information of L1 Instruction Cache Line 86
CACHE.IC.TAG() Address Tag of L1 Instruction Cache Line 87
CACHE.IC.VALID() Valid-flag of L1 Instruction Cache Line 87
CACHE.IC.VALIDMASK() Valid-flag mask of L1 Instruction Cache Line 87
CACHE.L2.DIRTY() Dirty-flag of L2 Cache Line 88
CACHE.L2.DIRTYMASK() Dirty-flag mask of L2 Cache Line 88
CACHE.L2.LRU() LRU information of L2 Cache Line 88
CACHE.L2.SHARED() Shared-flag of L2 Cache Line 89
CACHE.L2.SHAREDMASK() Shared-flag mask of L2 Cache Line 89
CACHE.L2.TAG() Address Tag of L2 Cache Line 89
CACHE.L2.VALID() Valid-flag of L2 Cache Line 90

CACHE.L2.VALIDMASK() Valid-flag mask of L2 Cache Line 90
CACHE.L3.DIRTY() Dirty-flag of L3 Cache Line 90
CACHE.L3.DIRTYMASK() Dirty-flag of L3 Cache Line 91
CACHE.L3.LRU() LRU information of L3 Cache Line 91
CACHE.L3.TAG() Address Tag of L3 Cache Line 91
CACHE.L3.VALID() Valid-flag of L3 Cache Line 92
CACHE.L3.VALIDMASK() Valid-flag mask of L3 Cache Line 92

CAnalyzer Functions (CombiProbe, ?Trace (MicroTrace)) 93
In This Section 93
CAnalyzer() Check if CAnalyzer command group is available 94
CAnalyzer.BOTHCables() TRUE if both debug cables are plugged 94
CAnalyzer.CableTYPE() Type of adapter 94
CAnalyzer.DebugCable() CombiProbe whisker cable is A or B 95
CAnalyzer.FEATURE() Query features of CAnalyzer hardware 95
CAnalyzer.FIRST() Get record number of first trace record 97
CAnalyzer.MAXSIZE() Get max. size of trace buffer in records 97
CAnalyzer.PIN() Status of trace pins 98
CAnalyzer.RECORD.ADDRESS() Get address recorded in trace record 98
CAnalyzer.RECORD.DATA() Get data recorded in trace record 98
CAnalyzer.RECORD.OFFSET() Get address in trace record as number 99
CAnalyzer.RECORD.TIME() Get timestamp of trace record 99
CAnalyzer.RECORDS() Get number of used trace records 99
CAnalyzer.REF() Get record number of reference record 99
CAnalyzer.SIZE() Get current trace buffer size in records 100
CAnalyzer.STATE() Get state of Compact Analyzer 100
CAnalyzer.TraceCLOCK() Get trace port frequency 101
CAnalyzer.TraceCONNECT() Name of trace sink of the SoC 102
CAnalyzer.TracePort() CombiProbe whisker cable is A or B 102
CAnalyzer.TRACK.RECORD() Get record number matching search 102

CERBEURS Functions 103
CERBERUS.IOINFO() IOINFO of Cerberus module 103
CERBERUS.IOINFO.IFLCK() TRUE if IF_LCK bit in Cerberus INONFO set 103

CHIP Functions 104
CHIP.EmulationDevice() TRUE if emulation device 104
CHIP.STEPping() Major silicon step of an TriCore AURIX device 104

CIProbe Functions (Analog Probe for CombiProbe or µTrace) 105
In This Section 105
CIProbe() TRUE if Compact Analyzer hardware 105
CIProbe.ADC.ENABLE() TRUE if channel is enabled 105
CIProbe.ADC.SHUNT() Get shunt-resistor value 105
CIProbe.MAXSIZE() Get max. size of trace buffer in records 106
CIProbe.RECORDS() Get number of used trace records 106

CIProbe.SIZE() Get current trace buffer size in records 106
CIProbe.STATE() Get state of Compact Analyzer for CIProbe 107
CIProbe.TRACK.RECORD() Get record number matching search 107

CMI Function 108
CMIBASE() Base addresses of CMI modules 108

COMPonent Functions 109
In This Section 109
COMPonent.AVAILABLE() TRUE if debug/trace peripherals available on CPU 109
COMPonent.BASE() Base address of debug/trace peripherals 110
COMPonent.NAME() Name of debug/trace peripherals 110
COMPonent.TYPE() Type of debug/trace peripherals 111

CORE Functions 112
In This Section 112
CONFIGNUMBER() Number of cores configured in TRACE32 113
CORE() Get the selected core 113
CORE.ISACTIVE() TRUE if this core is active 114
CORE.ISASSIGNED() TRUE if physical core is assigned to debug session 115
CORE.LOGICALTOPHYSICAL() This is the physical core number 116
CORE.NAMES() Physical core names assigned to TRACE32 117
CORENAME() Name of core within selected chip 118
CORE.NUMBER() Number of logical cores 118
CORE.PHYSICALTOLOGICAL() Logical core number of physical core 120

Count Functions 121
In This Section 121
Count.Frequency() Frequency of last measurement 121
Count.LEVEL() Level of frequency counter input 121
Count.Time() Time of last measurement 122
Count.VALUE() Samples of the Count.GO command 122

COVerage Functions 123
In This Section 123
COVerage.BDONE() Byte count of all executed instructions 124
COVerage.LOAD.KEY() Key from last ACD file 124
COVerage.SCOPE() Degree of code coverage 125
COVerage.SourceMetric() Active code coverage criterion 126
COVerage.TreeWalk() Walk symbol tree 127

CPU Functions 128
In This Section 128
CPU.ADDRESS() Start address of memory section 128
CPU.ADDRESS.PhysicalINDEX() Section start address of given core 128
CPU.FEATURE() TRUE if CPU feature exists 129
CPU.PINCOUNT() For internal usage only 131

CPUBONDOUT() Name of boundout processor 131
CPUCOREVERSION() Core or architecture version of CPU 131
CPUDERIVATE() Main part of processor name 131
CPUFAMILY() Family name of processor 132
CPUHELP() For internal usage only 132
CPUIS() TRUE if search string matches processor name 133
CPUIS64BIT() TRUE if 64-bit architecture 133

DAP Functions 134
In This Section 134
DAP.Available() TRUE if debugging via DAP is supported 134
DAP.USER<x>() Status of the DAP user pin 134

Data Functions 135
In This Section 135
Data.<value_width>() Memory contents in default endianness 135
Data.<value_width>.<endianness>() Mem. contents in specified byte order 138
Data.<value_width>.<access_width>() Mem. contents in specified width 140
Data.AL.ERRORS() Get number of errors detected by Data.AllocList 141
Data.Float() Get floating point number 141
Data.STRing() Get zero-terminated string 142
Data.STRingN() Get zero-terminated string with a maximum length 143
Data.SUM() Get checksum 143
Data.SWAP.<value_width>.<swap_width>() Swap byte groups in word 144
Data.WSTRING() Get zero-terminated wide string 145
Data.WSTRING.BigEndian() Get big-endian wide string 145
Data.WSTRING.LittleEndian() Get little-endian wide string 146

DEBUGGER Function 147
DEBUGGER.FEATURE() Check debugger feature 147

DEBUGMODE Function 148
DEBUGMODE() Current debug mode 148

DISASSEMBLE Function 149
DISASSEMBLE.ADDRESS() Disassembled instruction at address 149

DONGLEID Function 150
DONGLEID() Serial number of USB WibuKey 150

ELA Function (ARM Coresight Embedded Logic Analyzer) 151
ELABASE() ELA base address 151

DPP Function (C166/ST10 only) 151
DPP() Content of DPP register 151

EPOC Functions 152
In This Section 152
EPOC.DATAADDRESS() Start address of data area (EPOC debugger) 152

EPOC.ENTRYPOINT() Entry address of debug task 152
EPOC.TEXTADDRESS() Start address of code area (EPOC debugger) 152

ERROR Functions (target-dependent) 153
ERROR.ADDRESS() Address of last occurred memory access error 153

ETM Functions 154
In This Section 154
ETM() TRUE if ETM trace is available 154
ETM.ADDRCOMP() For internal usage only 155
ETM.ADDRCOMPTOTAL() Number of ETM address comparator pair 155
ETM.COUNTERS() Number of ETM counters 155
ETM.DATACOMP() Number of ETM data comparators 155
ETM.EXTIN() Number of internal ETM inputs 156
ETM.EXTOUT() Number of external ETM outputs 156
ETM.FIFOFULL() ETM fifofull logic 156
ETM.MAP() Number of ETM memory map decoders 156
ETM.PROTOCOL() Version of ETM protocol 157
ETM.SEQUENCER() Number of ETM sequencers 157
ETM.TraceCore() TRUE if the core is traced 157

EXTENDED Function (Z80 only) 158
EXTENDED() TRUE if register CBAR > 0 158

FDX Function 159
FDX.INSTRING() Content at FDX memory address 159
FDX.TargetSTALLS() Monitor FDX communication stalls on the target 159

FLAG Functions 160
In This Section 160
FLAG() TRUE if hardware flag system available 160
FLAG.READ() FLAG memory bytes with read access bit 160
FLAG.WRITE() FLAG memory bytes with write access bit 160

FLASH Functions 161
In This Section 161
FLASH.CFI.SIZE() Size of FLASH devices 162
FLASH.CFI.WIDTH() Data bus width of FLASH devices 162
FLASH.CLocK.Frequency() FLASH clock value 162
FLASH.ID() FLASH manufacturer and device ID 163
FLASH.List.STATE.PENDING() Number of pending sectors 164
FLASH.List.TYPE() FLASH family code of FLASH list entry 164
FLASH.ProgramMODE() FLASH programming modes 165
FLASH.ProgramMODE.OPTION() FLASH programming options 166
FLASH.SECTOR.BEGIN() Start address 167
FLASH.SECTOR.END() End address 167
FLASH.SECTOR.EXIST() TRUE if sector exists 167

FLASH.SECTOR.EXTRAvalue() Extra value of FLASH.Create 168
FLASH.SECTOR.NEXT() Address of next sector 169
FLASH.SECTOR.OTP() TRUE if OTP sector 169
FLASH.SECTOR.OPTION() Options of a FLASH sector 170
FLASH.SECTOR.RANGE() Address range of a FLASH sector 171
FLASH.SECTOR.SIZE() Size in bytes 171
FLASH.SECTOR.STATE() FLASH programming state 171
FLASH.SECTOR.TYPE() FLASH family code of sector 172
FLASH.SECTOR.WIDTH() Width of FLASH sector 173
FLASH.TARGET.BUILD() Build number of FLASH algorithm file 173
FLASH.TARGET.CODERANGE() Code range of FLASH algorithm 174
FLASH.TARGET.DATARANGE() Data range of FLASH algorithm 174
FLASH.TARGET.FILE() Name of FLASH algorithm file 174
FLASH.UNIT() Unit number of FLASH sector 175
FLASH.UNIT.BEGIN() Unit start address 175
FLASH.UNIT.END() Unit end address 175
FLASH.UNIT.EXIST() TRUE if unit exists 176
FLASH.UNIT.NEXT() Number of next unit 176

FLASHFILE Functions 177
In This Section 177
FLASHFILE.GETBADBLOCK.COUNT() Number of bad blocks 177
FLASHFILE.GETBADBLOCK.NEXT() Address of bad block 177
FLASHFILE.SPAREADDRESS() Address of spare area 178

FPU Functions (Floating Point Unit) 179
In This Section 179
FPU() FPU register contents 179
FPUCR() FPU control register contents 179
FPU.RAW() FPU register raw contents 179

FXU Function 180
FXU() Content of FXU register 180

GROUP Function 180
GROUP.EXIST() TRUE if group exists 180

Hardware Functions 181
In This Section 181
hardware.COMBIPROBE() TRUE if CombiProbe 181
hardware.ESI() TRUE if EPROM Simulator 181
hardware.ICD() TRUE if TRACE32 debug hardware 182
hardware.POWERDEBUG() TRUE if TRACE32 PowerDebug hardware 182
hardware.POWERINTEGRATOR() TRUE if a PowerIntergrator 182
hardware.POWERINTEGRATOR2() TRUE if a PowerIntegrator II 182
hardware.POWERNEXUS() TRUE is a NEXUS Adapter 183
hardware.POWERPROBE() TRUE is a PowerProbe 183

hardware.POWERTRACE() TRUE if a PowerTrace Module 183
hardware.POWERTRACE2() TRUE if a PowerTrace II 183
hardware.POWERTRACE2LITE() TRUE if a PowerTrace II LITE 184
hardware.POWERTRACE3() TRUE if a PowerTrace III 184
hardware.POWERTRACEPX() TRUE if a PowerTrace PX 184
hardware.POWERTRACESERIAL() TRUE if a PowerTrace Serial 184
hardware.POWERTRACESERIAL2() TRUE if a PowerTrace Serial II 185
hardware.QUADPROBE() TRUE if QuadProbe 185
hardware.UTRACE() TRUE if µTrace 185

HVX Function 186
HVX() Content of HVX register 186

I2C Functions 187
In This Section 187
I2C.DATA() Data read by I2C.TRANSFER 187
I2C.PIN() Pin status 187

ID Functions 188
In This Section 188
ID.CABLE() Hardware ID of debug cable 188
ID.PREPROcessor() Hardware ID of preprocessor 188
ID.SERialPort1() Type-ID of Adapter or Preprocessor at PowerTrace Serial 189
ID.WHISKER() ID of whisker cable 190
IDCODE() ID code of TAP in JTAG chain 193
IDCODENUMBER() Number of detected TAPs 193

Integrator Functions 194
In This Section 194
Integrator() TRUE if PowerIntegrator 194
Integrator.FIRST() Get record number of first trace record 194
Integrator.ADC.ENABLE() Bitmask of enabled analog channels 195
Integrator.ADC.SHUNT() Shunt-resistor value 195
Integrator.ANALOG() 195
Integrator.COUNTER.EVENT() Get value of trigger program event counter 195
Integrator.COUNTER.EXTERN() Value of trigger program external counter 196
Integrator.COUNTER.TIME() Get value of trigger program time counter 196
Integrator.DIALOGDSEL() For internal usage only 196
Integrator.DIALOGDSELGET() For internal usage only 196
Integrator.DSEL() For internal usage only 197
Integrator.FIND.PI_CHANNEL() For internal usage only 197
Integrator.FIND.PI_WORD() TRUE if signal word is defined 197
Integrator.FLAG() Check state of trigger program FLAG 197
Integrator.GET() Value of channel 198
Integrator.MAXSIZE() Get max. size of trace buffer in records 198
Integrator.PROBE() For internal usage only 198

Integrator.PROGRAMFILENAME() File name of trigger program 198
Integrator.RECORD.DATA() Get data recorded in trace record 199
Integrator.RECORD.TIME() Get timestamp of trace record 199
Integrator.RECORDS() Get number of used trace records 199
Integrator.REF() Get record number of reference record 199
Integrator.SIZE() Get current trace buffer size in records 200
Integrator.STATE() Get state of the Integrator 200
Integrator.TRACK.RECORD() Get record number matching search 200
Integrator.USB() For internal usage only 201

INTERFACE Functions 202
In This Section 202
INTERFACE.CADI() TRUE if connection to target is via CADI interface 202
INTERFACE.GDB() TRUE if connection to target is via GDB interface 203
INTERFACE.GDI() TRUE if connection to target via GDI interface 203
INTERFACE.HOST() TRUE if application is debugged on host 203
interface.HOSTMCI() TRUE if TRACE32 debug driver runs on host 203
INTERFACE.IRIS() TRUE if connection to target is via IRIS interface 204
INTERFACE.MCD() TRUE if connection to target via MCD interface 204
INTERFACE.NAME() Name of debugger 204
INTERFACE.QNX() TRUE if PBI=QNX 204
INTERFACE.SIM() TRUE if simulator 205

IOBASE Functions 206
In This Section 206
IOBASE() Base address of internal I/O's 206
IOBASE.ADDRESS() Base address of internal I/O's with access class 206
IOBASE2() Second base address of internal I/O's 206

IProbe Functions 207
In This Section 208
IProbe() TRUE if IPROBE 208
IProbe.ADC.ENABLE() TRUE if channel is enabled 208
IProbe.ADC.SHUNT() Shunt resistor value of channel 209
IProbe.ANALOG() TRUE if Analog Probe is plugged 210
IProbe.FIRST() Get record number of first trace record 210
IProbe.GET() Value of channel 210
IProbe.MAXSIZE() Get max. size of trace buffer in records 211
IProbe.PROBE() 211
IProbe.RECORD.DATA() Get data recorded in trace record 212
IProbe.RECORD.TIME() Get timestamp of trace record 212
IProbe.RECORDS() Get number of used trace records 213
IProbe.REF() Get record number of reference record 214
IProbe.SIZE() Get current trace buffer size in records 214
IProbe.STATE() Get state of IProbe 215

IProbe.TRACK.RECORD() Get record number matching search 215

JTAG Functions 216
In This Section 216
JTAG.MIPI34() Query special MIPI34 pins 216
JTAG.PIN() Level of JTAG signal 217
JTAG.SEQuence.RESULT() Get result of JTAG sequence 217
JTAG.SEQuence.EXIST() Check if a JTAG sequence exists 217
JTAG.SEQuence.LOCKED() Check if a JTAG sequence is locked 218
JTAG.SHIFT() TDO output of JTAG shift 218
JTAG.X7EFUSE.RESULT() Result of JTAG.X7EFUSE command 219
JTAG.X7EFUSE.CNTL() CNTL flags read by JTAG.X7EFUSE command 220
JTAG.X7EFUSE.DNA() DNA value read by JTAG.X7EFUSE command 220
JTAG.X7EFUSE.KEY() AES key read by JTAG.X7EFUSE command 221
JTAG.X7EFUSE.USER() User code read by JTAG.X7EFUSE command 221
JTAG.XUSEFUSE.RESULT() Result of JTAG.XUSEFUSE command 222
JTAG.XUSEFUSE.CNTL() CNTL value read by JTAG.XUSEFUSE command 222
JTAG.XUSEFUSE.DNA() DNA value read by JTAG.XUSEFUSE command 223
JTAG.XUSEFUSE.KEY() AES key read by JTAG.XUSEFUSE command 223
JTAG.XUSEFUSE.RSA() RSA hash read by JTAG.XUSEFUSE command 224
JTAG.XUSEFUSE.SEC() SEC value read by JTAG.XUSEFUSE command 224
JTAG.XUSEFUSE.USER() User code read by JTAG.XUSEFUSE command 225
JTAG.XUSEFUSE.USER128() 128 bit User code read by JTAG.XUSEFUSE 225

LOGGER Functions 226
In This Section 226
LOGGER.FIRST() Get record number of first trace record 226
LOGGER.RECORD.ADDRESS() Get address recorded in trace record 227
LOGGER.RECORD.DATA() Get data recorded in trace record 227
LOGGER.RECORD.OFFSET() Get address in trace record as number 227
LOGGER.RECORD.TIME() Get timestamp of trace record 228
LOGGER.RECORDS() Get number of used trace records 228
LOGGER.REF() Get record number of reference record 228
LOGGER.SIZE() Get current trace buffer size in records 228
LOGGER.STATE() Get state of Logger trace 229

MachO Format Function (Apple) 230
MACHO.LASTUUID() Universally unique identifier of MachO file 230

MAP Functions 231
In This Section 231
MAP.ROMSIZE() Size of the defined ROM 231

MCDS Functions 232
In This Section 232
MCDS.MODULE.NAME() Name of MCDS module 233
MCDS.MODULE.NUMBER() Number-part of MCDS module ID 233

MCDS.MODULE.REVision() Revision-part of MCDS module ID 233
MCDS.MODULE.TYPE() Type-part of MCDS module ID 234
MCDS.STATE() MCDS module is switched on/off 234
MCDS.TraceBuffer.LowerGAP() Trace buffer lower gap 235
MCDS.TraceBuffer.SIZE() Trace buffer size 236
MCDS.TraceBuffer.UpperGAP() Trace buffer upper gap 236

MMU Functions (Memory Management Unit) 237
In This Section 237
MMU() Value of MMU register 237
MMU.DEFAULTPT() Base address of default page table 238
MMU.DEFAULTTRANS.<range>() Query MMU setup 239
MMU.FORMAT() Currently selected MMU format 241
MMU.FORMAT.DETECTED() Auto-detection of page table format 242
MMU.FORMAT.DETECTED.ZONE() Auto-detection of page table format 243

MMX Function (MultiMedia eXtension) 244
MMX() Value of MMX register 244

MONITOR Function 244
MONITOR() TRUE if debugger is running as monitor 244

NEXUS Functions 245
In This Section 245
NEXUS() TRUE if Nexus trace is supported 245
NEXUS.RTTBUILD() RTT build register 245
NEXUS.PortMode() Current PortMode setting 246
NEXUS.PortSize() Current PortSize setting 246

Onchip Functions 247
In This Section 247
Onchip() TRUE if the onchip trace is available 247
Onchip.FIRST() Get record number of first trace record 247
Onchip.FLOW.ERRORS() Get number of flow errors / hard errors 247
Onchip.FLOW.FIFOFULL() Get number of FIFO overflows 248
Onchip.MAXSIZE() Get max. size of trace buffer in records 249
Onchip.RECORD.ADDRESS() Get address recorded in trace record 249
Onchip.RECORD.DATA() Get data recorded in trace record 249
Onchip.RECORD.OFFSET() Get address in trace record as number 249
Onchip.RECORD.TIME() Get timestamp of trace record 250
Onchip.RECORDS() Get number of used trace records 250
Onchip.REF() Get record number of reference record 250
Onchip.SIZE() Get current trace buffer size in records 250
Onchip.STATE() Get state of Onchip trace 251
Onchip.TraceCONNECT() Name of trace sink of the SoC 251
Onchip.TRACK.RECORD() Get record number matching search 252

PBI Function **253**
PBI() Name of used debug back-end 253

PCI Functions **254**
In This Section 254
PCI.Read.B() Byte from PCI register 254
PCI.Read.L() Long from PCI register 254
PCI.Read.W() Word from PCI register 254

PER Functions **255**
In This Section 255
PER.<width>() Memory contents in default endianness 255
PER.<width>.<endianness>() Memory contents in specified endianness 256
PER.ADDRESS() Address of register(field) 257
PER.ADDRESS.<sub_cmd>() Check access security in PER file 258
PER.ARG() Argument of PER.view command 258
PER.ARG.ADDRESS() Address argument of PER.view command 259
PER.BASE() Last BASE address 259
PER.Buffer.<width>() Value from buffer 260
PER.EVAL() Value of expression in PER file 261
PER.FILENAME() PER file name 261
PER.SAVEINDEX() Value from indexed register 262
PER.VALUE() Value of register(field) 262
PER.VALUE.STRING() Value of BITFLD as string 263

PERF Functions (Performance) **264**
In This Section 264
PERF.MEMORY.HITS() Number of memory samples 264
PERF.MEMORY.SnoopAddress() Snoop memory address 265
PERF.MEMORY.SnoopSize() Snoop size 265
PERF.METHOD() Recording method 265
PERF.MODE() Get Performance Analyzer recording mode 266
PERF.PC.HITS() Number of PC samples 266
PERF.RATE() Number of snoops per second 266
PERF.RunTime() Retained time for program run 267
PERF.SNOOPFAILS() Number of snoop fails 267
PERF.STATE() Get state of Performance Analyzer 267
PERF.TASK.HITS() Number of task samples 268

Port Analyzer Functions **269**
In This Section 269
PORT.GET() Value of channel 269
PORT.MAXSIZE() Get max. size of trace buffer in records 269
PORT.RECORDS() Get number of used trace records 269
PORT.REF() Get record number of reference record 270
PORT.SIZE() Get current trace buffer size in records 270

PORT.STATE() Get state of Port Analyzer 270
PORT.TRACK.RECORD() Get record number matching search 270
PORTANALYZER() 271

PORTSHARING Function 271
PORTSHARING() Current setting of PortSHaRing 271

POWER Functions 272
In This Section 272

PowerProbe Functions 273
In This Section 273
PROBE.COUNTER.EVENT() Get value of trigger program event counter 273
PROBE.COUNTER.EXTERN() Get value of trigger program external counter 273
PROBE.COUNTER.TIME() Get value of trigger program time counter 273
Probe.FIRST() Get record number of first trace record 274
PROBE.FLAG() Check state of trigger program FLAG 274
PROBE.GET() Value of channel 274
PROBE.MAXSIZE() Get max. size of trace buffer in records 274
PROBE.RECORD.DATA() Get data recorded in trace record 275
PROBE.RECORD.TIME() Get timestamp of trace record 275
PROBE.RECORDS() Get number of used trace records 275
PROBE.REF() Get record number of reference record 275
PROBE.SIZE() Get current trace buffer size in records 276
PROBE.STATE() Get state of PowerProbe 276
PROBE.TRACK.RECORD() Get record number matching search 277

Program Pointer Function 278
PP() Address of program pointer (access class, space ID, program counter) 278

Register Functions 279
Register() Content of register 279
Register.LIST() First / next register name 280
Register.Valid() Valid register value 281

RTS Functions 282
In This Section 282
RTS.ERROR() Check for flowerrors during RTS processing 282
RTS.NOCODE() Check for RTS NOCODE error 282
RTS.FIFOFULL() Check for FIFO full error in RTS 283
RTS.RECORD() Find record causing an error in RTS 283
RTS.RECORDS() Get number of trace records transferred to RTS 283
RTS.BUSY() Check if RTS is busy 283

RunTime Functions 284
In This Section 284
RunTime.ACCURACY() Accuracy of run-time counter 284
RunTime.ACTUAL() 284

RunTime.LAST() 284
RunTime.LASTRUN() 285
RunTime.REFA() 285
RunTime.REFB() 285

SMMU Functions 286
SMMU.BaseADDRESS() Base address of SMMU 286
SMMU.StreamID2SMRG() Find match for stream ID 286

SNOOPer Functions 288
In This Section 288
SNOOPer.FIRST() Get record number of first trace record 288
SNOOPer.MAXSIZE() Get max. size of trace buffer in records 289
SNOOPer.RECORD.ADDRESS() Get address recorded in trace record 289
SNOOPer.RECORD.DATA() Get data recorded in trace record 289
SNOOPer.RECORD.OFFSET() Get address in trace record as number 289
SNOOPer.RECORD.TIME() Get timestamp of trace record 290
SNOOPer.RECORDS() Get number of used trace records 290
SNOOPer.REF() Get record number of reference record 290
SNOOPer.SIZE() Get current trace buffer size in records 290
SNOOPer.STATE() Get state of SNOOPer trace 291

STATE Functions (Target State) 292
In This Section 292
STATE.HALT() 292
STATE.OSLK() 292
STATE.POWER() 293
STATE.PROCESSOR() 294
STATE.RESET() 295
STATE.RUN() 295
STATE.TARGET() State of target displayed in TRACE32 state line 295

SPE Function 295
SPE() Content from SPE register 295

SSE Function 296
SSE() Segment from SSE register 296

Stimuli Generator Function 297
hardware.STG() TRUE if Stimuli Generator hardware 297

sYmbol Functions 298
In This Section 298
sYmbol.AutoLOAD.CHECK() Update option for the symbol autoloader 298
sYmbol.AutoLOAD.CHECKCMD() Load command for symbol autoloader 298
sYmbol.AutoLOAD.CONFIG() Used sub-command 299
sYmbol.BEGIN() First address of symbol 299
sYmbol.COUNT() Number of symbols 300

sYmbol.END() Last address of symbol 300
sYmbol.EPILOG() Address of return point 301
sYmbol.EXIST() TRUE if symbol exists 302
sYmbol.EXIT() Exit address of function 302
sYmbol.FUNCTION() Function name 303
sYmbol.IMPORT() Import file names 303
sYmbol.ISFUNCTION() TRUE if symbol is function 303
sYmbol.ISVARIABLE() TRUE if symbol is variable 304
sYmbol.LANGUAGE() Selected high-level language 305
sYmbol.List.MAP.<x>() Information about address ranges on the target 305
sYmbol.LIST.PROGRAM() Path and file name of binary files 306
sYmbol.List.PROGRAM.<x>() Information about loaded programs 307
sYmbol.List.SECtion.<x>() Information about section ranges 308
sYmbol.LIST.SOURCE() File location of source file 310
sYmbol.MATCHES() Number of occurrences 310
sYmbol.NAME() Symbol path and name based on address 311
sYmbol.NAME.AT() Resolve ambiguous symbols based on address 311
sYmbol.NEXT.BEGIN() Start address of next symbol 312
sYmbol.RANGE() Address range of symbol 312
sYmbol.SEARCHFILE() Absolute path of source file 312
sYmbol.SECADDRESS() Start address of section 314
sYmbol.SECEND() End address of section 314
sYmbol.SECEXIST() Check for existence of a section 314
sYmbol.SECNAME() Section name 315
sYmbol.SECPRANGE() Physical address range of section 315
sYmbol.SECRANGE() Logical address range of section 315
sYmbol.SIZEOF() Size of debug symbol 316
sYmbol.SOURCEFILE() Name of source file 316
sYmbol.SOURCELINE() HLL-line number of address 317
sYmbol.SOURCEPATH() TRUE if path is search path 318
sYmbol.STATE() Value from sYmbol.state window 318
sYmbol.TRANSPOSE() Transpose program and module names 318
sYmbol.TYPE() Type of symbol 319
sYmbol.VARNAME() Name of variable or structure element 320

SYStem Functions 321

In This Section 321
SYStem.ACCESS.DENIED() TRUE if memory access is denied 322
SYStem.AMBA() TRUE if AMBA bus mode is selected 322
SYStem.BigEndian() TRUE if target core runs in big endian mode 322
SYStem.CADIconfig.RemoteServer() 323
SYStem.CADIconfig.Traceconfig() 324
SYStem.CONFIG.<tap_position>() 325
SYStem.CONFIG.DEBUGPORT() 325

SYStem.CONFIG.DEBUGPORTTYPE() 325
SYStem.CONFIG.JTAGTAP() Return the JTAG PRE and POST settings 326
SYStem.CONFIG.ListCORE() 329
SYStem.CONFIG.ListSIM() 330
SYStem.CONFIG.Slave() 330
SYStem.CONFIG.TAPState() 331
SYStem.CPU() Name of processor 331
SYStem.GTL.CALLCOUNTER() Amount of calls to GTL library 332
SYStem.GTL.CONNECTED() Connection status 332
SYStem.GTL.CYCLECOUNTER() load GTL interface for bit banging protocol 332
SYStem.GTL.LIBname() Name of GTL library 332
SYStem.GTL.ModelINFO() Info string from GTL API 333
SYStem.GTL.ModelNAME() Model Name 333
SYStem.GTL.PLUGINVERSION() Version number 333
SYStem.GTL.TransactorNAME() Transactor name 334
SYStem.GTL.TransactorTYPE() Transactor type 334
SYStem.GTL.VENDORID() Vendor ID 334
SYStem.GTL.VERSION() Version number 335
SYStem.HOOK() 335
SYStem.IMASKASM() 335
SYStem.IMASKHLL() 335
SYStem.INSTANCE() Index of TRACE32 PowerView instance 336
SYStem.INSTANCECOUNT() Count of GUIs connected to a PowerDebug 336
SYStem.IRISconfig.RemoteServer() 337
SYStem.JtagClock() 337
SYStem.LittleEndian() 337
SYStem.MCDCommand.ResultString() 338
SYStem.MCDconfig.LIBrary() 338
SYStem.Mode() 338
SYStem.NOTRAP() 1 if the option NOTRAP is active 339
SYStem.Option.DUALPORT() State of like-named command 339
SYStem.Option.MACHINESPACES() State of like-named command 339
SYStem.Option.MMUSPACES() State of like-named command 340
SYStem.Option.EnReset() State of like-named command 340
SYStem.Option.ResBreak() State of like-named command 340
SYStem.Option.SPILLLOCation() State of like-named command 341
SYStem.Option.ZoneSPACES() State of like-named command 341
SYStem.RESetBehavior() Current setting of RESetBehavior 342
SYStem.Up() TRUE if debugger has access to debug resources 342
SYStem.USECORE() 344
SYStem.USEMASK() 345

TASK Functions 346

In This Section 346

TASK() Name of current task 347
TASK.ACCESS() Access class 347
TASK.ACCESS.ZONE() Access class zone 347
TASK.BACK() Background task number 347
TASK.CONFIG() OS Awareness configuration information 348
TASK.CONFIGFILE() Path of loaded OS Awareness 348
TASK.COUNT() Number of tasks 348
TASK.CURRENT.MACHINEID() ID of current machine 349
TASK.CURRENT.SPACEID() ID of current MMU space 349
TASK.CURRENT.TASK() Magic value of current task 349
TASK.CURRENT.TASKNAME() Name of current task 349
TASK.FIRST() First task in list 350
TASK.FORE() Foreground task number 350
TASK.ID() ID of task 350
TASK.MACHINE.ACCESS() Default access class 350
TASK.MACHINE.ID() ID of machine 351
TASK.MACHINE.NAME() Name of machine 351
TASK.MACHINE.VTTB() VTTB of machine 352
TASK.MAGIC() Task magic number 352
TASK.MAGICADDRESS() 'magic address' 353
TASK.MAGICRANGE() Range of 'magic address' 353
TASK.MAGICSIZE() Size of 'magic address' 353
TASK.NAME() Name of task 354
TASK.NEXT() Next task in list 354
TASK.ORTIFILE() Path of loaded ORTI file 355
TASK.SPACE.COUNT() Number of spaces 355
TASK.SPACEID() Space ID of task 356

TERM Functions (Terminal Window) 357
In This Section 357
TERM.LINE() Get line from terminal window 357
TERM.READBUSY() TRUE as long as TERM.READ is in progress 357
TERM.RETURNCODE() Get returncode from terminal routine 358
TERM.TRIGGERED() Get trigger state of terminal window 358

TPIU Functions 360
In This Section 360
TPIU.PortMode() Port mode setting 360
TPIU.PortSize() Port size setting 360
TPIU.SWVPrescaler() SWVPrescaler value 361

TPUBASE Function 361
TPUBASE.ADDRESS() Address of TPU 361

Trace Functions 362
In This Section 362

Trace.FIRST() Get record number of first trace record 362
Trace.FLOW() TRUE if trace method is flow trace 363
Trace.FLOW.ERRORS() Get number of flow errors / hard errors 363
Trace.FLOW.FIFOFULL() Get number of FIFO overflows 364
Trace.MAXSIZE() Get max. size of trace buffer in records 364
Trace.METHOD() Currently configured trace method 365
Trace.METHOD.Analyzer() TRUE if the trace method is Analyzer 365
Trace.METHOD.ART() TRUE if the trace method is ART 365
Trace.METHOD.CAnalyzer() TRUE if the trace method is CAnalyzer 365
Trace.METHOD.FDX() TRUE if the trace method is FDX 366
Trace.METHOD.HAnalyzer() TRUE if the trace method is HAnalyzer 366
Trace.METHOD.Integrator() TRUE if the trace method uses the Integrator 366
Trace.METHOD.IProbe() TRUE if the trace method uses the IProbe 366
Trace.METHOD.LA() TRUE if the trace method is LA 367
Trace.METHOD.LOGGER() TRUE if the trace method is LOGGER 367
Trace.METHOD.ONCHIP() TRUE if the trace method is ONCHIP 367
Trace.METHOD.Probe() TRUE if trace method uses the PowerProbe 367
Trace.METHOD.SNOOPer() TRUE if the trace method is SNOOPer 368
Trace.RECORD.ADDRESS() Get address recorded in trace record 368
Trace.RECORD.DATA() Get data recorded in trace record 368
Trace.RECORD.OFFSET() Get address in trace record as number 369
Trace.RECORD.TIME() Get timestamp of trace record 369
Trace.RECORDS() Get number of used trace records 369
Trace.SIZE() Get current trace buffer size in records 370
Trace.STATE() Get state of PowerTrace hardware 370
Trace.STATistic.COUNT() Number of occurences of selected function 371
Trace.STATistic.EXIST() TRUE if function exists in trace statistics 371
Trace.STATistic.IMAX() Longest time between function entry and exit 371
Trace.STATistic.IMIN() Shortest time between function entry and exit 371
Trace.STATistic.Internal() Time spent within the selected function 372
Trace.STATistic.MAX() Maximum time of selected function 372
Trace.STATistic.MIN() Minimum time of selected function 372
Trace.STATistic.Total() Total time of selected function 372
Trace.TraceCONNECT() Name of trace sink of the SoC 373

TRACEPORT Function 374
In This Section 374
TRACEPORT.LaneCount() Number of serial lanes 374

TRACK Functions 375
In This Section 375
TRACK.ADDRESS() Get tracking address 375
TRACK.COLUMN() Number of column where the found item starts 375
TRACK.LINE() Number of line containing the found item 375
TRACK.RECORD() Number of record containing the found item 377

TRACK.STRing() Current selection in a TRACE32 window 377
TRACK.TIME() Timestamp of current tracking record 378

TRANS Functions (Debugger Address Translation) 379
In This Section 379
TRANS.LIST.NUMBER() Number of TRANS.List entries 379
TRANS.LIST.LOGRANGE() Query TRANS.List entry 380
TRANS.LIST.PHYSADDR() Query TRANS.List entry 381
TRANS.LIST.TYPE() Query TRANS.List entry 382
TRANS.ENABLE() TRUE if address translation is enabled 383
TRANS.INTERMEDIATE() Convert a guest logical address 383
TRANS.INTERMEDIATE.VALID() TRUE if address translation is valid 384
TRANS.LINEAR() Convert logical to linear address 384
TRANS.LINEAR.VALID() TRUE if address translation is valid 385
TRANS.LOGICAL() Convert physical to logical address 385
TRANS.LOGICAL.VALID() TRUE if address translation is valid 386
TRANS.PHYSICAL() Convert logical to physical address 386
TRANS.PHYSICAL.VALID() TRUE if address translation is valid 389
TRANS.TABLEWALK() TRUE if address translation table walk is ON 389

TSS Function 390
TSS() TSS base address 390

Var Functions 391
In This Section 391
Var.ADDRESS() Address of HLL expression 391
Var.BITPOS() Bit position inside a C bit field 391
Var.BITSIZE() Size of bit field element 392
Var.END() Last address of HLL expression 392
Var.EXIST() TRUE if HLL expression exists 393
Var.FVALUE() Contents of HLL expression 394
Var.ISBIT() TRUE if HLL expression is a bit field element 394
Var.RANGE() Address range of HLL expression 395
Var.SIZEOF() Size of HLL expression 395
Var.STRing() Zero-terminated string or variable contents 396
Var.TYPEOF() Type of HLL expression 396
Var.VALUE() Value of HLL expression 397

VCO Function 398
VCO() Frequency of VCO generator 398

VERSION Functions 399
In This Section 399
VERSION.BUILD() Upper build number 400
VERSION.BUILD.BASE() Lower build number 400
VERSION.CABLE() Hardware version of debug cable 401
VERSION.DATE() Version date YYYY/MM 401

VERSION.ENVironment() TRACE32 environment setting 401
VERSION.FirmWare.DEBUG() Version number of firmware 402
VERSION.LICENSE() Deprecated 402
VERSION.SERIAL() Serial number 402
VERSION.SERIAL.CABLE() First serial number of debug cable 402
VERSION.SERIAL.DEBUG() Serial number of debug module 403
VERSION.SERIAL.Integrator() Serial number of PowerIntegrator 403
VERSION.SERIAL.NEXUSadapter() Serial number of nexus adapter 403
VERSION.SERIAL.PREPROcessor() Serial number of preprocessor 403
VERSION.SERIAL.POWERPROBE() Serial number of PowerProbe 404
VERSION.SERIAL.POWERTRACEAUXPORT() S/N of device at PT aux port 404
VERSION.SERIAL.SERialPort1() S/N of device at Serial Port 1 of PT Serial 404
VERSION.SERIAL.WHISKER() S/N of whiskers at CombiProbe or µTrace 404
VERSION.SERIAL.TRACE() Serial number of trace module 405
VERSION.SOFTWARE() Release build or nightly build, etc. 406
VERSION.SOFTWARE.TYPE() Software build type 407

VPU Functions 408

In This Section 408
VPU() Value of VPU register 408
VPUCR() Value of VRSAVE or VSCR register 408

General Function Reference

Version 09-Oct-2023

History

21-Aug-2023 New function Register.Valid().

06-Jun-2023 New function VERSION.SOFTWARE.TYPE().

05-Jun-2023 New function PER.BASE().

02-May-2023 New functions: Analyzer.CONFIG.POWERTRACESERIAL2() and hardware.POWERTRACESERIAL2().

18-Apr-2023 New optional parameter for Symbol.SOURCEFILE().

20-Feb-2023 New functions: ID.SERialPort1() and VERSION.SERIAL.SERialPort1().

18-Nov-2022 New function: PBI().

18-Nov-2022 New functions: VERSION.SERIAL.Integrator, VERSION.SERIAL.NEXUSadapter, and VERSION.SERIAL.POWERPROBE.

02-Nov-2022 All **hardware** functions are now grouped in the chapter 'Hardware Functions'. The descriptions have also been updated.

22-Aug-2022 New **THUMB** parameter for CPU.FEATURE() function supported by the Arm architecture.

09-May-2022 New functions: SYStem.GTL.CALLCOUNTER() and SYStem.GTL.CYCLECOUNTER().

09-May-2022 New function: FDX.TargetSTALLS().

06-May-2022 New functions: PER.ADDRESS(), PER.VALUE(), and PER.VALUE.STRING().

08-Apr-2022 Added optional parameters to the functions TASK.MAGICADDRESS(), TASK.MAGICRANGE(), and TASK.MAGICSIZE().

01-Apr-2022 Removed functions: **hardware.SCU()**, **hardware.TA32()**, **SYStem.TRACEEXT()**, **SYStem.TRACEINT()**, and **Analyzer.CONFIG.RISCTRACE()**.

30-Mar-2022 New function: sYmbol.TRANSPOSE().

09-Mar-2022 New function: COVerage.LOAD.KEY().

04-Mar-2022 New function: ETM.TraceCORE().

04-Mar-2022 New function: CABLE.GalvanicISOlation.FIRMWARE().

24-Feb-2022 Removed functions: **INTERFACE.VAST()** and **INTERFACE.VDI()**.

10-Feb-2022 New function: COMPonent.TYPE().

03-Feb-2022 New function: TERM.READBUSY().

19-Jan-2022 New function: PER.SAVEINDEX().

21-Dec-2021 New functions: sYmbol.List.PROGRAM.<x>(), and sYmbol.List.SECtion.<x>().

01-Nov-2021 New functions: SYStem.GTL.ModelINFO(), SYStem.GTL.ModelNAME(), SYStem.GTL.TransactorNAME(), SYStem.GTL.TransactorTYPE(), and DEBUGGER.FEATURE().

29-Jul-2021 New functions: PER.ADDRESS.isNONSECURE(), PER.ADDRESS.isNONSECUREEX(), PER.ADDRESS.isSECURE(), and PER.ADDRESS.isSECUREEX().

12-Jul-2021 New function: SYStem.MCDCommand.ResultString().

31-May-2021 New functions: hardware.POWERTRACE3(), and Analyzer.CONFIG.POWERTRACE3().

24-May-2021 New function: SYStem.Option.ResBreak().

18-May-2021 New functions: sYmbol.ISFUNCTION(), and sYmbol.ISVARIABLE().

08-Apr-2021 New function: SYStem.Option.EnReset().

25-Mar-2021 New functions: VERSION.SERIAL.POWERTRACEAUXPORT(), and VERSION.SERIAL.WHISKER().

In This Document

This document lists all the *target-related functions* and *tool-related functions* available for the different debug systems.

In addition, the document provides important background information about functions in TRACE32 and explains the purpose and use of functions in TRACE32.

The capital letters in function names represent the short form of the function. Any function name can be written in its long or short form. For example **D**ata**.B**yte() could be also written as **D.B**()

How This Document is Organized

- **Difference between Functions and Commands in TRACE32**: This section is primarily intended for users who are new to TRACE32. As a new user, make sure that you also read the next two sections.
- **Purpose of Functions**: Briefly describes and illustrates the purpose of functions in TRACE32 by way of four examples.
- **How to Use Functions**: Briefly describes and illustrates how to use functions in TRACE32 by way of four examples.
- **Which return values of functions can be printed?**: Provides background information.
- **Groups of Functions** (e.g. **ACCESS Functions**, **ADDRESS Functions** etc.): Describes the *target-related* and *tool-related functions*, including source code examples, screenshots, and photos.

Difference between Functions and Commands in TRACE32

In TRACE32, functions are *not* the same as commands. Commands are used to perform actions, e.g. open a window, modify configuration settings, etc. Whereas functions are used to return information about hard and software and convert formats and data.

Functions have trailing parentheses(), whereas commands are used without parentheses:

Function()	Command
`SYStem.UP()`	`SYStem.state`

Functions and commands can have identical names to emphasize that they are related. But this is not always the case, as the example below shows:

Related function and command	Function()	Command
identical names	`Analyzer.SIZE()`	`Analyzer.SIZE <records>`
not identical names	`STATE.RUN()`	`Go`

Next:

- **Purpose of Functions**
- **How to Use Functions**

Purpose of Functions

Functions in TRACE32 have two main purposes:

1. Return status information about:
   - The target (see **Example 1**)
   - The TRACE32 tools (see **Example 2**)
   - The TRACE32 software (see **Example 3**)
2. Convert specific formats or data into other formats or data (see **Example 4**).

Example 1: Return Status of the Target

Returns the status of the target:

```
PRINT STATE.RUN()                   ; Returns the status of the run-flag
                                    ; as a boolean.
                                    ; TRUE: cpu is running in the target
                                    ; (or is out of control).
                                    ; FALSE: cpu is not running.
```

Example 2: Return Status of a TRACE32 Tool

Returns the status of a TRACE32 tool, here the serial cable:

```
PRINT VERSION.SERIAL.CABLE()        ; Returns the serial number of
                                    ; the debug license
                                    ; (Nexus adapter or debug cable).
```

Example 3: Return the Version Number

Returns the version number of the TRACE32 software:

```
PRINT VERSION.SOFTWARE()            ; Returns the version number of
                                    ; your TRACE32 installation.

PRINT SOFTWARE.VERSION()            ; Alternative command
```

Example 4: Convert a String

Convert a string to upper case:

```
PRINT STRing.UPR("hello world")     ; Converts the string to HELLO WORLD
```

How to Use Functions

In TRACE32, you can use functions as follows:

1. Within PRACTICE scripts (see **Example 1**)
2. To parametrize commands (see **Example 2**)
3. Type them directly into the TRACE32 command line
   - **Example 3** points out the importance of the output commands **PRINT** and **Data.Print**
   - **Example 4** points out the importance of access class specifiers for address functions.

Example 1: In PRACTICE Scripts

Functions are normally used within PRACTICE scripts:

```
...
; verify the FLASH contents
Data.LOAD.COSMIC demo.h12 /DIFF
IF FOUND()
     PRINT "Verify error after FLASH programming"
ELSE
     PRINT "FLASH programming completed successfully"
...
```

Example 2: As Parameters in Commands

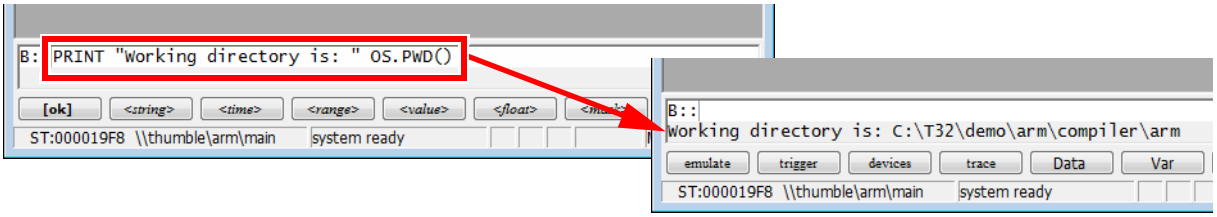
Functions can be used to parametrize commands:

```
; Get the value of register 13, and then dump the memory contents
; starting at this address

Data.dump Register(R13)      ; Command: Data.dump
                             ; Function Register() is used as command
                             ; parameter.
                             ; R13: Register 13 of ARM family chips
                             ; Note: "R" is part of the register name.

; Here, the register name is AL.

Data.dump Register(AL)       ; AL: Lower 8 bit of register AX of the
                             ; x86 family.
```

Example 3: Together with the Output Commands PRINT and Data.Print

You can type functions directly into the TRACE32 command line. In this case it is necessary to use an additional output command like **PRINT** or **Data.Print**, depending on the return value of the function.

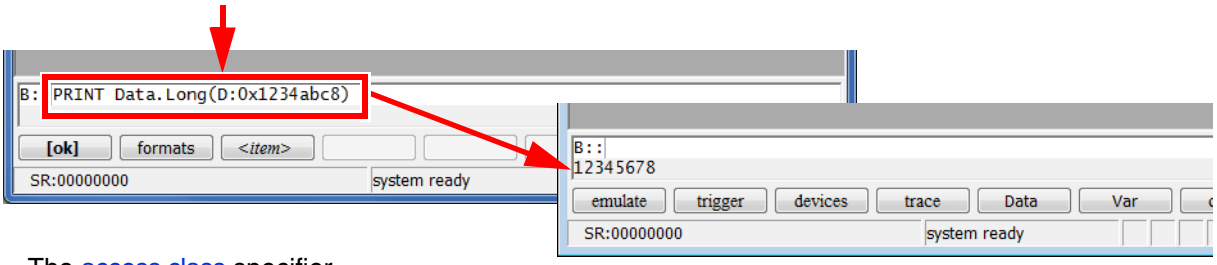
```
; Returns the current working directory.
PRINT "Working directory is: " OS.PresentWorkingDirectory()
```



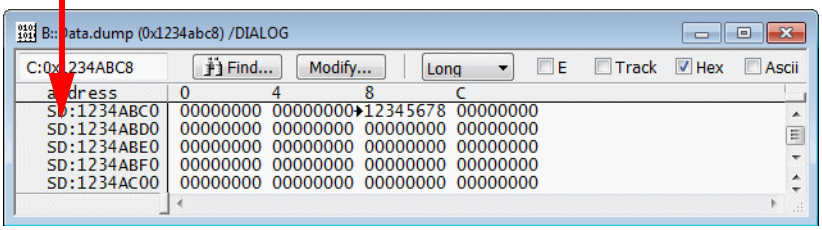
Example 4: Address Function and Their Access Class Specifiers

An address function, such as **Data.Long()**, always requires a parameter together with an access class specifier. In the example below, the access class specifier is `D` for data memory. A plain integer value, such as `0x1234ffff`, without an access class specifier is *not* a valid address.

```
; Returns the memory contents of a long value (32-bit) from the
; data memory (D:)
PRINT Data.Long(D:0x1234abc8)

; 'D:1024.' is a valid, decimal address due to postfix  "."
PRINT Data.Long(D:1024.)

; Fails because the address does not have a access class specifier.
PRINT Data.Long(0x1234ffff)
```



The access class specifier is D (**View** menu > **Dump**)



For lists of target-specific memory class specifiers, see **TRACE32** > **Help** menu > **Processor Architecture Manual**.

Simply search for “Access Classes”.

Which Return Values of Functions can be Printed?

The following return values of functions can be printed:

- Addresses
- ASCII values
- Boolean
- Numerical values:
  - Binary
  - Decimal
  - Float
  - Hex
- Ranges, address ranges, time ranges
- String
- Time values

Related Documents

For training material, refer to:

- **Training PRACTICE**

For more functions, refer to:

- **PowerView Function Reference**
- **Stimuli Generator Function Reference**

In This Section

See also

❏ ACCESS.isGUEST() ❏ ACCESS.isHYPERVISOR()

ACCESS.isGUEST() TRUE if access class belongs to guest

[build 90005 - DVD 02/2018]

Syntax: **ACCESS.isGUEST(***<address>***)**

Returns TRUE if the access class of the specified *<address>* belongs to a guest in the hypervisor environment.

- For ARM, this is the non-secure access class (N: and related ones).
- For x86, this is the guest access class (G: and related ones).
- For PowerPC, this is the guest access class (G: and related ones).

Parameter Type: Address.

Return Value Type: Boolean.

NOTE: If **SYStem.Option.MACHINESPACES** is set to **ON**, then the machine ID of an address is the better criterion to determine whether the address belongs to a guest machine or the hypervisor.
See function **ADDRESS.isGUEST()**.

Examples:

```
; on ARM, guest machines are usually running in non-secure mode, so
; guest addresses usually use access class N: and related.

PRINT ACCESS.isGUEST(N:0xC0000000)           ; TRUE

PRINT ACCESS.isGUEST(H:0xC0000000)           ; FALSE
```

Syntax: **ACCESS.isHYPERVISOR(***<address>***)**

Returns TRUE if the access class belongs to the hypervisor in the hypervisor environment.

- For ARM, this is the hypervisor access class (H: and related ones).
- For x86, this is the host access class (H: and related ones).
- For PowerPC, this is the hypervisor access class (H: and related ones).

Parameter Type: Address.

Return Value Type: Boolean.

NOTE:	If **SYStem.Option.MACHINESPACES** is set to **ON**, then the machine ID of an address is the better criterion to determine whether the address belongs to a guest machine or the hypervisor. See function **ADDRESS.isGUEST()**.

Examples:

```
; on ARM, the host machine which runs the hypervisor is usually running
; in hypervisor mode, so hypervisor addresses use access class H: and
; related
PRINT ACCESS.isHYPERVISOR(H:0xC0000000)      ; TRUE

PRINT ACCESS.isHYPERVISOR(N:0xC0000000)      ; FALSE

PRINT ACCESS.isHYPERVISOR(Z:0xC0000000)      ; FALSE
```

ADDRESS Functions

In This Section

See also

- ADDRESS.ACCESS()
- ADDRESS.ACCESS.CMPSTRICT()
- ADDRESS.INSTR.LEN()
- ADDRESS.isGUEST()
- ADDRESS.isINTERMEDIATE()
- ADDRESS.isNONSECUREEX()
- ADDRESS.isPHYSICAL()
- ADDRESS.isSECURE()
- ADDRESS.MACHINEID()
- ADDRESS.OFFSET()
- ADDRESS.RANGE.END()
- ADDRESS.SEGMENT()
- ADDRESS.STRACCESS()
- ADDRESS.ACCESS.CMP()
- ADDRESS.EXPANDACCESS()
- ADDRESS.isDATA()
- ADDRESS.isHYPERVISOR()
- ADDRESS.isNONSECURE()
- ADDRESS.isONCHIP()
- ADDRESS.isPROGRAM()
- ADDRESS.isSECUREEX()
- ADDRESS.MAU()
- ADDRESS.RANGE.BEGIN()
- ADDRESS.RANGE.SIZE()
- ADDRESS.SPACE()
- ADDRESS.WIDTH()

ADDRESS.ACCESS() Access class as ordinal number

Syntax: **ADDRESS.ACCESS(***<address>***)**
ADDRESS.SPACE(*<address>***)** (deprecated)

Gets the access class as ordinal number from the address.

Parameter Type: Address.

Return Value Type: Hex value.

ADDRESS.ACCESS.CMP() Compare access classes

[build 110619 - DVD 02/2020]

Syntax: **ADDRESS.ACCESS.CMP(***<address1>***,***<address2>***)**

Compares the access classes of *<address1>* and *<address2>* and returns TRUE if they are equal, FALSE otherwise. Missing information in one address will be neglected and the comparison may still return TRUE if the rest of the access class information is equal.

Parameter Type: Address.

Return Value Type: Boolean.

Examples:

```
PRINT ADDRESS.ACCESS.CMP(NP:0x10,NP:0x60)          ; TRUE
PRINT ADDRESS.ACCESS.CMP(N:0x50,NP:0x60)           ; TRUE
PRINT ADDRESS.ACCESS.CMP(ND:0x50,NP:0x50)          ; FALSE
```

ADDRESS.ACCESS.CMPSTRICT() Compare access classes, strict

[build 110619 - DVD 02/2020]

Syntax: **ADDRESS.ACCESS.CMPSTRICT(***<address1>***,***<address2>***)**

Compares the access classes of *<address1>* and *<address2>* and returns TRUE if they are equal, FALSE otherwise. Missing information in one address will yield the comparison to be FALSE.

Parameter Type: Address.

Return Value Type: Boolean.

Examples:

```
PRINT ADDRESS.ACCESS.CMP(NP:0x10,NP:0x60)          ; TRUE
PRINT ADDRESS.ACCESS.CMP(N:0x50,NP:0x60)           ; FALSE
PRINT ADDRESS.ACCESS.CMP(ND:0x50,NP:0x50)          ; FALSE
```

ADDRESS.EXPANDACCESS() Fully qualified access class

[build 75614 - DVD 09/2016]

Syntax: **ADDRESS.EXPANDACCESS(***<address>***)**

Converts an address combining the passed access class and the current CPU status. The returned access class can be referred to as fully qualified.

Parameter Type: Address.

Return Value Type: Address.

In this example, the passed access classes (C and A) and the returned expanded access classes (NSD and ANSD) belong to an ARM Cortex-A9 with TrustZone, which is in the non-secure supervisor state.

```
PRINT ADDRESS.EXPANDACCESS(C:0x0)                  ; returns NSD:0x0
PRINT ADDRESS.EXPANDACCESS(A:0x0)                  ; returns ANSD:0x0
```

ADDRESS.INSTR.LEN() Length of instruction

Syntax: **ADDRESS.INSTR.LEN(**<*address*>**)**

Returns the length in bytes of the instruction at a given address.

Parameter Type: Address.

Return Value Type: Hex value.

ADDRESS.isDATA() Check if memory class refers to data

Syntax: **ADDRESS.isDATA(**<*address*>**)**

Returns TRUE when the memory class of the address is referring to data.

Parameter Type: Address.

Return Value Type: Boolean.

ADDRESS.isGUEST() TRUE if address is guest address

[build 90005 - DVD 02/2018]

Syntax: **ADDRESS.isGUEST(**<*address*>**)**

Returns TRUE if the machine ID of the specified <*address*> belongs to a guest machine in the hypervisor environment. The use of this function requires **SYStem.Option.MACHINESPACES** to be set to **ON**.

Parameter Type: Address. Remember to include the machine ID in the <*address*>.

Return Value Type: Boolean.

Examples:

```
SYStem.Option.MACHINESPACES ON

; if the machine ID of an address is > 0, it is a guest address -
; regardless of the access class.
PRINT ADDRESS.isGUEST(D:0x1:::0xC0000000)          ; TRUE

PRINT ADDRESS.isGUEST(D:0x0:::0xC0000000)          ; FALSE
```

ADDRESS.isHYPERVISOR() TRUE if address is hypervisor address

[build 90005 - DVD 02/2018]

Syntax: **ADDRESS.isHYPERVISOR(**<*address*>**)**

Returns TRUE if the machine ID of the specified <*address*> belongs to the host machine where the hypervisor is running. The use of this function requires **SYStem.Option.MACHINESPACES** to be set to **ON**.

Parameter Type: Address. Remember to include the machine ID in the <*address*>.

Return Value Type: Boolean.

Examples:

```
SYStem.Option.MACHINESPACES ON

; if the machine ID of an address is 0, it is a hypervisor address -
; regardless of the access class.
PRINT ADDRESS.isHYPERVISOR(D:0x0:::0xC0000000)     ; TRUE

PRINT ADDRESS.isHYPERVISOR(D:0x1:::0xC0000000)     ; FALSE
```

ADDRESS.isINTERMEDIATE() Check if intermediate address

Syntax: **ADDRESS.isINTERMEDIATE(**<*address*>**)**

Returns TRUE when the memory class of the address is an intermediate address.

Parameter Type: Address.

Return Value Type: Boolean.

ADDRESS.isNONSECURE() TRUE if non-secure (TrustZone) access

32-bit and 64-bit ARM cores [build 77032 - DVD 02/16]

Syntax: **ADDRESS.isNONSECURE(**<*address*>**)**

Checks if the address passed as parameter will force a non-secure (TrustZone) access.

Parameter Type: Address.

Return Value Type: Boolean.

Example:

```
PRINT ADDRESS.isNONSECURE(AHB:0x0)          ; returns FALSE()
PRINT ADDRESS.isNONSECURE(ZAHB:0x0)         ; returns FALSE()
PRINT ADDRESS.isNONSECURE(NAHB:0x0)         ; returns TRUE()
```

Syntax: **ADDRESS.isNONSECUREEX(**<*address*>**)**

Checks if the address passed as parameter combined with the current CPU status will cause a non-secure (TrustZone) access. This function is a combination of **ADDRESS.isNONSECURE()** and **ADDRESS.EXPANDACCESS()**.

Parameter Type: Address.

Return Value Type: Boolean.

In example 1, the CPU is in non-secure state.

```
Register.Set NS 1                              ; non-secure state
PRINT ADDRESS.isNONSECUREEX(AHB:0x0)           ; returns TRUE()
PRINT ADDRESS.isNONSECUREEX(ZAHB:0x0)          ; returns FALSE()
PRINT ADDRESS.isNONSECUREEX(NAHB:0x0)          ; returns TRUE()
```

In example 2, the CPU is in secure state.

```
Register.Set NS 0                              ; secure state
PRINT ADDRESS.isNONSECUREEX(AHB:0x0)           ; returns FALSE()
PRINT ADDRESS.isNONSECUREEX(ZAHB:0x0)          ; returns FALSE()
PRINT ADDRESS.isNONSECUREEX(NAHB:0x0)          ; returns TRUE()
```

ADDRESS.MACHINEID() Extract machine ID

[build 90005 - DVD 02/2018]

Syntax: **ADDRESS.MACHINEID(**<*address*>**)**

Extracts the machine ID from the specified <*address*>.

Parameter Type: Address.

Return Value Type: Decimal value.

Examples:

```
SYStem.Option.MACHINESPACES ON

ECHO ADDRESS.MACHINEID(D:0x3:::0xC0000000)  ; returns 3.

ECHO ADDRESS.MACHINEID(H:0x0:::0xC0000000)  ; returns 0.
```

[build 86067 - DVD 09/2017]

Syntax: **ADDRESS.MAU(***<address>***)**

ADDRESS.WIDTH(*<address>***)** (deprecated)
[build 20337 - DVD 12/2009]

Data.MAU(*<address>***)** (deprecated)
[build 42354 - DVD 02/2013]

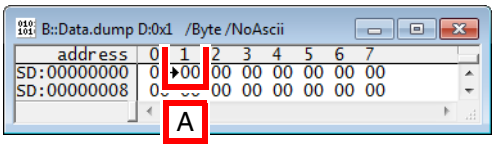
Returns the minimal addressable unit size.

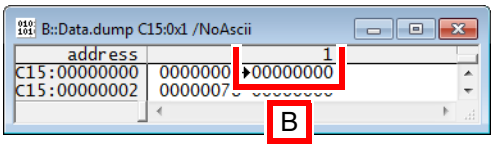
Parameter Type: Address.

Return Value Type: Decimal value.

In this example, the TRACE32 Instruction Set Simulator for ARM is used. The return value 1 means that each address points to 1 byte in memory. The return value 4 means that each address points to 4 bytes in memory.

```
ECHO ADDRESS.MAU(D:0x1)            ;returns 1
Data.dump D:0x1    /Byte /NoAscii

ECHO ADDRESS.MAU(C15:0x1)          ;returns 4
Data.dump C15:0x1 /NoAscii
```





A The address 0x1 of the access class SD: points to 1 byte in memory.

B The address 0x1 of the Coprocessor access class C15: points to 4 bytes in memory.

Syntax: **ADDRESS.OFFSET(***<address>***)**

The *<address>*-object of TRACE32, which is often returned by other functions, always contains the class and the numerical value of the specific memory address. The function **ADDRESS.OFFSET()** returns that numerical value from *<address>*. The class is omitted.

Parameter Type: Address.

Return Value Type: Hex value.

Example:

```
PRINT ADDRESS.OFFSET(SR:0000FF8)              ;returns 0FF8
PRINT ADDRESS.OFFSET(TRACK.ADDRESS())
PRINT ADDRESS.OFFSET(sieve)                   ;returns 2228
```

ADDRESS.isONCHIP() TRUE if on-chip address area

C166

Syntax: **ADDRESS.isONCHIP(***<address>***)**

Returns TRUE when the address refers to on-chip address area.

Parameter Type: Address.

Return Value Type: Boolean.

ADDRESS.isPHYSICAL() TRUE if physical address

Syntax: **ADDRESS.isPHYSICAL(***<address>***)**

Returns TRUE when the memory class of the address is a physical address.
Useful to determine if **TRANS.PHYSICAL()** succeed in translating a logical to a physical address.

Parameter Type: Address.

Return Value Type: Boolean.

ADDRESS.isPROGRAM() TRUE if program address

Syntax: **ADDRESS.isPROGRAM(**<*address*>**)**

Returns TRUE when the memory class of the address is referring to program.

Parameter Type: Address.

Return Value Type: Boolean.

ADDRESS.isSECURE() TRUE if secure (TrustZone) access

32-bit and 64-bit ARM cores [build 77032 - DVD 02/2016]

Syntax: **ADDRESS.isSECURE(**<*address*>**)**

Checks if the address passed as a parameter will force a Secure (TrustZone) access.

Parameter Type: Address.

Return Value Type: Boolean.

Example:

```
PRINT ADDRESS.isSECURE(AHB:0x0)              ; returns FALSE()
PRINT ADDRESS.isSECURE(ZAHB:0x0)             ; returns TRUE()
PRINT ADDRESS.isSECURE(NAHB:0x0)             ; returns FALSE()
```

Syntax: **ADDRESS.isSECUREEX(***<address>***)**

Checks if the address passed as a parameter combined with the current CPU status will cause an Secure (TrustZone) access. Basically this function is a combination of **ADDRESS.isSECURE()** and **ADDRESS.EXPANDACCESS()**.

Parameter Type: Address.

Return Value Type: Boolean.

In example 1, the CPU is in non-secure state.

```
Register.Set NS 1                          ; non-secure state
PRINT ADDRESS.isSECUREEX(AHB:0x0)          ; returns FALSE()
PRINT ADDRESS.isSECUREEX(ZAHB:0x0)         ; returns TRUE()
PRINT ADDRESS.isSECUREEX(NAHB:0x0)         ; returns FALSE()
```

In example 2, the CPU is in secure state.

```
Register.Set NS 0                          ; secure state
PRINT ADDRESS.isSECUREEX(AHB:0x0)          ; returns TRUE()
PRINT ADDRESS.isSECUREEX(ZAHB:0x0)         ; returns TRUE()
PRINT ADDRESS.isSECUREEX(NAHB:0x0)         ; returns FALSE()
```

ADDRESS.RANGE.BEGIN() Lowest address value of address range

[build 66601 - DVD 09/2015]

Syntax: **ADDRESS.RANGE.BEGIN(***<addressrange>***)**

Returns the lowest address value from *<addressrange>*.

Parameter Type: Address range.

Return Value Type: Address.

Example:

```
PRINT ADDRESS.RANGE.BEGIN(P:0x1000--0x2000)           ; returns P:0x1000
PRINT ADDRESS.RANGE.BEGIN(Var.RANGE(flags))
PRINT ADDRESS.RANGE.BEGIN(P:0x20..0x30||P:0x13++0x03) ; returns P:0x13
```

ADDRESS.RANGE.END() Highest address value of address range

[build 66601 - DVD 09/2015]

Syntax: **ADDRESS.RANGE.END(***<addressrange>***)**

Returns the highest address value from *<addressrange>*.

Parameter Type: Address range.

Return Value Type: Address.

Examples:

```
PRINT ADDRESS.RANGE.END(P:0x1000--0x2000)           // returns P:0x2000
PRINT ADDRESS.RANGE.END(Var.RANGE(flags))
PRINT ADDRESS.RANGE.END(P:0x20..0x30||P:0x13++0x03) // returns P:0x30
```

Syntax: **ADDRESS.RANGE.SIZE(***<addressrange>***)**

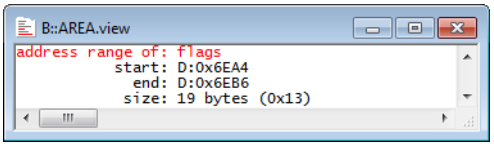
Returns the size of an address range.

Parameter Type: Address range.

Return Value Type: Hex value.

Example: In this script, the start and end address of the HLL variable `flags` is calculated as well as the size of its address range. The output to the **AREA.view** window is formatted with the **PRINTF** command.

```
PRIVATE &rg

&rg="flags"
PRINTF %COLOR.RED "%16s: %s" "address range of"   "&rg"
PRINTF "%16s: %#!A"          "start" ADDRESS.RANGE.BEGIN(Var.RANGE(&rg))
PRINTF "%16s: %#!A"          "end"   ADDRESS.RANGE.END(Var.RANGE(&rg))
PRINTF "%16s: %i bytes "     "size"  ADDRESS.RANGE.SIZE(Var.RANGE(&rg))
PRINTF %CONTinue "(%#x)"             ADDRESS.RANGE.SIZE(Var.RANGE(&rg))

AREA.view
```



ADDRESS.SEGMENT() Segment of an address

[build 22265 - DVD 04/2010]

Syntax: **ADDRESS.SEGMENT(**<*address*>**)**

Returns the segment (space ID) of an address.

Parameter Type: Address.

Return Value Type: Hex value.

Example:

```
PRINT ADDRESS.SEGMENT(D:0x012A:0xC00208A)
```

ADDRESS.STRACCESS() Access class of an address

[build 51173 - DVD 02/2014]

Syntax: **ADDRESS.STRACCESS(**<*address*>**)**

Returns the access class of an address. Useful in combination with **TRANS.PHYSICAL()** to verify that the translation was successful and the returned address is really physical or linear.

Parameter Type: Address.

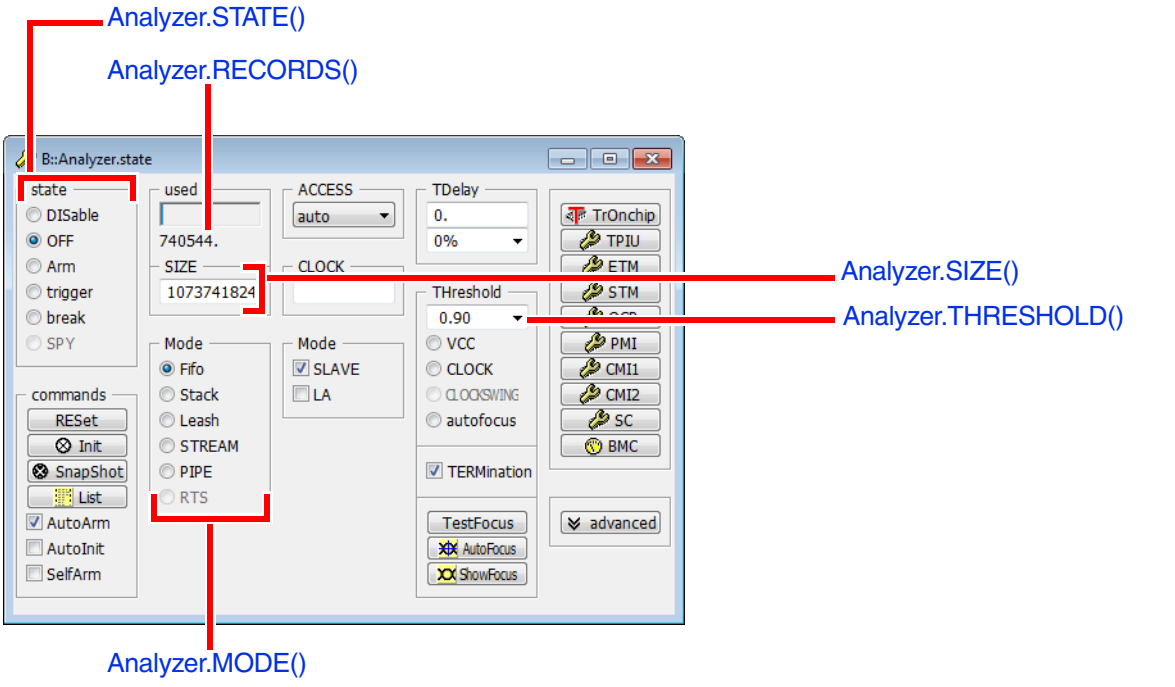
Return Value Type: String.

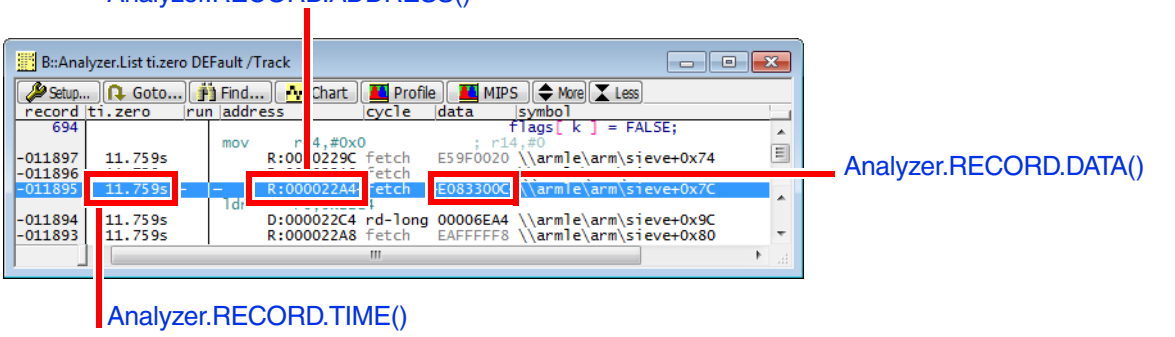
Example:

```
TRANSlation.Create 0--1000 9000
TRANSlation.ON

PRINT TRANS.PHYSICAL(vm:0)                    // returns AVM:9000 (for
                                              // architectures with MMU)

PRINT ADDRESS.OFFSET(TRANS.PHYSICAL(VM:0))    // returns 0x9000

PRINT ADDRESS.STRACCESS(TRANS.PHYSICAL(VM:0)) // returns "AVM:"
```

Analyzer Functions

This figure provides an overview of the return values of some of the Analyzer functions. For descriptions of the illustrated functions and the functions not shown here, see below.





In This Section

See also

- Analyzer()
- Analyzer.CONFIG.POWERTRACE()
- Analyzer.CONFIG.POWERTRACE3()
- Analyzer.CONFIG.POWERTRACESERIAL2()
- Analyzer.COUNTER.TIME()
- Analyzer.FIRST()
- Analyzer.FLOW.FIFOFULL()
- Analyzer.ISCHANNELUP()
- Analyzer.MODE()
- Analyzer.RECORD.ADDRESS()
- Analyzer.RECORD.OFFSET()
- Analyzer.RECORDS()
- Analyzer.SIZE()
- Analyzer.THRESHOLD()
- Analyzer.TRACK.RECORD()
- Analyzer.CONFIG()
- Analyzer.CONFIG.POWERTRACE2()
- Analyzer.CONFIG.POWERTRACESERIAL()
- Analyzer.COUNTER.EVENT()
- Analyzer.DSEL()
- Analyzer.FLOW.ERRORS()
- Analyzer.FOCUS.EYE()
- Analyzer.MAXSIZE()
- Analyzer.PROBEREVISION()
- Analyzer.RECORD.DATA()
- Analyzer.RECORD.TIME()
- Analyzer.REF()
- Analyzer.STATE()
- Analyzer.TraceCONNECT()
- Analyzer.TRIGGER.TIME()

Analyzer() Check if Analyzer command group is available

Syntax: **Analyzer()**
Analyzer.CONFIG() (deprecated)

Returns TRUE in the following cases:

- A TRACE32 trace hardware is connected (not TRACE32 CombiProbe and TRACE32 µTrace (MicroTrace)).
- A TRACE32 Instruction Set Simulator is used.
- A TRACE32 Front-End is used to debug a virtual target that provide trace memory.
- A TRACE32 Front-End is used to debug a RTL simulations / emulations that provides trace capability.

Return Value Type: Boolean.

Analyzer.CONFIG.<powertrace>() Check if specified PowerTrace connected

Syntax:	**Analyzer.CONFIG.***<powertrace>***()**
<powertrace>:	**POWERTRACE** | **POWERTRACE2** | **POWERTRACESERIAL** | **POWERTRACE3**
Full function name only required for HELP.Index.	**Analyzer.CONFIG.POWERTRACE()** **Analyzer.CONFIG.POWERTRACE2()** **Analyzer.CONFIG.POWERTRACESERIAL()** **Analyzer.CONFIG.POWERTRACESERIAL2()** **Analyzer.CONFIG.POWERTRACE3()**

Returns TRUE if the connected TRACE32 tool includes the specified PowerTrace type, FALSE otherwise.

<powertrace>	Description
POWERTRACE	Returns TRUE in case of TRACE32 POWERTRACE / ETHERNET.
POWERTRACE2	Returns TRUE in case of TRACE32 POWERTRACE II.
POWERTRACESERIAL	Returns TRUE in case of TRACE32 POWERTRACE SERIAL.
POWERTRACESERIAL2	Returns TRUE in case of TRACE32 POWERTRACE SERIAL II.
POWERTRACE3	Returns TRUE in case of TRACE32 POWERTRACE III.

Return Value Type: Boolean.

Analyzer.COUNTER.EVENT() Get value of trigger program event counter

NEXUS MPC5XXX MDO8/MDO12/MDO16

Syntax: **Analyzer.COUNTER.EVENT(***<counter_name>***)**

The trigger unit allows to count events by using EVENTCOUNTERs. EVENTCOUNTERs have to be declared in the trigger program by the following command:

EVENTCOUNTER *<counter_name>* [*<event>*]

Returns the current value of *<counter_name>* if the trigger program is loaded.

Parameter Type: String.

Return Value Type: Decimal value.

Analyzer.COUNTER.TIME() Get value of trigger program time counter

Syntax: **Analyzer.COUNTER.TIME(***<counter_name>***)**

The trigger unit allows to count events by using TIMECOUNTERs. TIMECOUNTERs have to be declared in the trigger program by the following command:

TIMECOUNTER *<counter_name>* [*<time>*]

Returns the current value of *<counter_name>* if the trigger program is loaded.

Parameter Type: String.

Return Value Type: Time value.

Analyzer.DSEL() For internal usage only

Syntax: **Analyzer.DSEL()**

Reserved for internal usage.

Return Value Type: String.

Analyzer.FIRST() Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **Analyzer.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

Analyzer.FLOW.ERRORS() Get number of flow errors / hard errors

Syntax: **Analyzer.FLOW.ERRORS()**

Returns the number of flow errors / hard errors found while processing the trace recording.

Return Value Type: Decimal value.

Please be aware that the return value of this function is the accumulated count of events that were encountered while processing the trace recording. All opened windows showing trace data contribute to this value. The value is reset when a new trace recording is made, or when the **Trace.FLOWSTART** or **Trace.FLOWPROCESS** command is executed.

The use of this function is only recommended if you want to find out if a specified part of a trace recording is error free. The part to be analyzed can be defined using **Trace.STATistic.FIRST** and **Trace.STATistic.LAST**. If the defined part is error free (and thus this function returns zero), the analysis results are reliable as well.

Example 1: This script shows how to return only the number of flow errors and hard errors in the trace that is currently visible within the **Analyzer.List** window. If you now scroll up or down in the **Analyzer.List** window or increase the window size, more trace data will be decoded, and thus the number of errors returned by the function may increase.

```
Analyzer.List

PRINT Analyzer.FLOW.ERRORS()

; scroll up or down in the window

PRINT Analyzer.FLOW.ERRORS()
```

Example 2: This script shows how to obtain the exact number of flow errors in the whole trace recording.

```
Trace.Find FLOWERROR /ALL
PRINT FOUND.COUNT()
```

Analyzer.FLOW.FIFOFULL() Get number of FIFO overflows

Syntax: **Analyzer.FLOW.FIFOFULL()**

Returns the number of **FIFO overflows** found while processing the trace recording.

Return Value Type: Decimal value.

Please be aware that the return value of this function is the accumulated count of events that were encountered while processing the trace recording. All opened windows showing trace data contribute to this value. The value is reset when a new trace recording is made, or when the **Trace.FLOWSTART** or **Trace.FLOWPROCESS** command is executed.

The use of this function is only recommended if you want to find out if a specified part of a trace recording is error free. The part to be analyzed can be defined using **Trace.STATistic.FIRST** and **Trace.STATistic.LAST**. If the defined part is error free (and thus this function returns zero), the analysis results are reliable as well.

Example: This script shows how to obtain the exact number of FIFO overflows in the whole trace recording.

```
Analyzer.Find FIFOFULL /ALL
PRINT FOUND.COUNT()
```

Syntax: **Analyzer.FOCUS.EYE(***<channel>***,***<c_time>***,***<c_voltage>***,***<tm>***,***<am>***,***<n>***)**

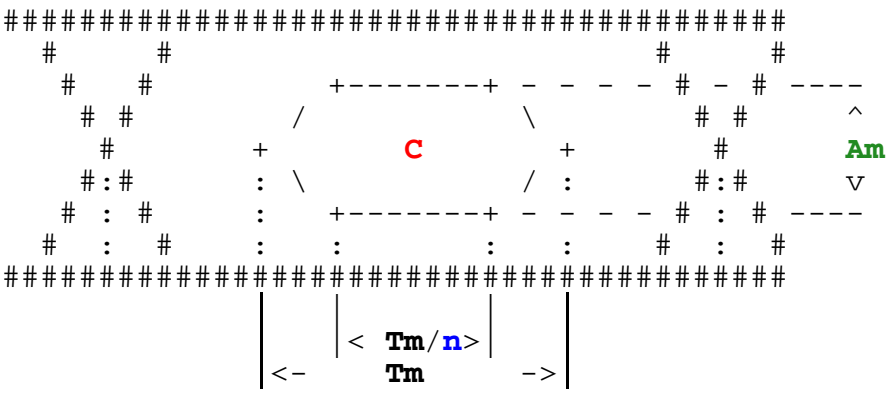
Checks a data eye previously scanned by the **Analyzer.TestFocusEye** command against violations. This function allows you to determine the quality of a data eye via a PRACTICE script, and not just through visual inspection.

Parameter and Description: Use the following parameters to define the region of interest (ROI) that you want to check within the data eye:

<channel>	**Parameter Type**: String. For a list of channel names, refer to the **Analyzer.ShowFocusEye** command. To check all channels, use `"all"`
<c_time>, *<c_voltage>*	**Parameter Type**: Float. Time is given on the x-axis, voltage is given on the y-axis. The intersection of the two values defines the center **C** of the ROI.
<tm>	**Parameter Type**: Float. Time
<am>	**Parameter Type**: Float. Voltage
<n>	**Parameter Type**: Float. Changes the ROI from a rectangle to a hexagon. Rectangle: **n** = 1.0 Hexagon: **n** > 1.0 The greater **n**, the wider the hexagon.

Return Value Type: Hex value.

Example: The function returns 0 if the data eye is clean, and it returns non-zero if there is anything in the data eye. The last tested pattern is shown in the **Analyzer.ShowFocusEye** window.




```
LOCAL &val

;Check the quality of the data eye for channel TS
&val=Analyzer.FOCUS.EYE(TS,0.0,1.65,2.0,0.7,1.0)
IF &val==0
    PRINT "Data eye is OK"

;Check the quality of the data eye for all channels
PRINT Analyzer.FOCUS.EYE("all",3.0,0.9,4.0,0.5,2.0)
```

Analyzer.ISCHANNELUP() Check if serial link is established

PowerTrace Serial, PREPROCESSOR SERIAL

Syntax: **Analyzer.ISCHANNELUP()**

Returns TRUE is the channel training was successful and the serial link is established. An established serial link is the prerequisite for receiving trace data.

In the case of FALSE you have to repeat the channel training.

Return Value Type: Boolean.

Analyzer.MAXSIZE() Get max. size of trace buffer in records

[build 38323 - DVD 08/2012]

Syntax: **Analyzer.MAXSIZE()**

Returns the maximum size of the Analyzer trace buffer in records if a TRACE32 trace hardware is connected.

Returns the current size of the Analyzer trace buffer in records if a TRACE32 Instruction Set Simulator or a TRACE32 Front-End is used.

Return Value Type: Decimal value.

Analyzer.MODE() Get Analyzer recording mode

Syntax: **Analyzer.MODE()**

Returns the current recording mode of the analyzer.

Return Value Type: Decimal value.

Return Value and Description:

0	**Fifo** mode
1	**Stack** mode
2	**Leash** mode
3	**PIPE** mode
4	**RTS** mode (real time profiling)
5	**STREAM** mode (endless trace)

Analyzer.MODE.FLOW() Check if Analyzer operates as flowtrace

[Go to figure]

Syntax: **Analyzer.MODE.FLOW()** (deprecated)

Returns TRUE if the analyzer operates as **FlowTrace**.

Return Value Type: Boolean.

Syntax:	**Analyzer.PCIE.CONFIG("***<register_field>***")**
<register_field>:	**DeviceID** | **VendorID** | **STATus** | **CoMmanD** | **ClassCode** | **REVision** | **HeaderType** | **BaseAddressRegister0** | **BaseAddressRegister1** | **BaseAddressRegister2** | **BaseAddressRegister3** | **BaseAddressRegister4** | **BaseAddressRegister5** | **SubsystemID** | **SubsystemVendorID** | **MaxPayloadSizeSupported** | **MaxPayloadSize** | **MaxLinkSpeed** | **MaxLinkWidth** | **LinkSpeed** | **LinkWidth**

Returns the value of the specified register field from the PCIe configuration space of the analyzer.

Parameter Type: String.

Return Value Type: Hex value.

Analyzer.PCIE.ISCONFIGURED() TRUE if prerequisites are fulfilled

[build 97410 - DVD 09/2018]

Syntax: **Analyzer.PCIE.ISCONFIGURED()**

Returns TRUE if the analyzer configuration is complete, so that the PCIe link can be used for data transfer.

Return Value Type: Boolean.

Example 1: This script checks if the manual PCIe trace setup performed in the previous steps was successful. The previous setup steps are not shown in this script.

```
IF Analyzer.PCIE.ISCONFIGURED()==FALSE()
(
   IF Analyzer.ISCHANNELUP()==FALSE()
      ;PCIe hardware link is down
   ELSE
      ;PCIe link is up, but configuration is not complete
)
```

Example 2: This script instructs TRACE32 to wait until the target operating system has successfully configured the analyzer for PCIe trace.

```
SCREEN.WAIT Analyzer.ISCHANNELUP()
SCREEN.WAIT Analyzer.PCIE.ISCONFIGURED()
SYStem.Mode.Attach
Break
...
```

Analyzer.PCIE.Register() Value of 32-bit register from PCIe configuration

[build 97410 - DVD 09/2018]

Syntax: **Analyzer.PCIE.Register(**<*register_offset*>**)**

Returns the value of the specified 32-bit register from the PCIe configuration space of the analyzer. The register offset is given in 32-bit increments.

Parameter Type: Decimal or hex or binary value. Range: **0.** to **1023.**

Return Value Type: Hex value.

Analyzer.PROBEREVISION() Get revision of StarCore NEXUS probe

Syntax: **Analyzer.PROBEREVISION()**

Returns the revision number of the NEXUS probe.The StarCore NEXUS probe is out of production since 2010.

Return Value Type: Decimal value.

Analyzer.RECORDS() Get number of used trace records

[Go to figure]

Syntax: **Analyzer.RECORDS()**

Returns the number of records in the Analyzer.

If the state is **OFF** and the current mode is **STREAM**, this function will block until all buffered data has been received.

Return Value Type: Decimal value.

Analyzer.RECORD.ADDRESS() Get address recorded in trace record

[Go to figure]

Syntax: **Analyzer.RECORD.ADDRESS(***<record_number>***)**

Returns the address from the specified record of the Analyzer trace. If the specified record does not contain an address C:0x0 is returned.

Parameter Type: Decimal value.

Return Value Type: Address.

Example 1:

```
PRINT Analyzer.RECORD.ADDRESS(-9.)      ;return value example: R:0x226C
```

Example 2: This example shows how to return the address recorded in the reference record. The address is then used to display the memory contents of the trace reference record in a **Data.dump** window.

```
Analyzer.List DEFault /Track ;open an Analyzer.List window
Analyzer.REF -11895.         ;set this trace record as reference record
Analyzer.GOTO Analyzer.REF() ;go to the specified trace reference record

;get the address of the trace reference record, and then display the
;memory contents for this trace reference record in a Data.dump window
Data.dump Analyzer.RECORD.ADDRESS(Analyzer.REF()) /Track
```

In the **Analyzer.List** window, right-click another trace record, and then select **Set Ref**.
After you have set a new trace reference record, the **Data.dump** window updates accordingly because of the **Track** option.

Analyzer.RECORD.DATA() Get data recorded in trace record

[Go to figure]

Syntax: **Analyzer.RECORD.DATA(***<record_number>***)**

Returns the data from the specified record of the Analyzer trace. If the specified record does not contain data the function returns 0FFFFFFFFFFFFFFF.

Parameter Type: Decimal value.

Return Value Type: Hex value.

Analyzer.RECORD.OFFSET() Get address in trace record as number

Syntax: **Analyzer.RECORD.OFFSET(***<record_number>***)**

Returns the number within the address from the specified record of the Analyzer trace. If the specified record does not contain an address 0 is returned.

```
PRINT Analyzer.RECORD.ADDRESS(-100.)       ; prints NR:0xFFFF:0xC0014A08

PRINT Analyzer.RECORD.OFFSET(-100.)        ; prints 0xC0014A08
```

Parameter Type: Decimal value.

Return Value Type: Hex value.

Syntax: **Analyzer.RECORD.TIME(**<*record_number*>**)**

Returns the timestamp of the specified record in fixed length format. The timestamp is relative to the trace zero point. For details refer to **Trace.ZERO**.

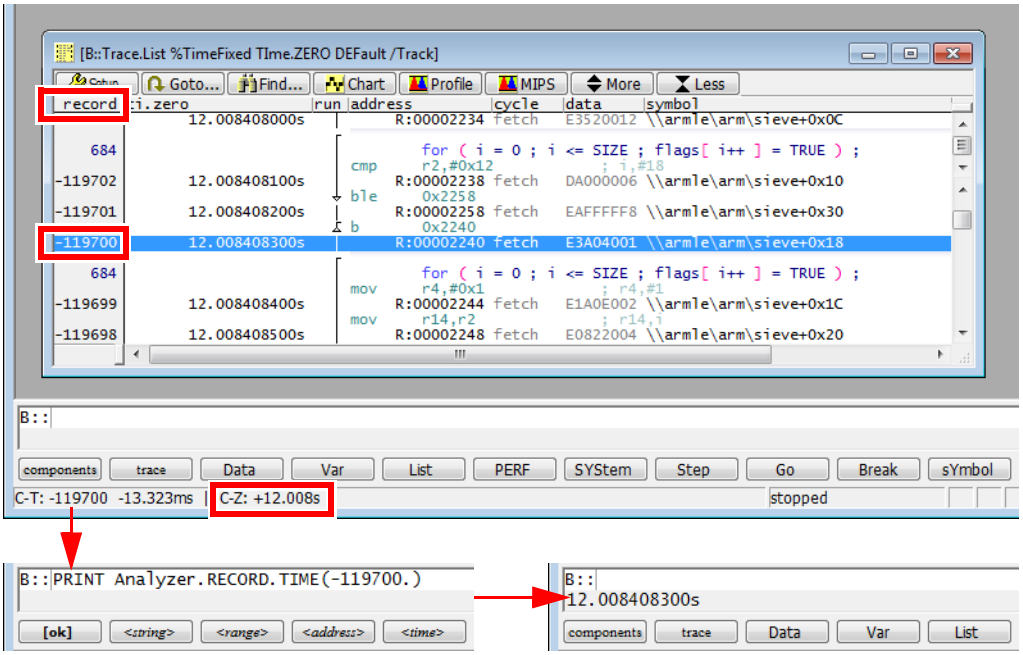
If the record does not provide timing information (dummy record), the timestamp of the preceding record is returned.

Parameter Type: Decimal value.

Return Value Type: Time value.

Example:

```
Trace.List %TimeFixed TIme.ZERO DEFault /Track

PRINT Analyzer.RECORD.TIME(-119700.)
```



Analyzer.REF() Get record number of reference record

Syntax: **Analyzer.REF()**

The command **Analyzer.REF** allows to mark a trace record as reference record. The function returns the record number of the reference record.

Return Value Type: Decimal value.

Analyzer.SIZE() Get current trace buffer size in records

[Go to figure]

Syntax: **Analyzer.SIZE()**

Returns the current size of the trace buffer in records.

Return Value Type: Decimal value.

Analyzer.STATE() Get state of Analyzer

[Go to figure]

Syntax: **Analyzer.STATE()**

Returns the current state of the Analyzer. If the state is **OFF** and the current mode is **STREAM**, this function will block until all buffered data has been received.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state
5	Analyzer hardware not available.
9	**OFF** state, but data is still being processed (**PIPE** and **RTS** modes only)

Analyzer.THRESHOLD() Get threshold voltage of parallel preprocessor

[Go to figure]

Syntax: **Analyzer.THRESHOLD()**

Returns the threshold voltage of the parallel preprocessor.

Return Value Type: Float.

Analyzer.TraceCONNECT() Name of trace sink of the SoC

[build 80222 - DVD Feb/2017]

Syntax: **Analyzer.TraceCONNECT()**

Returns the name of the currently selected trace sink of the SoC. In case no trace-sink is selected/available, the function returns **NONE**. The trace sink is selected with the **<trace>.TraceCONNECT** command.

Return Value Type: String.

Example: See **Onchip.TraceCONNECT()**.

Analyzer.TRACK.RECORD() Get record number matching search

[Go to figure]

Syntax: **Analyzer.TRACK.RECORD()**

After a successful search operation, this function returns the record number.

Return Value Type: Decimal value.

Example: This example shows how to search for and return the record numbers of the first, second, and all records of the symbol **sieve**.

```
; Finds the first occurrence of the symbol sieve
Trace.Find , Address sieve
; Returns the record number of the first occurrence
IF FOUND()
    PRINT Analyzer.TRACK.RECORD()


; Search for the next occurrence of the symbol sieve
Trace.Find ; Running the command without any arguments
           ; repeats the previous search
; Returns the record number of the next occurrence
IF FOUND()
    PRINT Analyzer.TRACK.RECORD()
```

Analyzer.TRIGGER.TIME() Time of trigger point in trace

[build 78297 - DVD 02/2017]

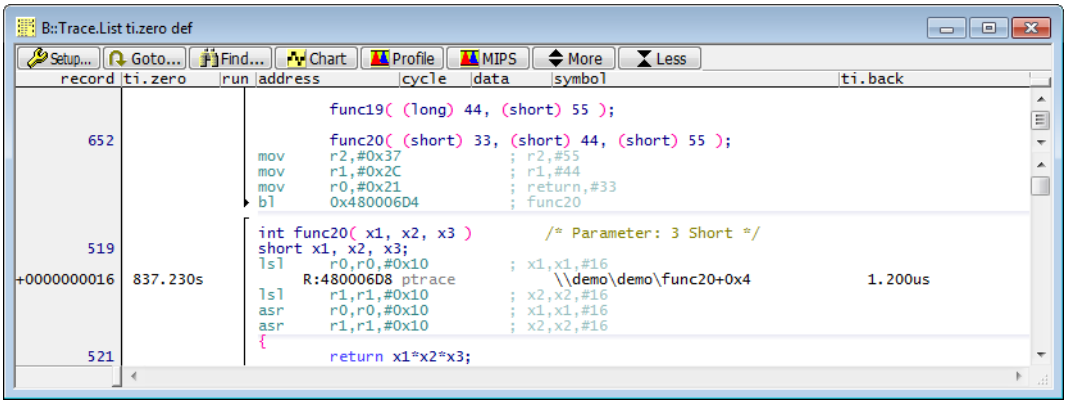
Syntax: **Analyzer.TRIGGER.TIME()**

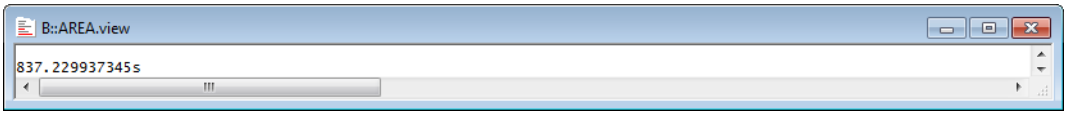
Returns the time of the trigger point in the trace.

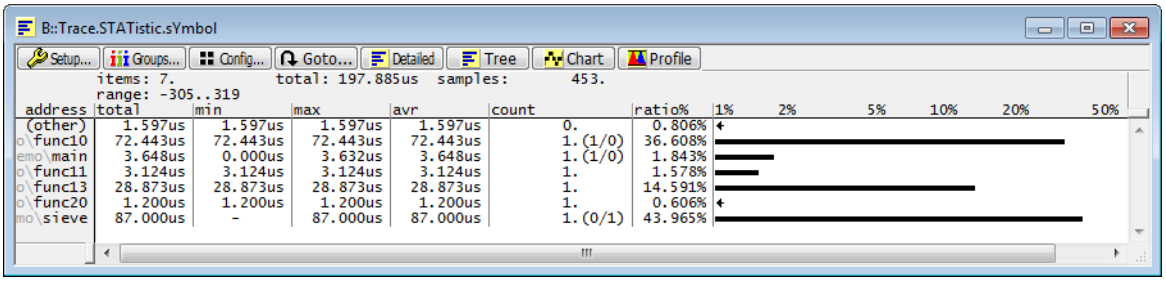
Return Value Type: Time value.

Example:

```
;define trace trigger point at function entry of func20()
Break.Set func20 /TraceTrigger

;and trace a little bit longer
 Trace.TDelay 1%

;run program for trace recording
  ...

;show statistics +/-100 us around the trigger point at func20()
PRINT Analyzer.TRIGGER.TIME()
Trace.STATistic.FIRST Analyzer.TRIGGER.TIME()-100.us
Trace.STATistic.LAST  Analyzer.TRIGGER.TIME()+100.us
Trace.STATistic.sYmbol
```







ARMARCHVERSION() ARM architecture version of CPU

ARM debuggers [build 57602 - DVD 02/2015]

Syntax: **ARMARCHVERSION(**["*<parameter>*"]**)**

<parameter>: **minor**
major

Returns the ARM architecture version of the selected CPU.

The parameter is optional (see table below).

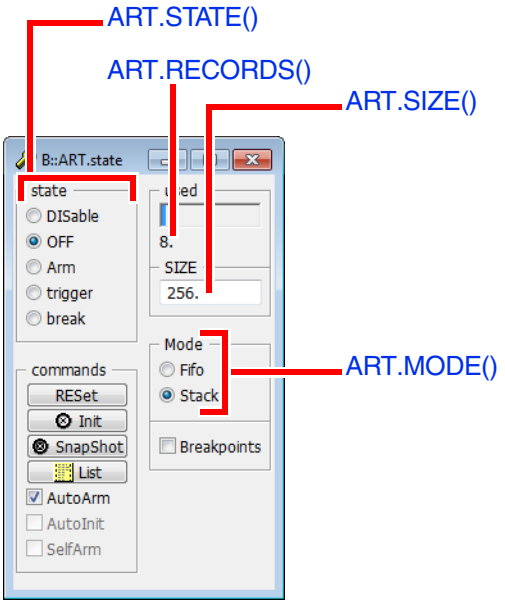
Return Value Type: Decimal value.

Return Value and Description:

Parameter	Return value	Description
None	0x0	ARMv4 / ARM7
	0x1	ARMv4 / ARM9
	0x5	ARMv5
	0x6	ARMv6
	0x7	ARMv7
	0x8	ARMv8
	0x8<x>	ARMv8.x, e.g. 0x81 for Armv8.1
	0x9	ARMv9
minor	Returns minor version number. E.g. `ARMARCHVERSION("minor")` returns "0x1" for Armv8.1.	
major	Returns major version number. E.g. `ARMARCHVERSION("major")` returns "0x8" for Armv8.1.	

Advanced Register Trace (ART) Functions

This figure provides an overview of the return values of some of the ART functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- ART.FIRST()
- ART.MAXSIZE()
- ART.MODE()
- ART.RECORD.ADDRESS()
- ART.RECORD.OFFSET()
- ART.RECORD.TIME()
- ART.RECORDS()
- ART.REF()
- ART.SIZE()
- ART.STATE()
- ART.TRACK.RECORD()

ART.FIRST() Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **ART.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

ART.MAXSIZE() Get max. size of trace buffer in records

[build 38323 - DVD 08/2012]

Syntax: **ART.MAXSIZE()**

Returns the maximum possible size of the ART trace buffer in records.

Return Value Type: Decimal value.

ART.MODE() Get ART recording mode

[Go to figure]

Syntax: **ART.MODE()**

Returns the current recording mode of the ART.

Return Value Type: Decimal value.

Return Value and Description:

0	**Fifo** mode
1	**Stack** mode

ART.RECORD.ADDRESS() Get address recorded in trace record

[build 38764]

Syntax: **ART.RECORD.ADDRESS(***<record_number>***)**

Returns the sampled address (access class and offset) from the specified record.

Parameter Type: Decimal value.

Return Value Type: Address.

ART.RECORD.OFFSET() Get address in trace record as number

[build 38764]

Syntax: **ART.RECORD.OFFSET(**<*record_number*>**)**

Returns the address-offset of the sampled address from the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

ART.RECORD.TIME() Get timestamp of trace record

[build 38764]

Syntax: **ART.RECORD.TIME(**<*record_number*>**)**

Returns the timestamp of the specified record. For an example, see **Analyzer.RECORD.TIME()**.

Parameter Type: Decimal value.

Return Value Type: Time value.

ART.RECORDS() Get number of used trace records

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **ART.RECORDS()**

Returns the number of records stored in the trace buffer.

Return Value Type: Decimal value.

ART.REF() Get record number of reference record

[build 38323 - DVD 08/2012]

Syntax: **ART.REF()**

The number of the selected reference record in the analyzer trace.

Return Value Type: Decimal value.

ART.SIZE() Get current trace buffer size in records

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **ART.SIZE()**

Returns the size of the trace buffer.

Return Value Type: Decimal value.

ART.STATE() Get state of ART trace

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **ART.STATE()**

Returns the state of the ART trace.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state

ART.TRACK.RECORD() Get record number matching search

[build 38323 - DVD 08/2012]

Syntax: **ART.TRACK.RECORD()**

After a successful search operation, this function returns the record number. For an example, see **Analyzer.TRACK.RECORD()**.

Return Value Type: Decimal value.

AUTOFOCUS Functions

In This Section

See also

- ❏ AUTOFOCUS()
- ❏ AUTOFOCUS.OK()
- ❏ AUTOFOCUS.FREQUENCY()

AUTOFOCUS() TRUE if AutoFocus preprocessor attached

[build 07808 - DVD 09/2007]

Syntax: **AUTOFOCUS()**

Returns TRUE if an AutoFocus preprocessor is attached.

Return Value Type: Boolean.

AUTOFOCUS.OK() TRUE if command execution successful

[build 23129, DVD 11/2010]

Syntax: **AUTOFOCUS.OK()**

Returns TRUE if the last execution of the command **Analyzer.AutoFocus** or **Analyzer.TestFocus** was successful.

Return Value Type: Boolean.

AUTOFOCUS.FREQUENCY() Frequency of trace-port clock

[build 23129, DVD 11/2010]

Syntax: **AUTOFOCUS.FREQUENCY()**

Returns the frequency of the trace-port clock detected by the last execution of the command **Analyzer.AutoFocus** or **Analyzer.TestFocus** was successful.

Return Value Type: Decimal value.

AVX Functions

In This Section

See also

- AVX
- AVX512
- AVX()
- AVX512()

AVX() — Content of AVX register

Syntax: **AVX(**<*register_name*>**.**<*column_number*>**)**

Returns a 32-bit segment of the selected 256-bit AVX register. See also **AVX** command group.

Parameter Type: String.

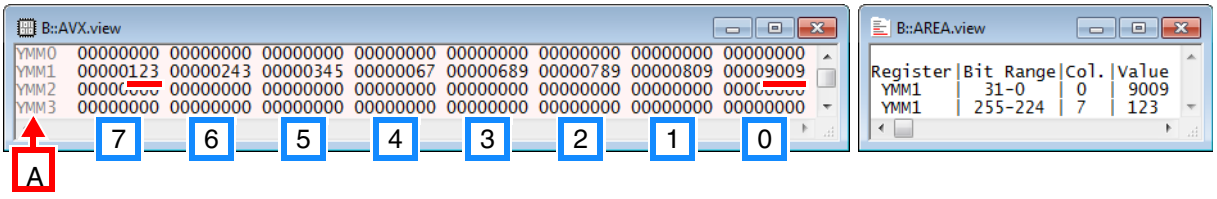
Parameter and Description:

<*register_name*>	The register names are listed in the **AVX.view** window.
<*column_number*>	The column numbers start at 0 and are read from right to left in the **AVX.view** window. See example below. A column `k` corresponds to the bit range `(k · 32 + 31)-(k · 32)`, where `0<=k<=7`

Return Value Type: Hex value.

Example: This demo script returns 32-bit values of the register **YMM1** from the column **0** (bits 31 to 0) and the column **7** (bits 255 to 224) of the **AVX.view** window.

```
AVX.view
AVX.Set YMM1 123  243  345  67  689  789  809  9009
PRINT "Register|Bit Range|Col.|Value"
PRINT " YMM1    |  31-0    | 0   | " AVX(YMM1.0) ;32-bit value from col. 0
PRINT " YMM1    | 255-224  | 7   | " AVX(YMM1.7) ;32-bit value from col. 7
```



A Register names. **0 - 7** Column numbers - from right to left - in the **AVX.view** window.

Syntax: **AVX512(***<register_name>***.***<column_number>***)**

Returns a 32-bit segment of the selected 512-bit AVX512 register. See also **AVX512** command group.

Parameter Type: String.

Parameter and Description:

<register_name>	The register names are listed in the **AVX512.view** window.
<column_number>	The column numbers start at 0 and are read from right to left in the **AVX512.view** window. For an example, refer to the related function **AVX()**. A column `k` corresponds to the bit range `(k · 32 + 31)-(k · 32)`, where `0<=k<=15`

Return Value Type: Hex value.

Break Functions

In This Section

See also

- Break.Alpha.EXIST()
- Break.Beta.EXIST()
- Break.Charly.EXIST()
- Break.Program.EXIST()
- Break.ReadWrite.EXIST()

Break.Alpha.EXIST() TRUE if Alpha breakpoint exists

[build 75340 - DVD 09/2016]

Syntax: **Break.Alpha.EXIST(**<*address*>**)**

Returns TRUE if an Alpha breakpoint exists.

Parameter Type: Address.

Return Value Type: Boolean.

Example:

```
List.Mix                          ;optional step: display a listing

Break.Set func2 /Alpha            ;set an Alpha breakpoint at the
                                  ;symbol 'func2'

PRINT Break.Alpha.EXIST(func2)    ;returns TRUE, i.e. the Alpha breakpoint
                                  ;exists at the symbol 'func2'
```

Break.Beta.EXIST() TRUE if Beta breakpoint exist

[build 75340 - DVD 09/2016]

Syntax: **Break.Beta.EXIST(**<*address*>**)**

Returns TRUE if a Beta breakpoint exists.

Parameter Type: Address.

Return Value Type: Boolean.

Break.Charly.EXIST() TRUE if Charly breakpoint exists

[build 75340 - DVD 09/2016]

Syntax: **Break.Charly.EXIST(***<address>***)**

Returns TRUE if a Charly breakpoint exists.

Parameter Type: Address.

Return Value Type: Boolean.

Break.Program.EXIST() TRUE if enabled program breakpoint exists

[build 85294 - DVD 09/2017]

Syntax: **Break.Program.EXIST(***<address>***)**

Returns TRUE if an enabled program breakpoint exists at the given address location.

Parameter Type: Address.

Return Value Type: Boolean.

Break.ReadWrite.EXIST() TRUE if enabled data address breakpoint exists

[build 85294 - DVD 09/2017]

Syntax: **Break.ReadWrite.EXIST(***<address>***)**

Returns TRUE if an enabled data address breakpoint (read, write, or read-write) exists at the given address location.

Parameter Type: Address.

Return Value Type: Boolean.

BMC Functions (Benchmark Counter)

In This Section

See also

- BMC
- BMC.CLOCK()
- BMC.COUNTER()
- BMC.COUNTER.BYNAME()
- BMC.COUNTER.BYNAME.CORE()
- BMC.COUNTER.CORE()
- BMC.OVERFLOW()
- BMC.OVERFLOW.BYNAME()
- BMC.OVERFLOW.BYNAME.CORE()
- BMC.OVERFLOW.CORE()

BMC.CLOCK() Frequency of core clock

[build 72352 - DVD 09/2016]

Syntax: **BMC.CLOCK()**

Returns the frequency set with the **BMC.CLOCK** command.

Return Value Type: Decimal value.

BMC.COUNTER() Value of a benchmark counter

Syntax: **BMC.COUNTER(***<counter_index>***)**

Returns the counter value of the benchmark counter with the specified index. See also **BMC** command group. In a multicore environment, the **BMC.COUNTER()** function returns the accumulated value of all cores.

Parameter Type: Decimal value.

Return Value Type: Hex value.

Example: For PMN3 use BMC.COUNTER(3).

```
PRINT BMC.COUNTER(3) ;prints the result to the TRACE32 message line
```

BMC.COUNTER.BYNAME() Value of a benchmark counter

Syntax: **BMC.COUNTER.BYNAME("***<counter_name>***")**

Returns the counter value of the benchmark counter with the specified name. See also **BMC** command group. In a multicore environment, the **BMC.COUNTER.BYNAME()** function returns the accumulated value of all cores.

Parameter Type: String.

Return Value Type: Hex value.

Example: For ETM1 use BMC.COUNTER.BYNAME("ETM1").

```
;prints the result to the TRACE32 message line
PRINT BMC.COUNTER.BYNAME("ETM1")
```

BMC.COUNTER.CORE() Value of a benchmark counter

Syntax: **BMC.COUNTER.CORE(***<counter_index>***, *<core_index>***)**

Returns the counter value of the benchmark counter of the specified core and index. See also **BMC** command group.

Parameter and Description:

<counter_index>	**Parameter Type**: Decimal value.
<core_index>	**Parameter Type**: Decimal value.

Return Value Type: Hex value.

Example: For PMN**3** of core #1 use BMC.COUNTER.CORE(**3, 1**).

```
;prints the result to the TRACE32 message line
PRINT BMC.COUNTER.CORE(3, 1)
```

BMC.COUNTER.BYNAME.CORE() Value of a benchmark counter

Syntax: **BMC.COUNTER.BYNAME.CORE("***<counter_name>***", *<core_index>*)**

Returns the counter value of the benchmark counter with the specified name. See also **BMC** command group. In a multicore environment, the **BMC.COUNTER.BYNAME.CORE()** function returns the accumulated value of all cores.

Parameter Type: String.

Return Value Type: Hex value.

Example: For PMN0 of core #1 use BMC.COUNTER.CORE("PMN0", **1**).

```
;prints the result to the TRACE32 message line
PRINT BMC.COUNTER.BYNAME.CORE("PMN0", 1)
```

BMC.OVERFLOW() TRUE if benchmark counter overflow

[build 72015 - DVD 08/2016]

Syntax: **BMC.OVERFLOW(***<counter_index>***)**

Returns TRUE if the benchmark counter has overflown. In a multicore environment, the **BMC.OVERFLOW** function returns the OR'ed overflow status of all cores.

Parameter Type: Decimal value.

Return Value Type: Boolean.

BMC.OVERFLOW.BYNAME() TRUE if benchmark counter overflow

[build 72015 - DVD 08/2016]

Syntax: **BMC.OVERFLOW.BYNAME(***<counter_name>***)**

Returns TRUE if the benchmark counter has overflown. In a multicore environment, the **BMC.OVERFLOW.BYNAME()** function returns the OR'ed overflow status of all cores.

Parameter Type: String.

Return Value Type: Boolean.

BMC.OVERFLOW.CORE() TRUE if benchmark counter overflow

[build 72015 - DVD 08/2016]

Syntax: **BMC.OVERFLOW.CORE(***<counter_index>***, ***<core_index>***)**

Returns TRUE if the benchmark counter of the specified core has overflown.

Parameter and Description:

<counter_index>	**Parameter Type**: Decimal value.
<core_index>	**Parameter Type**: Decimal value.

Return Value Type: Boolean.

BMC.OVERFLOW.BYNAME.CORE() TRUE if benchmark counter overflow

[build 72015 - DVD 08/2016]

Syntax: **BMC.OVERFLOW.BYNAME.CORE("***<counter_name>***", ***<core_index>***)**

Returns TRUE if the benchmark counter of the specified core has overflown.

Parameter and Description:

<counter_name>	**Parameter Type**: String.
<core_index>	**Parameter Type**: Decimal value.

Return Value Type: Boolean.

Boundary Scan Description Language (BSDL) Functions

In This Section

See also

- BSDL
- BSDL.CHECK.FLASHCONF()
- BSDL.GetDRBit()
- BSDL.CHECK.BYPASS()
- BSDL.CHECK.IDCODE()
- BSDL.GetPortLevel()

BSDL.CHECK.BYPASS() Chain bypass test

[build 28569 - DVD 06/2011]

Syntax: **BSDL.CHECK.BYPASS()**

Executes a boundary scan BYPASS test and returns TRUE if it passes.

Return Value Type: Boolean.

BSDL.CHECK.FLASHCONF() Flash configuration test

[build 28569 - DVD 06/2011]

Syntax: **BSDL.CHECK.FLASHCONF()**

Checks the FLASH definition and the pin mapping for the boundary scan interface; returns TRUE, when FLASH configuration has no errors.

Return Value Type: Boolean.

BSDL.CHECK.IDCODE() Chain IDCODE test

[build 28569 - DVD 06/2011]

Syntax: **BSDL.CHECK.IDCODE()**

Executes a boundary scan IDCODE test and returns TRUE if it passes.

Return Value Type: Boolean.

BSDL.GetDRBit() Data register bit

[build 35348 - DVD 08/2012]

Syntax: **BSDL.GetDRBit(***<chip_number>***,***<bit_number>***)**

Returns the last read value of bit *<bit_number>* from chip *<chip_number>*. If the requested bit was not read before, -1 is returned.

Parameter and Description:

<chip_number>	**Parameter Type**: Decimal or hex or binary value. Number of IC in the boundary scan chain. Range: **1.** to *<n>***.**
<bit_number>	**Parameter Type**: Decimal or hex or binary value. Bit index of the current data register. Range: **0.** to *<i>***.**

Return Value Type: Decimal value.

BSDL.GetPortLevel() Port level value

[build 36947 - DVD 08/2012]

Syntax: **BSDL.GetPortLevel(***<chip_number>***,"***<port_name>***")**

Returns the port level of *<port_name>* from chip *<chip_number>* from the last boundary scan capture.

Parameter and Description:

<chip_number>	**Parameter Type**: Decimal or hex or binary value. Number of IC in the boundary scan chain. Range: **1.** to *<n>***.**
<port_name>	**Parameter Type**: String. Valid port name from the BSDL file for the specified *<chip_number>*.

Return Value Type: Decimal value.

Return Value and Description:

0	Low signal level
1	High signal level
2	Unknown level (e.g. floating output port)
-1	Error

CABLE Functions

In This Section

See also

- CABLE.GalvanicISOlation()
- CABLE.GalvanicISOlation.SERIAL()
- CABLE.SERIAL()
- CABLE.GalvanicISOlation.FIRMWARE()
- CABLE.NAME()
- CABLE.TWOWIRE()

CABLE.GalvanicISOlation() Cable has galvanic isolation

[build 81072 - DVD 02/2017]

Syntax: **CABLE.GalvanicISOlation()**

Returns TRUE if the cable has galvanic isolation.

Return Value Type: Boolean.

CABLE.GalvanicISOlation.FIRMWARE() Adapter firmware version

[build 81097 - DVD 02/2017]

Syntax: **CABLE.GalvanicISOlation.FIRMWARE()**

Return the firmware version of the galvanic isolation adapter as a string. The same string is also shown in the **VERSION.HARDWARE** window. An empty string is returned if no adapter is plugged.

Return Value Type: String.

CABLE.GalvanicISOlation.SERIAL() Serial number of adapter

[build 81072 - DVD 02/2017]

Syntax: **CABLE.GalvanicISOlation.SERIAL()**

Returns the serial number of the galvanic isolation adapter. The same serial number is also shown in the **VERSION.HARDWARE** window. An empty string is returned if no adapter is plugged.

Return Value Type: String.

CABLE.NAME() Name of debug cable

[build 21341 - DVD 04/2010]

Syntax: **CABLE.NAME()**

Returns the debug cable name. The same cable name is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

Example:

```
PRINT CABLE.NAME()     ; Returns the cable name, e.g.:
                       ; - OCDS Uni-Dir Debug Cable V0
                       ; - ARM Debug Cable V4b
```

CABLE.SERIAL() Serial number of debug cable

[build 21341 - DVD 04/2010]

Syntax: **CABLE.SERIAL()**

Returns the first serial number of the plugged debug cable. It is the same serial number that is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

CABLE.TWOWIRE() TRUE if two-wire debugging supported

[build 21341 - DVD 04/2010]

Syntax: **CABLE.TWOWIRE()**

Returns TRUE if the used debug hardware supports two-wire debugging like cJTAG or SWD.

Return Value Type: Boolean.

Example:

```
;Check if the cpu family is ARM. If not, the block is skipped.
IF CPUFAMILY()=="ARM"
(
      IF CABLE.TWOWIRE()  ;DebugCable supports Serial Wire Debug (SWD)?
           SYStem.CONFIG SWD
)
```

CACHE Functions

In This Section

The CACHE functions give access to the status data of all cache lines, e.g. Valid, Dirty tag address and others. Cache lines are addressed by their Set and Way indices.

If the target processor implements an L1 Unified Cache (used for both instruction and data), use the **CACHE.IC** functions to access the status information.

See also

- CACHE
- CACHE.DC.DIRTY()
- CACHE.DC.DIRTYMASK()
- CACHE.DC.LRU()
- CACHE.DC.TAG()
- CACHE.DC.VALID()
- CACHE.DC.VALIDMASK()
- CACHE.IC.DIRTY()
- CACHE.IC.DIRTYMASK()
- CACHE.IC.LRU()
- CACHE.IC.TAG()
- CACHE.IC.VALID()
- CACHE.IC.VALIDMASK()
- CACHE.L2.DIRTY()
- CACHE.L2.DIRTYMASK()
- CACHE.L2.LRU()
- CACHE.L2.SHARED()
- CACHE.L2.SHAREDMASK()
- CACHE.L2.TAG()
- CACHE.L2.VALID()
- CACHE.L2.VALIDMASK()
- CACHE.L3.DIRTY()
- CACHE.L3.DIRTYMASK()
- CACHE.L3.LRU()
- CACHE.L3.TAG()
- CACHE.L3.VALID()
- CACHE.L3.VALIDMASK()

CACHE.DC.DIRTY() Dirty-flag of L1 Data Cache Line

Syntax: **CACHE.DC.DIRTY(***<set>***,***<way>***)**

Returns TRUE if the specified set/way of the data cache is DIRTY.

On processors with unified L1 cache, use **CACHE.IC.DIRTY()**.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.DC.DIRTYMASK() Dirty-flag mask of L1 Data Cache Line

Syntax: **CACHE.DC.DIRTYMASK(***<set>***,***<way>***)**

Returns a value containing all DIRTY flags of the specified set/way of the data cache. Use for caches with multiple DIRTY flags/sectors per cache line.

On processors with unified L1 cache, use **CACHE.IC.DIRTYMASK()**.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.DC.LRU() LRU information of L1 Data Cache Line

[build 52317]

Syntax: **CACHE.DC.LRU(***<set>***)**

Returns the index of the least recently used (next to be replaced) way of the specified set. Only a few processors provide this information.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.DC.TAG() Address Tag of L1 Data Cache Line

Syntax: **CACHE.DC.TAG(***<set>***,***<way>***)**

Returns the address tag of the specified set/way of the data cache.

On processors with unified L1 cache, use **CACHE.IC.TAG()**.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.DC.VALID() Valid-flag of L1 Data Cache Line

Syntax: **CACHE.DC.VALID(***<set>***,***<way>***)**

Returns TRUE if the specified set/way of the data cache is VALID.

On processors with unified L1 cache, use **CACHE.IC.VALID()**.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.DC.VALIDMASK() Valid-flag mask of L1 Data Cache Line

Syntax: **CACHE.DC.VALIDMASK(***<set>***,***<way>***)**

Returns a value containing all VALID flags of the specified set/way of the data cache. Use for caches with multiple valid flags/sectors per cache line.

Use **CACHE.IC.VALIDMASK()** for systems with unified L1 cache.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.IC.DIRTY() Dirty-flag of L1 Unified Cache Line

Syntax: **CACHE.IC.DIRTY(**<*set*>**,**<*way*>**)**

Returns TRUE if the specified set/way of the unified L1 cache is DIRTY. For instruction caches, the result is always FALSE.

Parameter and Description:

<*set*>	**Parameter Type**: Decimal or hex or binary value.
<*way*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.IC.DIRTYMASK() Dirty-flag mask of L1 Unified Cache Line

Syntax: **CACHE.IC.DIRTYMASK(**<*set*>**,**<*way*>**)**

For processors with unified L1 cache. Returns a value containing all DIRTY flags of the specified set/way of the cache line, and zero for pure instruction caches. Use for caches with multiple dirty flags/sectors per cache line.

Parameter and Description:

<*set*>	**Parameter Type**: Decimal or hex or binary value.
<*way*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.IC.LRU() LRU information of L1 Instruction Cache Line

[build 52317]

Syntax: **CACHE.IC.LRU(**<*set*>**)**

Returns the index of the least recently used (next to be replaced) way of the specified set. Only a few processors provide this information.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.IC.TAG() Address Tag of L1 Instruction Cache Line

Syntax: **CACHE.IC.TAG(***<set>***,***<way>***)**

Returns the address tag of the specified set/way of the instruction (or unified) cache.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.IC.VALID() Valid-flag of L1 Instruction Cache Line

Syntax: **CACHE.IC.VALID(***<set>***,***<way>***)**

Returns TRUE if the specified set/way of the instruction (or unified) cache is VALID.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.IC.VALIDMASK() Valid-flag mask of L1 Instruction Cache Line

Syntax: **CACHE.IC.VALIDMASK(***<set>***,***<way>***)**

Returns a value containing all VALID flags of the specified set/way of the instruction (or unified) cache. Use for caches with multiple valid flags/sectors per cache line.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.L2.DIRTY() Dirty-flag of L2 Cache Line

Syntax: **CACHE.L2.DIRTY(**<*set*>**,**<*way*>**)**

Returns TRUE if the specified set/way of the L2 cache has the DIRTY flag set. If the cache ways are split up in sectors, the result is true if at least one of the DIRTY flags is set.

Parameter and Description:

<*set*>	**Parameter Type**: Decimal or hex or binary value.
<*way*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.L2.DIRTYMASK() Dirty-flag mask of L2 Cache Line

Syntax: **CACHE.L2.DIRTYMASK(**<*set*>**,**<*way*>**)**

Returns a value containing all DIRTY flags of the specified set/way of the L2 cache. Use for caches with multiple dirty flags/sectors per cache line.

Parameter and Description:

<*set*>	**Parameter Type**: Decimal or hex or binary value.
<*way*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.L2.LRU() LRU information of L2 Cache Line

[build 52317]

Syntax: **CACHE.L2.LRU(**<*set*>**)**

Returns the index of the least recently used (next to be replaced) way of the specified set. Only a few processors provide this information.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.L2.SHARED() Shared-flag of L2 Cache Line

Syntax: **CACHE.L2.SHARED(***<set>***,***<way>***)**

Returns TRUE if the specified set/way of the L2 cache has the SHARED flag set. If the cache ways are split up in sectors, the result is true if at least one of the SHARED flags is set.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.L2.SHAREDMASK() Shared-flag mask of L2 Cache Line

Syntax: **CACHE.L2.SHAREDMASK(***<set>***,***<way>***)**

Returns a value containing all SHARED flags of the specified set/way of the L2 cache. Use for caches with multiple shared flags/sectors per cache line.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.L2.TAG() Address Tag of L2 Cache Line

Syntax: **CACHE.L2.TAG(***<set>***,***<way>***)**

Returns the address tag of the specified set/way of the L2 cache.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.L2.VALID() Valid-flag of L2 Cache Line

Syntax: **CACHE.L2.VALID(**<*set*>**,**<*way*>**)**

Returns TRUE if the specified set/way of the L2 cache has the VALID flag set. If the cache ways are split up in sectors, the result is true if at least one of the VALID flags is set.

Parameter and Description:

<*set*>	**Parameter Type**: Decimal or hex or binary value.
<*way*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.L2.VALIDMASK() Valid-flag mask of L2 Cache Line

Syntax: **CACHE.L2.VALIDMASK(**<*set*>**,**<*way*>**)**

Returns a value containing all VALID flags of the specified set/way of the L2 cache. Use for caches with multiple valid flags/sectors per cache line.

Parameter and Description:

<*set*>	**Parameter Type**: Decimal or hex or binary value.
<*way*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.L3.DIRTY() Dirty-flag of L3 Cache Line

Syntax: **CACHE.L3.DIRTY(**<*set*>**,**<*way*>**)**

Returns TRUE if the specified set/way of the L3 cache has the DIRTY flag set. If the cache ways are split up in sectors, the result is true if at least one of the DIRTY flags is set.

Parameter and Description:

<*set*>	**Parameter Type**: Decimal or hex or binary value.
<*way*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.L3.DIRTYMASK() Dirty-flag of L3 Cache Line

Syntax: **CACHE.L3.DIRTYMASK(***<set>***,***<way>***)**

Returns a value containing all DIRTY flags of the specified set/way of the L3 cache. Use for caches with multiple dirty flags/sectors per cache line.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.L3.LRU() LRU information of L3 Cache Line

[build 52317]

Syntax: **CACHE.L3.LRU(***<set>***)**

Returns the index of the least recently used (next to be replaced) way of the specified set. Only a few processors provide this information.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

CACHE.L3.TAG() Address Tag of L3 Cache Line

Syntax: **CACHE.L3.TAG(***<set>***,***<way>***)**

Returns the address tag of the specified set/way of the L3 cache.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

Syntax: **CACHE.L3.VALID(***<set>*,*<way>***)**

Returns TRUE if the specified set/way of the L3 cache has the VALID flag set. If the cache ways are split up in sectors, the result is true if at least one of the VALID flags is set.

Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Boolean.

CACHE.L3.VALIDMASK() Valid-flag mask of L3 Cache Line

Syntax: **CACHE.L3.VALIDMASK(***<set>*,*<way>***)**

Returns a value containing all VALID flags of the specified set/way of the L3 cache. Use for caches with multiple valid flags/sectors per cache line.

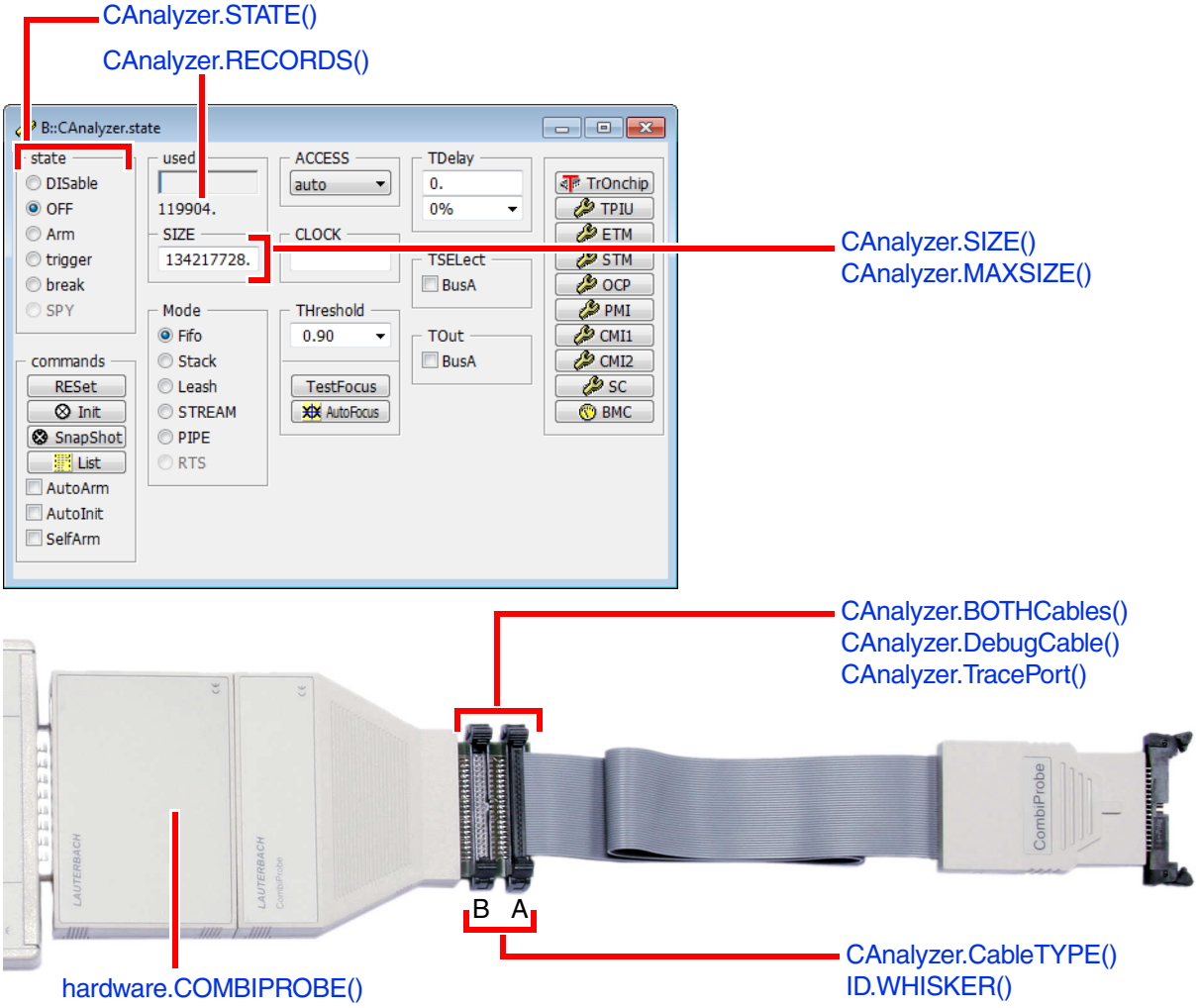
Parameter and Description:

<set>	**Parameter Type**: Decimal or hex or binary value.
<way>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

CAnalyzer Functions (CombiProbe, µTrace (MicroTrace))

This figure provides an overview of the return values of some of the CAnalyzer functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- CAnalyzer()
- CAnalyzer.CableTYPE()
- CAnalyzer.FEATURE()
- CAnalyzer.MAXSIZE()
- CAnalyzer.RECORD.ADDRESS()
- CAnalyzer.RECORD.OFFSET()
- CAnalyzer.RECORDS()
- CAnalyzer.SIZE()
- CAnalyzer.TraceCLOCK()
- CAnalyzer.TracePort()
- CAnalyzer.BOTHCables()
- CAnalyzer.DebugCable()
- CAnalyzer.FIRST()
- CAnalyzer.PIN()
- CAnalyzer.RECORD.DATA()
- CAnalyzer.RECORD.TIME()
- CAnalyzer.REF()
- CAnalyzer.STATE()
- CAnalyzer.TraceCONNECT()
- CAnalyzer.TRACK.RECORD()

CAnalyzer() Check if CAnalyzer command group is available

[Go to figure]

Syntax: **CAnalyzer()**

Returns TRUE if a Compact Analyzer hardware is available, meaning that the **CAnalyzer.*** command group and the **<trace>.METHOD CAnalyzer** selection can be used.

To query for specific hardware, use **hardware.COMBIPROBE()**, **hardware.UTRACE()** or **ID.WHISKER()**. To query for specific features, e. g. whether parallel trace can be used or whether a configurable trace termination is available, use **CAnalyzer.FEATURE()**.

Refer to **"CAnalyzer"** in General Commands Reference Guide C, page 26 (general_ref_c.pdf) for more information.

Return Value Type: Boolean.

CAnalyzer.BOTHCables() TRUE if both debug cables are plugged

[build 49264 - DVD 02/2014] [Go to figure]

Syntax: **CAnalyzer.BOTHCables()**

Returns TRUE if both DebugCableA and DebugCableB are plugged to the CombiProbe or µTrace (MicroTrace).

Return Value Type: Boolean.

CAnalyzer.CableTYPE() Type of adapter

[Go to figure]

Syntax: **CAnalyzer.CableTYPE(***<int>***)**

Parameter Type: Decimal value.

Return Value Type: Decimal value.

Returns the type of adapter plugged to the Compact Analyzer hardware. This function is similar to the function **ID.WHISKER()**, except for the following differences:

- It returns a decimal value instead of a hexadecimal value.
- In simulator mode or if the plugged whisker is LA-4509 COMBIPROBE-IN-MIPI34, it returns the value 1.
- For the QuadProbe, it always returns 0.

Lauterbach recommends using **ID.WHISKER()** in new scripts.

Example:

```
;return value 1 means that a standard whisker cable is plugged at
;connector A
PRINT CAnalyzer.CableTYPE(0)
```

CAnalyzer.DebugCable() CombiProbe whisker cable is A or B

[build 49264 - DVD 02/2014] [Go to figure]

Syntax: **CAnalyzer.DebugCable()**

Returns which CombiProbe whisker cable is used for debugging: "A" or "B"

Return Value Type: String.

CAnalyzer.FEATURE() Query features of CAnalyzer hardware

[build 81635 - DVD 02/2017]

Syntax: **CAnalyzer.FEATURE(***<feature>***)**

Returns whether the Lauterbach hardware that provides the CAnalyzer functionality supports a certain feature. The return value of this function only depends on the Lauterbach hardware and possibly the TRACE32 software version, not on any user settings or the target system.

The main use of this function is for scripts to avoid using commands that may be locked.

Parameter Type: String.

Return Value Type: Boolean.

Return Value and Description:

TRUE()	Feature is supported by current hardware and software. This does not imply that the feature is usable with the current target system.
FALSE()	Feature is not supported or not known to the current software version.

The following *<feature>* parameters are currently supported:

<feature>	Description
SWV	Hardware supports ARM Serial Wire Viewer / Serial Wire Output.
SWV.SAMPLE	Hardware supports setting sample delays for individual bits of an SWV transmission and also supports focus diagnostics (see **CAnalyzer.ShowFocus**, implies **SWV**).
PTI	Hardware supports any parallel trace port.
PTI.SAMPLE	Hardware supports setting individual sample delays for individual data lines of a parallel trace port and also supports focus diagnostics (see **CAnalyzer.ShowFocus**, implies **PTI**).
PTI.TERMINATION	Hardware supports parallel termination for the data lines of a parallel trace port that can be controlled using the command **CAnalyzer.TERMination** (implies **PTI**).
TPIU	Hardware supports decoding and filtering options for ARM TPIU / CoreSight trace ports (implies **PTI**).
TPIU.SAMPLE	Shorthand for **TPIU** and **PTI.SAMPLE**.
TPIU.TERMINATION	Shorthand for **TPIU** and **PTI.TERMINATION**.
STP	Hardware supports decoding and filtering options for direct STM (STPv1 or STPv2) output (implies **PTI**).
STP.SAMPLE	Shorthand for **STP** and **PTI.SAMPLE**.

Example:

```
; Script produced by STORE * CAnalyzerFocus. To avoid "Command locked"
; error messages when executing this script on different hardware, all
; commands are guarded by explicit checks whether hardware support is
; available.

IF CAnalyzer()
(
    IF CAnalyzer.FEATURE(TPIU.SAMPLE)
    (
        CAnalyzer.SAMPLE D0 -1.555
        CAnalyzer.SAMPLE D1 -2.177
        CAnalyzer.SAMPLE D2 -2.177
        CAnalyzer.SAMPLE D3 -1.866
    )
    CAnalyzer.THreshold 1.3
    IF CAnalyzer.FEATURE(TPIU.TERMination)
    (
        CAnalyzer.TERMination ON
    )
)
```

CAnalyzer.FIRST() Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **CAnalyzer.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

CAnalyzer.MAXSIZE() Get max. size of trace buffer in records

[Go to figure]

Syntax: **CAnalyzer.MAXSIZE()**

Returns the maximum possible size of the Compact Analyzer trace buffer in records (value depends on the currently selected tracing mode, too).

Return Value Type: Decimal value.

CAnalyzer.PIN() Status of trace pins

[build 62704 - DVD 09/2015]

Syntax: **CAnalyzer.PIN(**<*pin_name*>**)**

Returns the status of trace pins as binary.

Parameter Type: String.

Return Value Type: Binary value.

CAnalyzer.RECORD.ADDRESS() Get address recorded in trace record

[build 38764]

Syntax: **CAnalyzer.RECORD.ADDRESS(**<*record_number*>**)**

Returns the sampled address (access class and offset) from the specified record. For an example, see **Analyzer.RECORD.ADDRESS()**.

Parameter Type: Decimal value.

Return Value Type: Address.

CAnalyzer.RECORD.DATA() Get data recorded in trace record

[build 38764]

Syntax: **CAnalyzer.RECORD.DATA(**<*record_number*>**)**

Returns the sampled data of the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

CAnalyzer.RECORD.OFFSET() Get address in trace record as number

Syntax: **CAnalyzer.RECORD.OFFSET(***<record_number>***)**

Returns the address-offset of the sampled address from the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

CAnalyzer.RECORD.TIME() Get timestamp of trace record

[build 38764]

Syntax: **CAnalyzer.RECORD.TIME(***<record_number>***)**

Returns the timestamp of the specified record. For an example, see **Analyzer.RECORD.TIME()**.

Parameter Type: Decimal value.

Return Value Type: Time value.

CAnalyzer.RECORDS() Get number of used trace records

[Go to figure]

Syntax: **CAnalyzer.RECORDS()**

Returns the number of records in the Compact Analyzer.

If the state is **OFF** and the current mode is **STREAM**, this function will block until all buffered data has been received.

Return Value Type: Decimal value.

CAnalyzer.REF() Get record number of reference record

Syntax: **CAnalyzer.REF()**

Returns the number of the selected reference record in the Compact Analyzer trace.

Return Value Type: Decimal value.

CAnalyzer.SIZE() Get current trace buffer size in records

[Go to figure]

Syntax: **CAnalyzer.SIZE()**

Returns the actual defined logical size of the Compact Analyzer trace buffer in records.

Return Value Type: Decimal value.

CAnalyzer.STATE() Get state of Compact Analyzer

Go to figure]

Syntax: **CAnalyzer.STATE()**

Returns the state of the Compact Analyzer.

If the state is **OFF** and the current mode is **STREAM**, this function will block until all buffered data has been received.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state
8	**SPY** state
9	**OFF** state, but data is still being processed (**PIPE** and **RTS** modes only)

Syntax: **CAnalyzer.TraceCLOCK()**

Get the current value of the **CAnalyzer.TraceCLOCK** setting. This is useful to query the SWV baud rate detected by the **CAnalyzer.AutoFocus** command from a PRACTICE script.

If the frequency was not yet set or automatic detection was unsuccessful, a negative value is returned. Otherwise, the frequency is returned in Hz.

Return Value Type: Decimal value.

Example 1: Print the detected trace clock after automatic detection

```
; Set the trace output to Serial Wire Viewer mode
TPIU.PortType SWV

; Perform automatic rate detection
CAnalyzer.AutoFocus

IF CAnalyzer.TraceCLOCK()>=0.
(
    PRINT "Detected baud rate is " \
          FORMAT.FLOAT(0,3,CAnalyzer.TraceCLOCK()/1.0e6) \
          " MHz."
)
```

Example 2: A script to automatically find a suitable SWV prescaler is included with your TRACE32 installation. To access this script, run the following command

```
PSTEP ~~/demo/arm/etc/serial_wire_viewer/swv_autocalibration.cmm
```

CAnalyzer.TraceCONNECT() Name of trace sink of the SoC

[build 80222 - DVD Feb/2017]

Syntax: **CAnalyzer.TraceCONNECT()**

Returns the name of the currently selected trace sink of the SoC. In case no trace-sink is selected/available, the function returns **NONE**. The trace sink is selected with the **<trace>.TraceCONNECT** command.

Return Value Type: String.

Example: See **Onchip.TraceCONNECT()**.

CAnalyzer.TracePort() CombiProbe whisker cable is A or B

[build 49264 - DVD 02/2014] [Go to figure]

Syntax: **CAnalyzer.TracePort()**

Returns which CombiProbe whisker cable is used for tracing: "A" or "B"

Return Value Type: String.

CAnalyzer.TRACK.RECORD() Get record number matching search

Syntax: **CAnalyzer.TRACK.RECORD()**

After a successful search operation, this function returns the record number. For an example, see **Analyzer.TRACK.RECORD()**.

Return Value Type: Decimal value.

CERBERUS.IOINFO() IOINFO of Cerberus module

TriCore [build 66413 - DVD 02/2016]

Syntax: **CERBERUS.IOINFO()**

Returns the IOINFO of the Cerberus module.

Return Value Type: Hex value.

CERBERUS.IOINFO.IFLCK() TRUE if IF_LCK bit in Cerberus INONFO set

TriCore [build 66413 - DVD 02/2016]

Syntax: **CERBERUS.IOINFO.IFLCK()**

Returns TRUE if the IF_LCK bit in the Cerberus IOINFO is set, FALSE otherwise.

Return Value Type: Boolean.

CHIP.EmulationDevice() TRUE if emulation device

[build 74844 - DVD 09/2016]

Syntax: **CHIP.EmulationDevice()**

For PowerPC: This function returns TRUE if an Emulation Device was detected.

For TriCore/PCP/GTM/C166/XC2000ED: This function returns TRUE if an Emulation Device was selected in the CPU list, e.g. TC1797ED, TC275TE.

Return Value Type: Boolean.

CHIP.STEPping() Major silicon step of an TriCore AURIX device

TriCore [build 49713 - DVD 02/2014]

Syntax: **CHIP.STEPping()**

Returns the major silicon step of an TriCore AURIX device.

Return Value Type: String.

CIProbe Functions (Analog Probe for CombiProbe or µTrace)

In This Section

See also

- CIProbe()
- CIProbe.ADC.ENABLE()
- CIProbe.ADC.SHUNT()
- CIProbe.MAXSIZE()
- CIProbe.RECORDS()
- CIProbe.SIZE()
- CIProbe.STATE()
- CIProbe.TRACK.RECORD()

CIProbe() TRUE if Compact Analyzer hardware

Syntax: **CIProbe()**

Returns TRUE if a Compact Analyzer (CombiProbe or µTrace (MicroTrace)) hardware is plugged and used with a logic analyzer/analog probe.

Return Value Type: Boolean.

CIProbe.ADC.ENABLE() TRUE if channel is enabled

Syntax: **CIProbe.ADC.ENABLE(***<channel>***)**

Returns TRUE if the specified analog channel of the Analog Probe is enabled.

Return Value Type: Boolean.

CIProbe.ADC.SHUNT() Get shunt-resistor value

Syntax: **CIProbe.ADC.SHUNT(***<channel>***)**

Returns the shunt-resistor value of the specified current measurement *<channel>* of the Analog Probe.

Return Value Type: Float.

CIProbe.MAXSIZE() Get max. size of trace buffer in records

[build 38323 - DVD 08/2012]

Syntax: **CIProbe.MAXSIZE()**

Returns the maximum size of the Compact Analyzer (CombiProbe) trace buffer, which is assigned to the CIProbe, in records.

Return Value Type: Decimal value.

CIProbe.RECORDS() Get number of used trace records

Syntax: **CIProbe.RECORDS()**

The number of records currently stored in the trace buffer.

Return Value Type: Decimal value.

CIProbe.SIZE() Get current trace buffer size in records

[build 38323 - DVD 08/2012]

Syntax: **CIProbe.SIZE()**

Returns the actual defined logical size of the Compact Analyzer (CombiProbe) trace buffer which is assigned to the CIProbe, in records.

Return Value Type: Decimal value.

Syntax: **CIProbe.STATE()**

Returns the state of the Compact Analyzer for CIProbe.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state

CIProbe.TRACK.RECORD() Get record number matching search

Syntax: **CIProbe.TRACK.RECORD()**

After a successful search operation, this function returns the record number. For an example, see **Analyzer.TRACK.RECORD()**.

Return Value Type: Decimal value.

CMIBASE() Base addresses of CMI modules

[build 58596]

Syntax: **CMIBASE(***<instance>***)**

<instance>: **1.** | **2.**

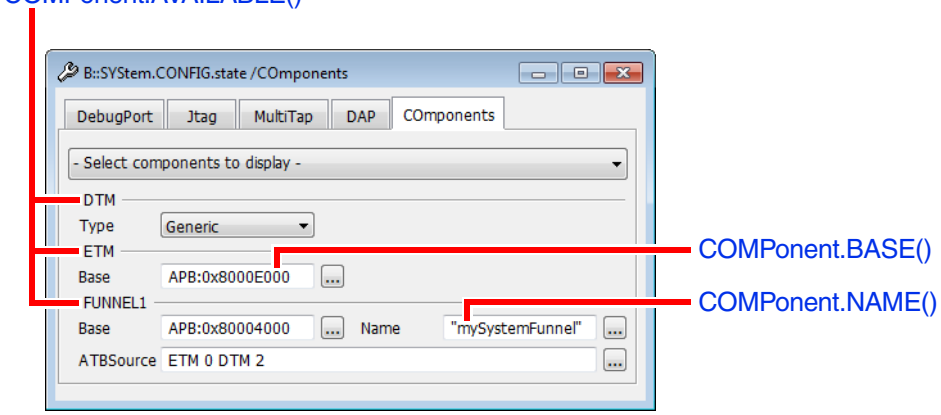
Returns the base address of the primary or secondary CMI module. A base address is set with the command **SYStem.CONFIG.CMI.Base**.

Parameter Type: Decimal value.

Return Value Type: Address.

COMPonent Functions

This figure provides an overview of the return values of the COMPonent functions. For descriptions of the illustrated functions, see below.



In This Section

See also

❏ COMPonent.AVAILABLE() ❏ COMPonent.BASE() ❏ COMPonent.NAME() ❏ COMPonent.TYPE()

COMPonent.AVAILABLE() TRUE if debug/trace peripherals available on CPU

[Go to figure]

Syntax: **COMPonent.AVAILABLE("***<component_name>***")**

Returns TRUE if the specified debug/trace peripheral *<component_name>* is available on this CPU.

Parameter Type: String.

Return Value Type: Boolean.

Example:

```
; set up base address of ETB1 and make it available
SYStem.CONFIG.ETB1.Base 0x123

; prints TRUE
PRINT COMPonent.AVAILABLE("ETB1")
```

COMPonent.BASE() Base address of debug/trace peripherals

[Go to figure]

Syntax: **COMPonent.BASE("***<component_name>***",***<core>***)**

Returns the base address for the specified debug/trace peripheral *<component_name>* and *<core>*.

Parameter and Description:

<component_name>	**Parameter Type**: String.
<core>	**Parameter Type**: Decimal value. In case *<core>* is **-1**, the current core is used.

Return Value Type: Address.

Example:

```
; set up base address of ETB1
SYStem.CONFIG.ETB1.Base 0x123

; print offset of base address (123)
PRINT ADDRESS.OFFSET(COMPonent.BASE("ETB1",-1))
```

COMPonent.NAME() Name of debug/trace peripherals

[build 94361 - DVD 09/2018] [Go to figure]

Syntax: **COMPonent.NAME("***<component_name>***",***<core>***)**

Returns the user-defined name for the specified debug/trace peripheral *<component_name>* and *<core>*, if a name was defined for that component.

Parameter and Description:

<component_name>	**Parameter Type**: String.
<core>	**Parameter Type**: Decimal value. In case *<core>* is **-1**, the current core is used.

Return Value Type: String. An empty string is returned if the component does not exist or has no user-defined name.

Example:

```
; assign a user-defined name to the 1st CoreSight Funnel
SYStem.CONFIG.FUNNEL1.Name "STM-Funnel"

; print name of the 1st Funnel
ECHO COMPonent.NAME("FUNNEL1",-1)
```

COMPonent.TYPE() Type of debug/trace peripherals

[build 95283 - DVD 09/2018]

Syntax: **COMPonent.TYPE("***<component_name>***")**

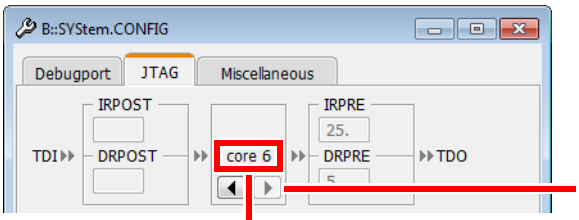
Returns the user-defined type for the specified debug/trace peripheral *<component_name>*.

Parameter Type: String.

Return Value Type: String. An empty string is returned if the component does not exist.

CORE Functions

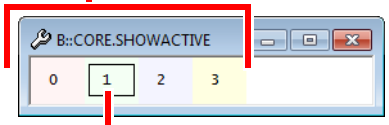
These figures provide an overview of the return values of some of the functions. For descriptions of the illustrated functions and the functions not shown here, see below.



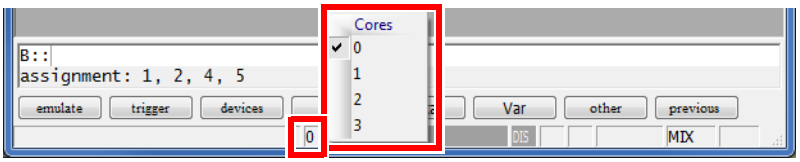
`SYStem.CONFIG.CoreNumber 6.` informs TRACE32 that this multicore chip has six cores. (The right arrow button is deactivated at core 6.)

CONFIGNUMBER()

CORE.NUMBER()



CORE()



The same values as in the CORE.SHOWACTIVE window (left)

In This Section

See also

- CORE
- CORE()
- CORE.ISASSIGNED()
- CORE.NAMES()
- CORE.PHYSICALTOLOGICAL()
- CORENUMBER()
- CONFIGNUMBER()
- CORE.ISACTIVE()
- CORE.LOGICALTOPHYSICAL()
- CORE.NUMBER()
- CORENAME()

[build 34251 - DVD 02/2012] [Go to figure]

Syntax: **CONFIGNUMBER()**

Returns the number of cores as set by the command **SYStem.CONFIG.CoreNumber**.

Return Value Type: Decimal value.

Example:

```
;Inform TRACE32 about the total number of cores of a multicore chip
SYStem.CONFIG.CoreNumber 6.

;Specify the location of the debug registers  (CoreSight ARM only)
SYStem.CONFIG COREBASE 100, 200, 300, 400, 500, 600

;Start core assignment at this <core> of this <chip>
SYStem.CONFIG.CORE                1.             1.

;- The cores 1, 2, 4, 5 (= four cores) are assigned to the TRACE32
;  PowerView GUI
;- The cores 3 and 6 are skipped (= two cores)
CORE.ASSIGN 1. 2. 4. 5.

SYStem.Up

;Open the configuration window.
SYStem.CONFIG.state /Jtag  ;When clicking the right arrow button on
                           ;the JTAG tab, you cannot go beyond core 6.

;Returns 6 because you have informed TRACE32 that this multicore
;chip has 6 cores.
PRINT CONFIGNUMBER()
```

CORE() Get the selected core

[Go to figure]

Syntax: **CORE()**

Returns the core selected with the **CORE.select** or **CORE.List** command.

Return Value Type: String.

Syntax: **CORE.ISACTIVE("*<core>*")**

Returns TRUE if the specified core is active, FALSE if the core is inactive.

Parameter Type: String. If the *<core>* consists of just a number, then append a dot; see example 2. The core names are displayed in the **Cores** drop-down list of the state line and in the **CORE.SHOWACTIVE** window. If the string is empty, then the state of the currently selected core is returned.

Return Value Type: Boolean.

Example 1: This script line shows how to specify a core name in a big.LITTLE system

```
PRINT CORE.ISACTIVE("1b")    ;prints TRUE to the message line if the core
                             ;named "1b" is active
```

Example 2: This script line shows how to specify a core name in an SMP system

```
PRINT CORE.ISACTIVE("3.")    ;prints TRUE to the message line if core
                             ;number 3 is active
                             ;remember to append a dot to the core number
```

Example 3:

```
PRINT CORE.ISACTIVE(CORE.NAMES(5.))
```

Syntax: **CORE.ISASSIGNED(***<core_number>***)**

Returns TRUE if a physical core is assigned to the current debug session, FALSE otherwise. To assign cores to TRACE32, use **CORE.ASSIGN**.

Parameter and Description:

<core_number>	**Parameter Type**: Decimal value. Number of a physical core. Range: **1.** <= *core_number* <= **CONFIGNUMBER()** • Min.: **1.** • Max.: Return value of **CONFIGNUMBER()**

Return Value Type: Boolean.

Example:

```
;The physical cores 1, 3, 5 are assigned to the debug session.
;The other cores are skipped.
CORE.ASSIGN 1. 3. 5.

SYStem.Up

;Returns FALSE in this example because the physical core 2 was skipped.
IF CORE.ISASSIGNED(2.)==TRUE()
(
  PRINT CORE.PHYSICALTOLOGICAL(2)
)
```

Syntax: **CORE.LOGICALTOPHYSICAL(***<core_number>***)**

Returns the physical core number of a logical core.

Parameter and Description:

<core_number>	**Parameter Type**: Decimal value. Core number of a logical core. Range: **0.** <= *core_number* <= **CORE.NUMBER()** • Min.: **0.** • Max.: Return value of **CORE.NUMBER()-1.**

Return Value Type: Decimal value.

Return Value and Description:

1 ... *<n>*	Number of the physical core. The highest number *<n>* equals the return value of **CONFIGNUMBER()**.
Error message	The error message “value too large” is displayed if *<core_number>* is out of bounds.

Example:

```
;The physical cores 1, 2, 3, 4 are assigned to the debug session.
CORE.ASSIGN 1. 2. 3. 4.

SYStem.Up

PRINT CORE.LOGICALTOPHYSICAL(0.) CORE.LOGICALTOPHYSICAL(1.)
PRINT CORE.LOGICALTOPHYSICAL(2.) CORE.LOGICALTOPHYSICAL(3.)
;Result:
;The logical  core numbers 0, 1, 2, 3 correspond to
;the physical core numbers 1, 2, 3, 4
```

See also: **CORE.PHYSICALTOLOGICAL()**, **CORE.ISASSIGNED()**

Syntax: **CORE.NAMES(**<*index*>**)**

Retrieves all physical core names of the SMP system.

Parameter Type: Decimal value.

Return Value Type: String.

Example: The **CORE.NAMES()** function is used to loop through the cores assigned to TRACE32, and the **CORE.ISACTIVE()** function returns whether a core is active or inactive.

```
SYStem.CPU Cortexa15A7
SYStem.CONFIG.CoreNumber 14.

;assign the cores of a big.LITTLE SMP system to TRACE32
CORE.ASSIGN BIGLITTLE 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14.

;enter debug mode
SYStem.Up

GOSUB printstate

CORE.SHOWACTIVE

ENDDO

printstate:

&i=0.
nextcore:
&name=CORE.NAMES(&i)
IF "&name"!=""
(
   &state=CORE.ISACTIVE("&name")
   IF &state==TRUE()
     PRINT "active core : &name"
   &i=&i+1.
   GOTO nextcore
)

RETURN
```

CORENAME() Name of core within selected chip

[Go to figure]

Syntax: **CORENAME()**

Returns the name of the core within a selected chip.

Return Value Type: String. The name and spelling is the same that is shown in the **SYStem.CONFIG.state** window on the **Miscellaneous** tab.

CORE.NUMBER() Number of logical cores

[build 65779 - DVD 09/2015] [Go to figure] [Examples]

Syntax: **CORE.NUMBER()**
CORENUMBER() (deprecated)

Returns the number of logical cores assigned to the current debug session.

For each assigned core, the **CORE.SHOWACTIVE** window and the **Cores** list contain one entry. To assign cores to TRACE32, use the command **CORE.ASSIGN**.

NOTE:	Do not confuse the number of cores currently assigned to TRACE32 with the actual number of cores physically implemented on a multicore chip. It is possible to assign fewer cores to TRACE32 than there are on a multicore chip.

Return Value Type: Decimal value.

Return Value and Description:

1	Single-core debug session.
Integer greater than **1**	SMP debug session (Symmetrical Multicore Processing).

Example 1: Let's assume a multicore chip has 6 cores, and just 4 of them are assigned to TRACE32. As a result, **CORE.NUMBER()** returns 4.

```
;select a multicore chip
SYStem.CPU CortexA7MPCore

;Inform TRACE32 about the total number of cores of a multicore chip
SYStem.CONFIG.CoreNumber 4.

;Specify the location of the debug registers   (CoreSight ARM only)
SYStem.CONFIG COREBASE DAP:0x100 DAP:0x200 DAP:0x300 DAP:0x400
;Start core assignment at this <core> of this <chip>
SYStem.CONFIG.CORE                1.             1.

;- The cores 1, 2, 4 (= three cores) are assigned to the TRACE32
;  PowerView GUI
;- The core 3 is skipped
CORE.ASSIGN 1. 2. 4.

;Prints 3 because three cores were assigned to the debug session
PRINT CORE.NUMBER()
```

Example 2: A single-core debug session is set up for core 3 of a multicore chip.

```
SYStem.CPU CortexA7MPCore       ;select a multicore chip

;Inform TRACE32 about the total number of cores of a multicore chip
SYStem.CONFIG.CoreNumber 4.

;Specify the location of the debug registers   (CoreSight ARM only)
SYStem.CONFIG COREBASE DAP:0x100 DAP:0x200 DAP:0x300 DAP:0x400

;Start core assignment at this <core> of this <chip>
SYStem.CONFIG.CORE                1.             1.

CORE.ASSIGN 3.                  ;assign only core 3 to the debug session
```

Syntax: **CORE.PHYSICALTOLOGICAL(***<core_number>***)**

Returns the logical core number of a physical core. If the core is not assigned to the current debug session, an error is shown in the message line.

Parameter and Description:

<core_number>	**Parameter Type**: Decimal value. Number of a physical core. Range: **1.** <= *core_number* <= **CONFIGNUMBER()** • Min.: **1.** • Max.: The return value of **CONFIGNUMBER()**

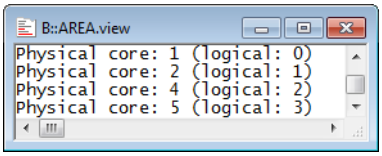
Return Value Type: Decimal value.

Return Value and Description:

0 ... *<n>*	Number of the logical core. The highest number *<n>* equals the return value of **CORE.NUMBER()**.
Error message	The error message “value not allowed” is displayed if *<core_number>* refers to a physical core that is not assigned to the current debug session. See **CORE.ISASSIGNED()**.

Example: For each physical core assigned to the TRACE32 PowerView GUI, the physical and logical core numbers are printed to an **AREA.view** window.

```
&physical=1.    ;the number of the first physical core is always 1

WHILE &physical<=CONFIGNUMBER()
(
  IF CORE.ISASSIGNED(&physical)==TRUE()
  (
    ;returns the physical core number for this logical core
    PRINT "Physical core: " &physical
    PRINT %CONTinue " (logical: " CORE.PHYSICALTOLOGICAL(&physical) ")"
  )
  &physical=&physical+1. ;increment to next physical core
)
```



Reason for the gaps in the sequence of physical core numbers:
Only the cores 1, 2, 4, and 5 were assigned with the command CORE.ASSIGN 1. 2. 4. 5.
The cores 3 and 6 were intentionally skipped.

See also: **CORE.LOGICALTOPHYSICAL()**

Count Functions

The **Count** functions give access to the measurement results of the **Count.state** window.

In This Section

See also

- Count
- Count.Frequency()
- Count.LEVEL()
- Count.Time()
- Count.VALUE()

Count.Frequency() Frequency of last measurement

[build 67178 - DVD 02/2016]

Syntax: **Count.Frequency()**

Returns the frequency in Hz of the last measurement started with the command **Count.GO**. If the last measurement was not a frequency measurement, the function aborts with an error.

Return Value Type: Decimal value.

Example:

```
; select signal source (here: MCKO) and operation mode
Count.Select MCKO
Count.Mode Frequency

; do measurement
Count.Go

PRINT "F=" Count.Frequency() "Hz"                ;Output: "F=169440Hz"
```

Count.LEVEL() Level of frequency counter input

Syntax: **Count.LEVEL()**

Current logical level of the frequency counter input.

Return Value Type: Time value.

Count.Time() Time of last measurement

Syntax: **Count.Time()**

Returns the time of the last period or pulse duration measurement started with the command **Count.GO**. If the last measurement was not a period or pulse duration measurement, the function aborts with an error.

Return Value Type: Time value.

Example:

```
; select signal source (here: PowerProbe eXt.3) and operation mode
Probe.CSelect eXt.3
Count.Mode Period

; do measurement
Count.Go

PRINT "Period=" Count.Time()        ;Output: "Period=0.000105900s"
```

Count.VALUE() Samples of the Count.GO command

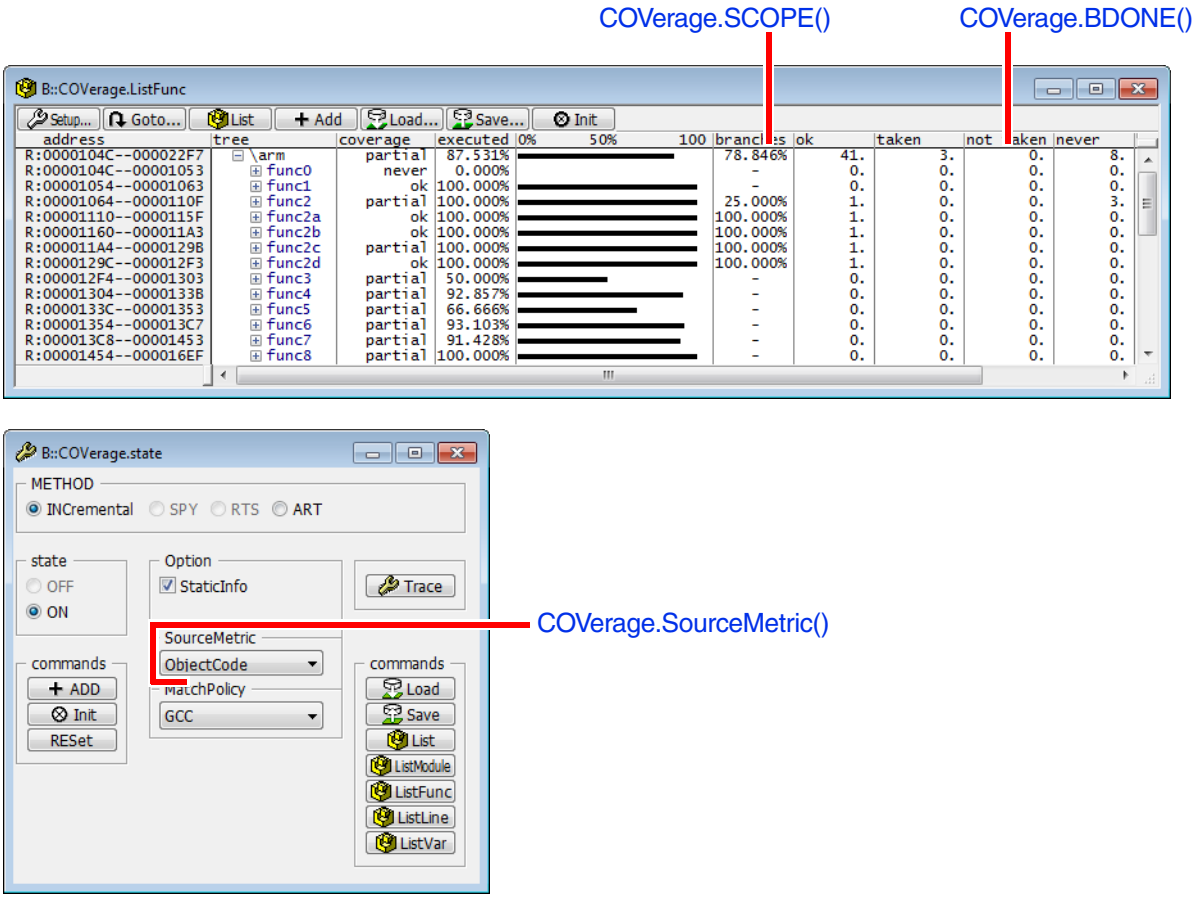
Syntax: **Count.VALUE()**

If the **Counter** is configured to count events, this function returns the result samples when the command **Count.GO** is issued. If the Counter is configured for frequency, period or pulse duration measurement, the result of the function is undefined.

Return Value Type: Hex value.

COVerage Functions

This figure provides an overview of the return values of some of the functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- ■ COVerage
- ❑ COVerage.BDONE()
- ❑ COVerage.LOAD.KEY()
- ❑ COVerage.SCOPE()
- ❑ COVerage.SourceMetric()
- ❑ COVerage.TreeWalk()

COVerage.BDONE() Byte count of all executed instructions

[Go to figure]

Syntax: **COVerage.BDONE(**<*address_range*>**)**

Returns a byte count of all instructions that have been executed.

Parameter Type: Address range.

Return Value and Description:

-1	The data is invalid, or the user has clicked **Stop** on the TRACE32 main toolbar to abort the function.
Greater than or equal to **0**	Byte count of all executed instructions within the specified range.

Return Value Type: Decimal value.

Example: Different types of input parameters are shown. The function accepts both address ranges and other functions that return address ranges as input.

```
; address range
PRINT COVerage.BDONE(P:0x1FFFC220--P:0x1FFFC299)

; function main
PRINT COVerage.BDONE(Var.RANGE(main))

; module crt0 of the executable sort
PRINT COVerage.BDONE(sYmbol.RANGE(\\sort\crt0))
```

NOTE:	The specified range has to be identical to the limits of the function / functions being tested. Otherwise inaccuracies may occur.

COVerage.LOAD.KEY() Key from last ACD file

[build 113973 - DVD 02/2020]

Syntax: **COVerage.LOAD.KEY()**

Returns the key from the last ACD file (*.acd) that was loaded with **COVerage.LOAD**.

Return Value Type: String.

Syntax: **COVerage.SCOPE(**<*address_range*>**)**

Returns the degree of code coverage within the specified range.

Parameter Type: Address range.

Return Value Type: Decimal value.

Return Value and Description:

-1	Returns **-1** if the data is invalid or if the user has clicked **Stop** on the TRACE32 main toolbar to abort the function.
0	At least one instruction not fully executed.
1	At least one branch only taken.
2	At least one branch never taken.
3	At least one branch never and one only taken.
4	Fully executed.

Example: This script demonstrates the usage of the **COVerage.SCOPE()** function with different types of input parameters. The function **COVerage.SCOPE()** accepts both address ranges and other functions that return address ranges as input.

```
; address range
COVerage.SCOPE(P:0x1FFFC220--P:0x1FFFC299)

; function main
COVerage.SCOPE(Var.RANGE(main))

; module crt0 of the executable sort
COVerage.SCOPE(sYmbol.RANGE(\\sort\crt0))
```

NOTE:	The specified range has to be identical to the limits of the function / functions being tested. Otherwise inaccuracies may occur.

Syntax: **COVerage.SourceMetric()**

Returns the current code coverage criterion for HLL lines that was set with the command **COVerage.Option.SourceMetric**.

Return Value Type: String.

Return Value and Description:

ObjectCode	The tags of the corresponding block of assembly instructions are mapped to the HLL line.
Statement	Statement coverage
Decision	Decision coverage
MCDC	Modified condition/decision coverage (MC/DC)

For details, see **COVerage.Option.SourceMetric**.

Example: See ~~/demo/coverage/multi_file_report/create_report.cmm.

Syntax:	**COVerage.TreeWalk(***<action>***)**
<action>:	**Init** | **Recurse** | **CONTinue**

Traverses the code coverage tree that has been prepared by the command **COVerage.TreeWalkSETUP**.

Parameter Type: String.

Parameter and Description:

Init	Returns the first node of the tree.
Recurse	Returns the next node of the tree.
Continue	Returns the next node on the same or higher level.

Return Value Type: String.

Example 1:

```
PRIVATE &node

COVerage.TreeWalkSETUP                        ; create a tree with all
                                              ; code coverage symbols

&node=COVerage.TreeWalk("Init")               ; get the first tree element
WHILE "&node"!=""
(
   IF STRing.SCAN("&node","\",0.)==0.         ; element is a module
   (
      PRINT "The next module is: &node"
   )
   ELSE IF STRing.SCAN("&node","--",0.)>-1.   ; element is an HLL line
   (
      PRINT "The next HLL line is: &node"
   )
   ELSE                                       ; element is a function
   (
      PRINT "The next function is: &node"
   )
   &node=COVerage.TreeWalk("Recurse")         ; get the next tree element
)
```

Example 2: A more complex demo script is included in your TRACE32 installation. To access the script, run this command:
`B::CD.PSTEP ~~/demo/coverage/multi_file_report/create_report.cmm`

CPU Functions

In This Section

See also

- SYStem.CPU
- CPU.ADDRESS()
- CPU.FEATURE()
- CPU.SUBFAMILY()
- CPUCOREVERSION()
- CPUFAMILY()
- CPUIS()
- CPU()
- CPU.ADDRESS.PhysicalINDEX()
- CPU.PINCOUNT()
- CPUBONDOUT()
- CPUDERIVATE()
- CPUHELP()
- CPUIS64BIT()

CPU.ADDRESS() Start address of memory section

ARC, TriCore [build 106881 - DVD 09/2019]

Syntax:	**CPU.ADDRESS(***<section>***)**
<section>: (ARC)	**XCCM** | **YCCM**
<section>: (TriCore)	**SFR** | **CSFR**

Returns the start address of the memory or register block *<section>*.

Parameter Type: String.

Return Value Type: Address.

CPU.ADDRESS.PhysicalINDEX() Section start address of given core

TriCore [build 125278 - DVD 09/2020]

Syntax:	**CPU.ADDRESS.PhysicalINDEX("***<section>***",***<core_number>***)**

Returns the start address of the memory or register block *<section>* for the given core.

Parameter and Description:

<section>	**Parameter Type**: String. Name of the memory or register block. For legal values see **CPU.ADDRESS()**.
<core_number>	**Parameter Type**: Decimal value. Number of a physical core. **Range**: 1. <= core_number <= **CONFIGNUMBER()** • MIN.: 1. • MAX.: The return value of **CONFIGNUMBER()**

Return Value Type: Address.

CPU.FEATURE() TRUE if CPU feature exists

[build 57631] [Example]

Syntax: **CPU.FEATURE(***<feature_string>***)**

Tests whether the selected CPU has a certain feature. The function returns TRUE if the CPU implements the specified feature. If the feature it not supported, or if *<feature_string>* is unknown, the function returns FALSE. The tables below show the generic and architecture dependent feature strings.

Parameter Type: String.

Parameter and Description:

All architectures - Allowed keywords for <feature_string>	
BMC [build 71934 - DVD 02/2016]	TRUE if the selected CPU implements benchmark counters / performance monitors.
PCSNOOP [build 67662 - DVD 02/2016]	TRUE if the program counter of the selected CPU can be read by the debugger while the CPU is running.

ARM architecture - Allowed keywords for <feature_string>	
HYPERVISOR	TRUE if the selected CPU has a hypervisor zone (ARM Virtualization Extension).
JAZELLE	TRUE if the selected CPU has the ARM Jazelle execution mode.
LPAE	TRUE if the selected CPU has an MMU and supports the Large Physical Address Extension (LPAE) mode.
MMU	TRUE if the selected CPU has an MMU.
SECURE	TRUE if the selected CPU has a secure zone (e.g. ARM TrustZone).

ARM architecture - Allowed keywords for <feature_string>	
VBAR	TRUE if the VBAR register is available in the selected CPU.
SECUREEL2	TRUE is the selected CPU has a non-secure and a secure EL2 mode.
THUMB	TRUE if the selected CPU supports thumb instruction set.

PowerPC / Power Architecture - Allowed keywords for <feature_string>	
BMC	TRUE if the selected core has performance monitor registers.
COREMPU	TRUE if the selected core has a memory protection unit (MPU).
EFPU	TRUE if the selected core has the EFPU floating point unit.
EFPU2	TRUE if the selected core has the EFPU2 floating point unit.
FLE	TRUE if the selected core supports the std. PowerPC opcodes
HYPERVISOR	TRUE if the selected core implements the hypervisor programming model.
L1ICACHE	TRUE if the selected core has an L1 instruction cache
L1DCACHE	TRUE if the selected core has an L1 data cache
L1UNIFIEDCACHE	TRUE if the selected core has a unified L1 cache
L2CACHE	TRUE if the selected core has an L2 cache
MMU	TRUE if the selected core has a memory management unit.
ONCHIP_PCFIFO	TRUE if the selected core has the PCFIFO on-chip trace (MPC55XX/56XX)
ONCHIP_TR2MEM	TRUE if the processor implements trace-to-memory (MPC57XX/QorIQ)
SPE	TRUE if the selected core has the signal processing engine (SPE).
VLE	TRUE if the selected core supports variable length encoded instruction set

RH850 architecture - Allowed keywords for <feature_string>	
FXU [build 108274 - DVD 09/2019]	TRUE if the selected CPU has FXU registers (extended floating point unit).

TriCore architecture - Allowed keywords for <feature_string>	
HSM [build 124752 - DVD 09/2020]	TRUE if the selected CPU has an HSM core.

Return Value Type: Boolean.

Example:

```
SYStem.CPU CortexA15          ;select CPU CortexA15
PRINT CPU.FEATURE(SECURE)     ;SECURE feature available? --> TRUE
```

CPU.PINCOUNT() — For internal usage only

[build 75532 - DVD 09/2016]

Syntax: **CPU.PINCOUNT()**

For internal use only.

CPUBONDOUT() — Name of boundout processor

Syntax: **CPUBONDOUT()**

Returns the name of the bondout processor.

Return Value Type: String.

CPUCOREVERSION() — Core or architecture version of CPU

Syntax: **CPUCOREVERSION()**

Returns the CPU's core or architecture version, e.g. "TriCore v1.3.1".

Return Value Type: String.

CPUDERIVATE() — Main part of processor name

Syntax: **CPUDERIVATE()**

Returns the main part of the processor name.

Return Value Type: String.

CPUFAMILY() — Family name of processor

Syntax: **CPUFAMILY()**

Returns the family name of the processor.

Return Value Type: String.

CPUHELP() — For internal usage only

Syntax: **CPUHELP()**

Reserved for internal usage.

Return Value Type: String.

CPUIS() TRUE if search string matches processor name

Syntax: **CPUIS(**<*search_string*>**)**

Returns a boolean value after comparing the currently selected name of the processor with the given search string. The search string can contain wildcards and characters are **not** interpreted as **case-sensitive**.

Parameter Type: String.

Return Value Type: Boolean.

Example:

```
IF (CPUIS("TC*ED"))
    PRINT "TriCore Emulation Device detected"

;   instead of the more complex alternative:
IF ((CPU()=="TC1798ED")||(CPU()=="TC1793ED")||(CPU()=="TC1791ED")
    PRINT "TriCore Emulation Device detected"
```

CPUIS64BIT() TRUE if 64-bit architecture

Syntax: **CPUIS64BIT()**
ARM64() (deprecated)

Returns TRUE when the target architecture is 64-bit e.g. ARM64.

Return Value Type: Boolean.

DAP Functions

In This Section

See also

❏ DAP.Available() ❏ DAP.USER0() ❏ DAP.USER1()

DAP.Available() TRUE if debugging via DAP is supported

Syntax: **DAP.Available()**

Returns TRUE if both the attached debug cable and the selected CPU support debugging via the DAP interface.

Return Value Type: Boolean.

DAP.USER<x>() Status of the DAP user pin

ICD-TriCore and ICD-C166

Syntax: **DAP.USER0()**
DAP.USER1()

Returns the status of the DAP `USER0` or `USER1` pin. Low level is FALSE, high level is TRUE. For details, see **"Application Note Debug Cable TriCore"** (app_tricore_ocds.pdf).

Return Value Type: Boolean.

Data Functions

In This Section

See also

- Data.AL.ERRORS()
- Data.Float()
- Data.Long()
- Data.Long.Byte()
- Data.Long.Long()
- Data.LongLong()
- Data.LongLong.LittleEndian()
- Data.PByte()
- Data.Quad.BigEndian()
- Data.Quad.LittleEndian()
- Data.Quad.Quad()
- Data.SByte()
- Data.Short.BigEndian()
- Data.SLong()
- Data.STRingN()
- Data.TByte()
- Data.Word.BigEndian()
- Data.Word.LittleEndian()
- Data.WSTRING()
- Data.WSTRING.LittleEndian()
- Data.Byte()
- Data.HByte()
- Data.Long.BigEndian()
- Data.Long.LittleEndian()
- Data.Long.Word()
- Data.LongLong.BigEndian()
- Data.MAU()
- Data.Quad()
- Data.Quad.Byte()
- Data.Quad.Long()
- Data.Quad.Word()
- Data.Short()
- Data.Short.LittleEndian()
- Data.STRing()
- Data.SUM()
- Data.Word()
- Data.Word.Byte()
- Data.Word.Word()
- Data.WSTRING.BigEndian()

Data.<value_width>() Memory contents in default endianness

[Examples]

Syntax:	**Data.**<*value_width*>**(**<*address*>**)**
<*value_width*>:	**Byte** | **Short** | **Word** | **TByte** | **Long** | **PByte** | **HByte** | **SByte** | **SLong** | **Quad** | **LongLong**
Full function name only required for HELP.Index:	**Data.Byte(**<*address*>**)** **Data.Short(**<*address*>**)** **Data.Word(**<*address*>**)** **Data.TByte(**<*address*>**)** **Data.Long(**<*address*>**)** **Data.PByte(**<*address*>**)** **Data.HByte(**<*address*>**)** **Data.SByte(**<*address*>**)** **Data.SLong(**<*address*>**)** **Data.Quad(**<*address*>**)** **Data.LongLong(**<*address*>**)**

Reads a value from memory using the specified <*width*> and <*address*>. The address must be classified, e.g. D:0x200, while each symbol has its own implicit memory class.

The possible memory classes depend on the target CPU. The appropriate list can be found in the chapter "*Access Classes*" of the particular **Processor Architecture Manual**.

<value_width>	Description of Data.<value_width>()
Byte	Returns a single byte from memory.
Short [build 20186 - DVD 12/2009]	Returns a word (16-bit) from memory. Same as function **Data.Word()**. **NOTE**: Please don't mix up the **function** **`Data.Short()`** and its function short form **`D.S()`** with the **command** **`Data.Set`** and its command short form **`D.S`** **NOTE**: **`D.S()`** was an alias for **`Data.STRing()`** in former versions of TRACE32.
Word	Returns a word (16-bit) from memory.
TByte	Returns a 3-byte value from memory.
Long	Returns a long value (32-bit) from memory. **NOTE**: Please don't mix up the **function** **`Data.Long()`** and its function short form **`D.L()`** with the **command** **`Data.List`** and its command short form **`D.L`**
PByte	Returns a 5-byte value from memory.
HByte	Returns a 6-byte value from memory.
SByte	Returns a 7-byte value from memory.
SLong	Returns a signed long value from memory - sign extended internally to a 64-bit value.
Quad	Returns a 64-bit value from memory. Same as function **Data.LongLong()**.
LongLong [build 20186 - DVD 12/2009]	Returns a 64-bit value from memory. Same as function **Data.Quad()**.

Parameter Type: Address.

Return Value Type: Hex value.

Example 1:

```
PRINT Data.Byte(D:0x200)    // long form of the function name

PRINT Data.Byte(a+17.)      // prints the memory contents of the address
                            // of target program symbol "a" plus offset
                            // 17 independent of the symbol type

PRINT Data.Byte(0x200)      // fails - parameter is a numeric constant
                            // and no address
                            // access mode specifier e.g. "D:" or "P:"
                            // is missing
```

Example 2:

```
PRINT Data.TByte(D:0x200)

PRINT Data.TByte(0x200)    // fails - parameter is a numeric constant
                           // and no address
                           // access mode specifier e.g. "D:" or "P:"
                           // is missing
```

Example 3:

```
// Set bit 4 of 32-bit value in memory
Data.Set A:0x10200 %Long Data.Long(A:0x10200)|0x10

// Clear second byte of 32-bit value in memory
Data.Set A:0x10300 %Long Data.Long(A:0x10300)&~0xff00
```

Syntax:	**Data.***<value_width>***.***<endianness>***(***<address>***)**
<value_width>:	**Byte** | **Short** | **Word** | **TByte** | **Long** | **PByte** | **HByte** | **SByte** | **SLong** | **Quad** | **LongLong**
<endianness>:	**LittleEndian** | **BigEndian**
Full function name only required for HELP.Index:	**Data.Short.BigEndian(***<address>***)** **Data.Short.LittleEndian(***<address>***)** **Data.Word.BigEndian(***<address>***)** **Data.Word.LittleEndian(***<address>***)** **Data.Long.BigEndian(***<address>***)** **Data.Long.LittleEndian(***<address>***)** **Data.LongLong.BigEndian(***<address>***)** **Data.LongLong.LittleEndian(***<address>***)** **Data.Quad.BigEndian(***<address>***)** **Data.Quad.LittleEndian(***<address>***)**

Reads a value of the size *<value_width>* from memory at the given address in the given byte order.

The address must be classified, e.g. D:0x200, while each symbol has its own implicit memory class.

The possible memory classes depend on the target CPU. The appropriate list can be found in the chapter "*Access Classes*" of the particular **Processor Architecture Manual**.

<value_width>.<endianness>	**Description of Data.<value_width>.<endianness>()**
Short.BigEndian	Returns a word (16-bit) from memory, while the byte order of the word is forced to big endian. Same as function **Data.Word.BigEndian()**.
Short.LittleEndian	Returns a word (16-bit) from memory, while the byte order of the word is forced to little endian. Same as function **Data.Word.LittleEndian()**.
Word.BigEndian	Returns a word (16-bit) from memory, while the byte order of the word is forced to big endian. Same as function **Data.Short.BigEndian()**.
Word.LittleEndian	Returns a word (16-bit) from memory, while the byte order of the word is forced to little endian. Same as function **Data.Short.LittleEndian()**.
Long.BigEndian	Returns a long value (32-bit) from memory, while the byte order of the word is forced to big endian.
Long.LittleEndian	Returns a long value (32-bit) from memory, while the byte order of the word is forced to little endian.
LongLong.BigEndian	Returns a 64-bit value from memory, while the byte order of the word is forced to big endian. Same as function **Data.Quad.BigEndian()**.

<value_width>.<endianness>	Description of Data.<value_width>.<endianness>()
LongLong.LittleEndian	Returns a 64-bit value from memory, while the byte order of the word is forced to little endian. Same as function **Data.Quad.LittleEndian()**.
Quad.BigEndian	Returns a 64-bit value from memory, while the byte order of the word is forced to big endian. Same as function **Data.LongLong.BigEndian()**.
Quad.LittleEndian	Returns a 64-bit value from memory, while the byte order of the word is forced to little endian. Same as function **Data.LongLong.LittleEndian()**.

Parameter Type: Address.

Return Value Type: Hex value.

Examples:

```
// Set bit 4 of 32-bit value in memory
Data.Set A:0x10200 %Long %BE Data.Long.BigEndian(A:0x10200)|0x10

// Clear second byte of 32-bit value in memory
Data.Set A:0x10300 %Long %BE Data.Long.BigEndian(A:0x10300)&~0xff00
```

Syntax:	**Data.***<value_width>***.***<access_width>***(***<address>***)**
<value_width>:	**WORD** | **LONG** | **QUAD**
<access_width>:	**Byte** | **Word** | **Long**
Full function name only required for HELP.Index:	**Data.Word.Byte(***<address>***)** **Data.Word.Word(***<address>***)** **Data.Long.Byte(***<address>***)** **Data.Long.Word(***<address>***)** **Data.Long.Long(***<address>***)** **Data.Quad.Byte(***<address>***)** **Data.Quad.Word(***<address>***)** **Data.Quad.Long(***<address>***)** **Data.Quad.Quad(***<address>***)**

Reads a value of the size *<value_width>* from memory at the given address. The memory is therefore accessed with *<access_width>* if this access width is supported by the CPU and/or memory class.

The address must be classified, e.g. D:0x200, while each symbol has its own implicit memory class.

The possible memory classes depend on the target CPU. The appropriate list can be found in the chapter "*Access Classes*" of the particular **Processor Architecture Manual**.

<value_width>.<access_width>	Description of Data.<value_width>.<access_width>()
Word.Byte	Performs two 8-bit accesses to return a 16-bit value from memory.
Word.Word	Alias for **Data.Word()**.
Long.Byte	Performs four 8-bit accesses to return a 32-bit value from memory.
Long.Word	Performs two 16-bit accesses to return a 32-bit value from memory.
Long.Long	Alias for **Data.Long()**.
Quad.Byte	Performs eight 8-bit accesses to return a 64-bit value from memory.
Quad.Word	Performs four 16-bit accesses to return a 64-bit value from memory.
Quad.Long	Performs two 32-bit accesses to return a 64-bit value from memory.
Quad.Quad	Alias for **Data.Quad()**.

Parameter Type: Address.

Return Value Type: Hex value.

Example:

```
ECHO Data.Long.Byte(D:0x200)
```

See also commands: **MAP.BUS8**, **MAP.BUS16**, and **MAP.BUS32**

Data.AL.ERRORS() Get number of errors detected by Data.AllocList

Syntax: **Data.AL.ERRORS()**

Returns the number of errors detected by the **Data.AllocList** command.

Return Value Type: Decimal value.

Data.Float() Get floating point number

Syntax: **Data.Float("***<format>***",***<address>***)**

Reads a floating point number from memory.

Parameter and Description:

<format>	**Parameter Type**: String.
<address>	**Parameter Type**: Address.

Return Value Type: Float.

Examples:

```
PRINT Data.Float("IEEE",D:0x200)
PRINT Data.Float("IEEE",Var.ADDRESS(f_voltage[0].fVal))
PRINT Data.Float("IEEEDBL",D:0x200)
```

Syntax: **Data.STRing(**<*address*>**)**

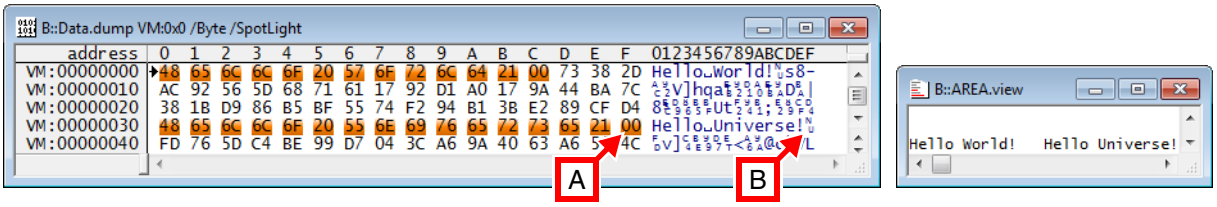
Reads a zero-terminated string from memory. The address must be classified, e.g. D:0x200, while each symbol has its own implicit memory class.

Parameter Type: Address.

Return Value Type: String.

Example:

```
LOCAL &str1 &str2

Data.Set VM:0x0  "Hello World!" 0       ;set two zero-terminated strings
Data.Set VM:0x30 "Hello Universe!" 0    ;to the TRACE32 virtual memory

&str1=Data.STRing(VM:0x0)               ;read the first string
&str2=Data.STRing(VM:0x30)              ;read the second string

PRINT "&str1   &str2"                   ;print both to the message line
AREA.view                               ;display both in the AREA window
```



A In the byte-formatted output, **00** indicates a zero-terminated string.

B In the ASCII-formatted output, **NU** indicates a zero-terminated string.

Data.STRingN() Get zero-terminated string with a maximum length

Syntax: **Data.STRingN(**<*address*>**,**<*length*>**)**

Reads a string from memory. The result is a zero-terminated string. The address must be classified, e.g. D:0x3a0, while each symbol has its own implicit memory class. If the string length is smaller than <*length*>, only <*length*> characters are returned.

Parameter and Description:

<*address*>	**Parameter Type**: Address.
<*length*>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: String.

Example:

```
PRINT Data.STRingN(D:0x3a0,10.)    ;reads 10 characters starting at
                                   ;memory address D:0x3a0
```

Data.SUM() Get checksum

Syntax: **Data.SUM()**

Gets the checksum of the last executed **Data.SUM** command.

Return Value Type: Hex value.

Example:

```
Data.Set P:0x00--0xff %Byte 1
Data.SUM P:0x00--0xff
PRINT Data.SUM()                   ;displays the value 0x100
```

Syntax:	**Data.SWAP.***<value_width>***.***<swap_width>***(***<value>***)**
<value_width>:	**Word** | **Long** | **Quad**
<swap_width>:	**Byte** | **Word** | **Long**
Full function name only required for HELP.Index:	**Data.SWAP.Word.Byte(***<value>***)** **Data.SWAP.Long.Byte(***<value>***)** **Data.SWAP.Long.Word(***<value>***)** **Data.SWAP.Quad.Byte(***<value>***)** **Data.SWAP.Quad.Word(***<value>***)** **Data.SWAP.Quad.Long(***<value>***)**

Swaps byte groups of the size *<swap_width>* in *<value>*. The input *<value>* is truncated or extended to the given *<value_width>*. This can be used to swap a data word between big and little endianness for example.

<value_width>.<swap_width>	**Description of Data.SWAP.<value_width>.<swap_width>()**
Word.Byte	Swaps bytes in a 16-bit word <value>[15:0]. Result: <value>[7:0][15:8]
Long.Byte	Swaps bytes in a 32-bit word <value>[31:0]. Result: <value>[7:0][15:8][23:16][31:24]
Long.Word	Swaps 16-bit words in a 32-bit word <value>[31:0]. Result: <value>[15:0][31:16]
Quad.Byte	Swaps bytes in a 64-bit word <value>[63:0]. Result: <value>[7:0][15:8][23:16][31:24][39:32][47:40][55:48][63:56]
Quad.Word	Swaps 16-bit words in a 64-bit word <value>[63:0]. Result: <value>[15:0][31:16][47:32][63:48]
Quad.Long	Swaps 32-bit words in a 64-bit word <value>[63:0]. Result: <value>[31:0][63:32]

Parameter Type: Hex value.

Return Value Type: Hex value.

Examples:

```
                                      // Result:
PRINT Data.SWAP.Long.Byte(0xFF339922) // 0x229933FF
PRINT Data.SWAP.Long.Word(0xFF339922) // 0x9922FF33
PRINT Data.SWAP.Word.Byte(0xFF339922) // 0x2299              (truncated)
PRINT Data.SWAP.Quad.Long(0x9922)     // 0x0000992200000000 (extended)
```

Data.WSTRING() Get zero-terminated wide string

Syntax: **Data.WSTRING(**<*address*>**)**

Reads a zero-terminated wide string from memory. The address must be classified, e.g. D:0x200, while each symbol has its own implicit memory class. The function extracts the least significant byte of each wide character.

If a BOM is found at the beginning of the string, it is removed from the output. An error is thrown if the BOM does not match the current endianness of the core.

Parameter Type: Address.

Return Value Type: String.

Example:

```
PRINT Data.WSTRING(D:0x200)
```

See also: **Data.STRing()**

Data.WSTRING.BigEndian() Get big-endian wide string

[build 43885 - DVD 02/2013]

Syntax: **Data.WSTRING.BigEndian(**<*address*>**)**

Reads a zero-terminated big-endian wide string from memory. The address must be classified, e.g. D:0x200, while each symbol has its own implicit memory class. The function extracts the least significant byte of each wide character.

If a BOM is found at the beginning of the string, it is removed from the output. An error is thrown if the BOM signals that the string is not big-endian.

Parameter Type: Address.

Return Value Type: String.

Syntax: **Data.WSTRING.LittleEndian(***<address>***)**

Reads a zero-terminated little-endian wide string from memory. The address must be classified, e.g. D:0x200, while each symbol has its own implicit memory class. The function extracts the least significant byte of each wide character.

If a BOM is found at the beginning of the string, it is removed from the output. An error is thrown if the BOM signals that the string is not little-endian.

Parameter Type: Address.

Return Value Type: String.

DEBUGGER.FEATURE() Check debugger feature

[build 140670 - DVD 02/2022]

Syntax:	**DEBUGGER.FEATURE(***<feature>***)**
<feature>:	**INSTRUCTIONSETSIMULATION**

Returns TRUE if the started PowerView executable contains an instruction set simulator.

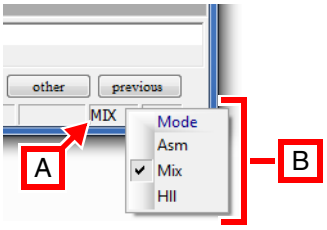
Parameter Type: String.

Return Value Type: Boolean.

DEBUGMODE() Current debug mode

Syntax: **DEBUGMODE()**

Returns the current debug mode.



A Displays the current debug mode in the state line.

B Right-click the current debug mode to change the debug mode manually.

Return Value Type: String.

Return Value and Description:

ASM	Assembler mode. For more information, see **Mode.Asm** command.
MIX	Mixed mode. For more information, see **Mode.Mix** command.
HLL	High-level language mode. For more information, see **Mode.Hll** command.

Example: If the **DEBUGMODE()** function returns a value other than HLL, then the **Mode.Hll** command automatically sets the debug mode to HLL.

```
IF DEBUGMODE()!="HLL"
     Mode.Hll
```

DISASSEMBLE.ADDRESS() Disassembled instruction at address

[build 18929 - DVD 12/2009]

Syntax: **DISASSEMBLE.ADDRESS(***<address>***)**

Returns the disassembled instruction at a given address.

Parameter Type: Address.

Return Value Type: String.

DONGLEID() Serial number of USB WibuKey

Syntax:	**DONGLEID(***<wibukey_index>***)**
<wibukey_index>:	**0** …

Returns the serial number of your USB WibuKey.

Return Value Type: Decimal value.

ELA Function (ARM Coresight Embedded Logic Analyzer)

ELABASE() — ELA base address

[build 77745 - DVD 02/2017]

Syntax: **ELABASE()**

Returns the base address of the ARM Coresight Embedded Logic Analyzer.

Return Value Type: Address.

See also: **ELA** command group.

DPP Function (C166/ST10 only)

DPP() — Content of DPP register

C166/ST10 only

Syntax: **DPP(***<register>***)**

Returns the content of the selected DPPn register.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

In This Section

See also

❏ EPOC.DATAADDRESS() ❏ EPOC.ENTRYPOINT() ❏ EPOC.TEXTADDRESS()

EPOC.DATAADDRESS() Start address of data area (EPOC debugger)

For EPOC debugger dbg only

Syntax: **EPOC.DATAADDRESS()**

Returns the start address of the data area of the currently active debug task.

Return Value Type: Hex value.

EPOC.ENTRYPOINT() Entry address of debug task

For EPOC debugger dbg only

Syntax: **EPOC.ENTRYPOINT()**

Returns the entry address of the currently active debug task.

Return Value Type: Hex value.

EPOC.TEXTADDRESS() Start address of code area (EPOC debugger)

For EPOC debugger dbg only

Syntax: **EPOC.TEXTADDRESS()**

Returns the start address of the code area of the currently active debug task.

Return Value Type: Hex value.

ERROR.ADDRESS() Address of last occurred memory access error

Syntax: **ERROR.ADDRESS()**

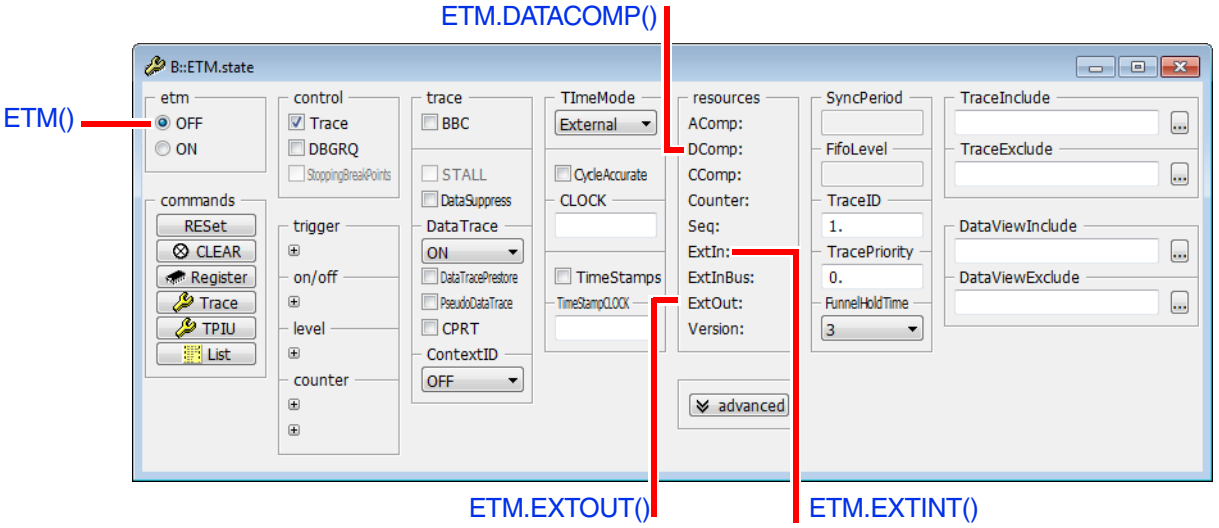
Returns the address of the occurred error from the last executed TRACE32 command.

Return Value Type: Address

```
ON.ERROR GOSUB error_handler

<Data.Set commands>

ON.ERROR nothing
ENDDO

error_handler:
  LOCAL &addr &cont
  &addr=ERROR.ADDRESS()
  DIALOG.YESNO "Error occurred accessing address &addr, continue?"
  ENTRY &cont
  IF &cont==FALSE()
   ENDDO
RETURN
```

ETM Functions

This figure provides an overview of the return values of some of the functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- ■ ETM
- ❑ ETM()
- ❑ ETM.ADDRCOMP()
- ❑ ETM.ADDRCOMPTOTAL()
- ❑ ETM.COUNTERS()
- ❑ ETM.DATACOMP()
- ❑ ETM.EXTIN()
- ❑ ETM.EXTOUT()
- ❑ ETM.FIFOFULL()
- ❑ ETM.MAP()
- ❑ ETM.PROTOCOL()
- ❑ ETM.SEQUENCER()
- ❑ ETM.TraceCore()

ETM() — TRUE if ETM trace is available

[Go to figure]

Syntax: **ETM()**

Returns TRUE if ETM trace is available.

Return Value Type: Boolean.

ETM.ADDRCOMP() For internal usage only

Syntax: **ETM.ADDRCOMP()**

Returns the index number of the last ETM address comparator assigned (for internal usage only).

Return Value Type: Decimal value.

ETM.ADDRCOMPTOTAL() Number of ETM address comparator pair

Syntax: **ETM.ADDRCOMPTOTAL()**

Returns the number of ETM address comparator pairs available.

Return Value Type: Decimal value.

ETM.COUNTERS() Number of ETM counters

Syntax: **ETM.COUNTERS()**

Returns the number of ETM counters available.

Return Value Type: Decimal value.

ETM.DATACOMP() Number of ETM data comparators

[Go to figure]

Syntax: **ETM.DATACOMP()**

Returns the number of ETM data comparators available.

Return Value Type: Decimal value.

ETM.EXTIN() Number of internal ETM inputs

[Go to figure]

Syntax: **ETM.EXTIN()**

Returns the number of external ETM inputs.

Return Value Type: Decimal value.

ETM.EXTOUT() Number of external ETM outputs

[Go to figure]

Syntax: **ETM.EXTOUT()**

Returns the number of external ETM outputs.

Return Value Type: Decimal value.

ETM.FIFOFULL() ETM fifofull logic

Syntax: **ETM.FIFOFULL()**

Returns 1 if ETM fifofull logic available.

Return Value Type: Decimal value.

ETM.MAP() Number of ETM memory map decoders

Syntax: **ETM.MAP()**

Returns the number of ETM memory map decoders.

Return Value Type: Decimal value.

ETM.PROTOCOL() Version of ETM protocol

Syntax: **ETM.PROTOCOL()**

Returns the ETM protocol version.

Return Value Type: Decimal value.

ETM.SEQUENCER() Number of ETM sequencers

Syntax: **ETM.SEQUENCER()**

Returns the number of ETM sequencers available.

Return Value Type: Decimal value.

ETM.TraceCore() TRUE if the core is traced

Arm [build 127768 - DVD 02/2021]

Syntax: **ETM.TraceCore(***<n>***)**

Allows to read back the configuration set by **ETM.TraceCORE**. Returns TRUE if the specified core is traced.

Parameter Type: Decimal value.

Return Value Type: Boolean.

EXTENDED Function (Z80 only)

EXTENDED() — TRUE if register CBAR > 0

Syntax: **EXTENDED()**

Returns TRUE if the register CBAR is > 0 (only Z80).

Return Value Type: Boolean.

FDX.INSTRING() Content at FDX memory address

Syntax: **FDX.INSTRING(**<*address*>**)**

Returns the content at the given FDX memory address.

Parameter Type: Address.

Return Value Type: String.

FDX.TargetSTALLS() Monitor FDX communication stalls on the target

[build 147494 - DVD 09/2022]

Syntax: **FDX.TargetSTALLS()**

Returns TRUE when one of the FDX channels has been signalled as a stall of the FIFO used to transfer data from the target to the host since the last initialization.

Return Value Type: Boolean.

FLAG Functions

In This Section

See also

❏ FLAG() ❏ FLAG.READ() ❏ FLAG.WRITE()

FLAG() TRUE if hardware flag system available

Syntax: **FLAG()**

Returns TRUE if hardware flag system is available.

Return Value Type: Boolean.

FLAG.READ() FLAG memory bytes with read access bit

Syntax: **FLAG.READ(**<*address_range*>**)**

Returns the number of bytes with set Read access bit of the Flag memory.

Parameter Type: Address range.

Return Value Type: Decimal value.

FLAG.WRITE() FLAG memory bytes with write access bit

Syntax: **FLAG.WRITE(**<*address_range*>**)**

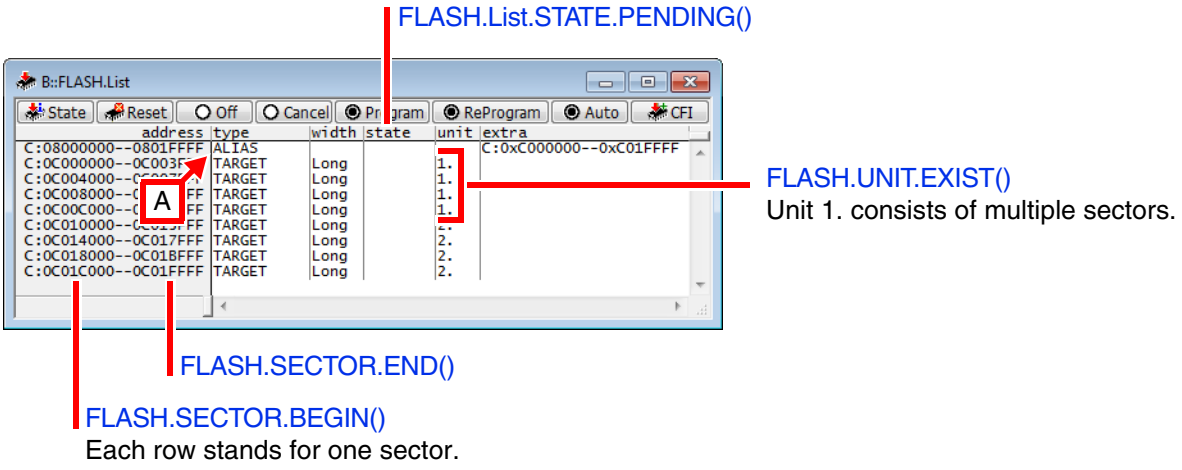
Returns the number of bytes with set Write access bit of the Flag memory.

Parameter Type: Address range.

Return Value Type: Decimal value.

FLASH Functions

This figure provides an overview of the return values of some of the functions. For descriptions of the illustrated functions and the functions not shown here, see below.



A Alias range.

NOTE:	All **FLASH.SECTOR.*()** functions do not take any ALIAS range into account, but observe only declared sectors. For an example of an ALIAS range, see [**A**] in the figure above.

In This Section

See also

- FLASH
- FLASH.CFI.WIDTH()
- FLASH.ID()
- FLASH.List.TYPE()
- FLASH.ProgramMODE.OPTION()
- FLASH.SECTOR.END()
- FLASH.SECTOR.EXTRAvalue()
- FLASH.SECTOR.OPTION()
- FLASH.SECTOR.RANGE()
- FLASH.SECTOR.STATE()
- FLASH.SECTOR.WIDTH()
- FLASH.TARGET.CODERANGE()
- FLASH.TARGET.FILE()
- FLASH.TARGET2.DATARANGE()
- FLASH.UNIT()
- FLASH.UNIT.END()
- FLASH.UNIT.NEXT()
- FLASH.CFI.SIZE()
- FLASH.CLocK.Frequency()
- FLASH.List.STATE.PENDING()
- FLASH.ProgramMODE()
- FLASH.SECTOR.BEGIN()
- FLASH.SECTOR.EXIST()
- FLASH.SECTOR.NEXT()
- FLASH.SECTOR.OTP()
- FLASH.SECTOR.SIZE()
- FLASH.SECTOR.TYPE()
- FLASH.TARGET.BUILD()
- FLASH.TARGET.DATARANGE()
- FLASH.TARGET2.CODERANGE()
- FLASH.TARGET2.FILE()
- FLASH.UNIT.BEGIN()
- FLASH.UNIT.EXIST()

FLASH.CFI.SIZE() Size of FLASH devices

[build 13759]

Syntax: **FLASH.CFI.SIZE(***<address>***,***<bus_width>***)**

Returns the size of single or parallel CFI-conform FLASH devices.

Parameter and Description:

<address>	**Parameter Type**: Address.
<bus_width>	**Parameter Type**: String.

Return Value Type: Hex value. Returns 0 if TRACE32 cannot read the CFI information.

Example:

```
PRINT FLASH.CFI.SIZE(P:0x0,Word)
PRINT FLASH.CFI.SIZE(D:0x40000000,Long)
```

FLASH.CFI.WIDTH() Data bus width of FLASH devices

[build 13759]

Syntax: **FLASH.CFI.WIDTH(***<address>***)**

Returns the data bus width of single or parallel CFI-conform FLASH devices.

Parameter Type: Address.

Return Value Type: String. Returns an empty string if TRACE32 cannot read the CFI information.

FLASH.CLocK.Frequency() FLASH clock value

[build 32807 - DVD 06/2011]

Syntax: **FLASH.CLocK.Frequency()**

Returns the FLASH clock value configured by the **FLASH.CLocK** command. In case of the **FLASH.CLocK AUTO** command, the function returns the frequency of the last time-measurement.

Return Value Type: Decimal value.

Syntax: **FLASH.ID(***<id_type>***)**

<id_type>: **MANID** | **MANIDBANK** | **DEVID** | **DEVID2** | **DEVID3**

After the **FLASH.GETID** command has been executed, you can use the **FLASH.ID()** function to return individual values from the output of the **FLASH.GETID** command, such as the manufacturer ID or device ID of a FLASH device.

Parameter Type: String.

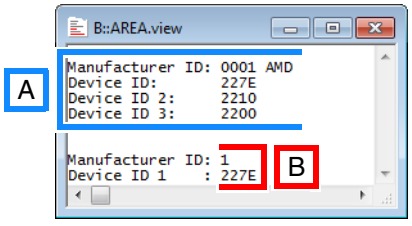
Parameter and Description:

MANID	Manufacturer ID
MANIDBANK	Manufacturer ID bank
DEVID, **DEVID2**, **DEVID3**	Device ID 1, 2, or 3

Return Value Type: Hex value. Returns 0 if (a) the requested ID does not exist or (b) the **FLASH.GETID** command has not yet been executed or (c) the **FLASH.GETID** command has encountered an error.

Example:

```
FLASH.GETID 0x0 Word    ;print the FLASH manufacturer ID and the device
                        ;IDs to the message area A000
AREA.view               ;optional step: open an AREA window to view the
                        ;output of the FLASH.GETID command

RePeaT 2.               ;optional step: insert two empty lines
   PRINT ""

;you can now use the FLASH.ID() function to return individual values:
PRINT "Manufacturer ID: " FLASH.ID(MANID)
PRINT "Device ID 1    : " FLASH.ID(DEVID)
```



A Output of the **FLASH.GETID** command

B Individual values returned by the **FLASH.ID()** function from the output of the **FLASH.GETID** command.

FLASH.List.STATE.PENDING() Number of pending sectors

[build 27943 - DVD 06/2011] [Go to figure]

Syntax: **FLASH.List.STATE.PENDING()**

Returns the number of pending sectors in the **FLASH.List** window.

Return Value Type: Decimal value.

Example: The function **FLASH.List.STATE.PENDING()** returns the number of modified sectors when **AUTO** or **ReProgram** mode is active.

```
FLASH.ReProgram.ALL
Data.LOAD.auto *
PRINT "Modified sectors: " FLASH.List.STATE.PENDING()
FLASH.ReProgram.off
```

FLASH.List.TYPE() FLASH family code of FLASH list entry

[build 59355]

Syntax: **FLASH.List.TYPE(**<*address*>**)**

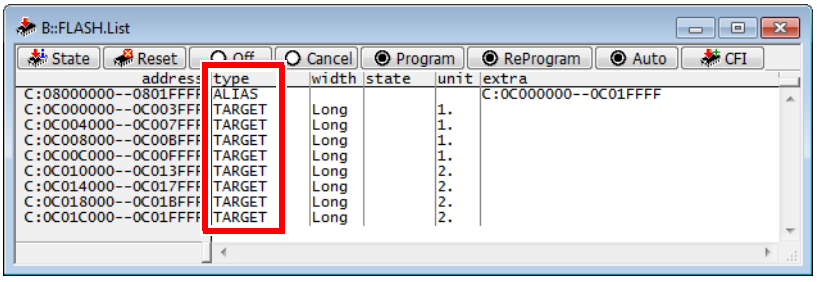
Returns the FLASH family code of a FLASH list entry. The code is displayed in the **type** column of the **FLASH.List** window.

Parameter Type: String.

Return Value Type: Address.

Example:

```
PRINT FLASH.List.TYPE(C:0x08000001) ;returns ALIAS

PRINT FLASH.List.TYPE(C:0x0C004003) ;returns TARGET
```



Syntax: **FLASH.ProgramMODE()**

Returns the active FLASH programming mode.

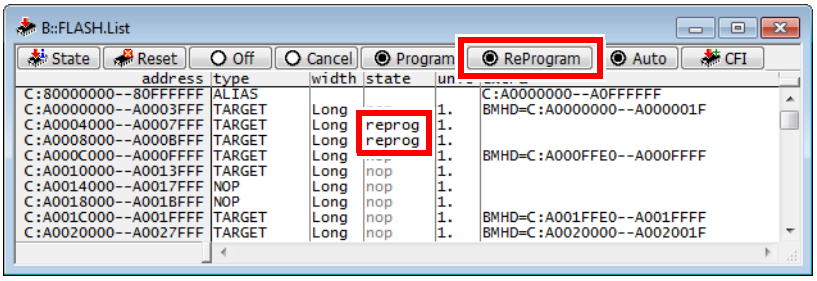
Return Value Type: String. An empty string is returned if none of the FLASH programming modes listed in the table below is active.

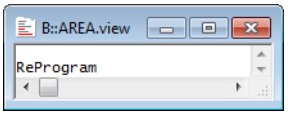
Return Value and Description:

AUTO	The auto programming mode has been activated with **FLASH.AUTO**.
Program	The FLASH programming mode has been activated with **FLASH.Program**.
ReProgram	The FLASH reprogramming mode has been activated with **FLASH.ReProgram**.

Example:

```
FLASH.List

;activate the FLASH reprogramming mode
FLASH.ReProgram C:0xA0004000--0xA000BFFF ;the active mode is shown in
                                         ;the 'state' column of the
                                         ;FLASH.List window

PRINT FLASH.ProgramMODE()                ;returns the active mode
                                         ;'ReProgram'
```





Syntax: **FLASH.ProgramMODE.OPTION()**

Returns the active FLASH programming option.

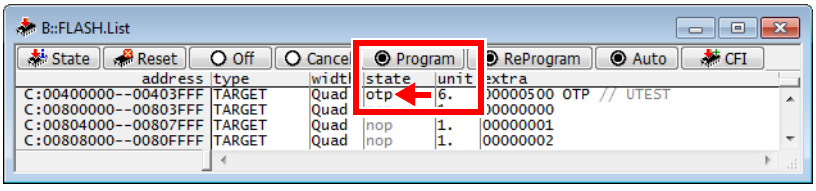
Return Value Type: String. An empty string is returned if none of the options listed in the table below is active.

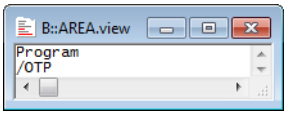
Return Value and Description:

/OTP	FLASH programming for the OTP sector is active. See also **FLASH.Program ... /OTP**.
/CENSORSHIP	Auto programming for the FLASH security bytes is active. See also **FLASH.AUTO ... /CENSORSHIP**.

Example:

```
FLASH.Program 6. /OTP              ;activate the FLASH programming mode
                                   ;for the OTP sector

FLASH.List                         ;the active option OTP is shown in the
                                   ;'state' col. of the FLASH.List window

PRINT FLASH.ProgramMODE()          ;returns the active mode 'Program'

PRINT FLASH.ProgramMODE.OPTION()   ;returns the active option '/OTP'
```





FLASH.SECTOR.BEGIN() Start address

[build 32807 - DVD 02/2012] [Go to figure]

Syntax: **FLASH.SECTOR.BEGIN(**<*address*>**)**

Returns the start address of the sector. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: Address.

FLASH.SECTOR.END() End address

[build 32807 - DVD 02/2012] [Go to figure]

Syntax: **FLASH.SECTOR.END(**<*address*>**)**

Returns the end address of the sector. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: Address.

FLASH.SECTOR.EXIST() TRUE if sector exists

[build 32807 - DVD 02/2012]

Syntax: **FLASH.SECTOR.EXIST(**<*address*>**)**

Returns TRUE if the sector exists. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: Boolean.

Syntax: **FLASH.SECTOR.EXTRAvalue(***<address>***)**

Returns the extra value created with the command **FLASH.Create**. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

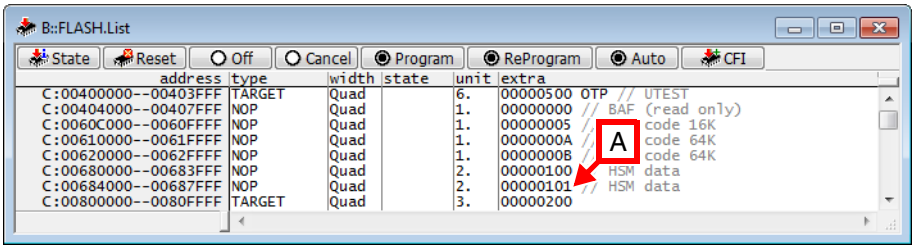
The extra value and other information is displayed in the **extra** column of the **FLASH.List** window. The function **FLASH.SECTOR.EXTRAvalue()**, however, returns only the extra value.

Parameter Type: Address.

Return Value Type: Hex value.

Example:

```
;                                                  <extra_value>
;                                                        |
FLASH.Create 2. 0x00684000--0x00687FFF NOP  Quad 0x0101 /INFO "HSM data"

FLASH.List

;returns 101, i.e. the <extra_value>, without leading zeros
PRINT FLASH.SECTOR.EXTRAvalue(C:0x00684000)
```



A The extra value, here 101, is displayed with leading zeros in the **FLASH.List** window.

FLASH.SECTOR.NEXT() Address of next sector

[build 32807 - DVD 02/2012]

Syntax: **FLASH.SECTOR.NEXT(**<*address*>**)**

Returns the address of the next sector or address 0x0. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: Address.

FLASH.SECTOR.OTP() TRUE if OTP sector

[build 49133]

Syntax: **FLASH.SECTOR.OTP(**<*address*>**)**

Returns TRUE if the sector is declared as OTP. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: Boolean.

Syntax: **FLASH.SECTOR.OPTION(***<address>*,*<option>* | **ALL)**

Returns the options of a FLASH sector in the same syntax as used by the **FLASH.Create** command. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter and Description:

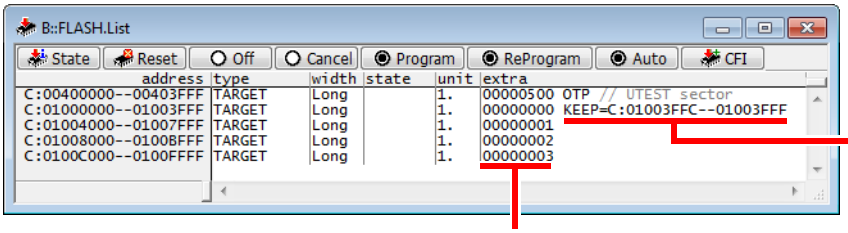
<address>	**Parameter Type**: Address. Any address of a sector. The address has to be classified, e.g. **C**:0x1000000.
<option>	**Parameter Type**: String. You can pass any *<option>* of the **FLASH.Create** command, provided the result of the option is displayed in the **FLASH.List** window in the **extra** column. See screenshot below. An error message is displayed if an invalid option name is used.
ALL	**Parameter Type**: String. Returns all options of a sector that are displayed in the **extra** column. See screenshot below.

NOTE:	To return the extra value of a sector, use **FLASH.SECTOR.EXTRAvalue()**.

Return Value Type: String. An empty string is returned if the option is not used for a sector.

Example:

```
FLASH.Create 0x1000000--0x100FFFF 0x4000 TARGET Long 0x0   \
                                      /AutoInc /KEEP 0x1003FFC--0x1003FFF
FLASH.Create 0x0400000--0x0403FFF 0x4000 TARGET Long 0x500 \
                                                 /OTP /INFO "UTEST sector"
FLASH.List

PRINT FLASH.SECTOR.OPTION(C:0x1000000,KEEP)
                                      ;returns: /KEEP 0x1003FFC--0x1003FFF

PRINT FLASH.SECTOR.OPTION(C:0x0400000,ALL)
                                      ;returns: /OTP /INFO "UTEST sector"
```



Examples of *<options>* displayed in the **extra** column.
FLASH.SECTOR.OPTION()

FLASH.SECTOR.EXTRAvalue().
The option **AutoInc** of FLASH.Create affects the extra value.

FLASH.SECTOR.RANGE() Address range of a FLASH sector

Syntax: **FLASH.SECTOR.RANGE(***<address>***)**

Returns the address range of a sector. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: Address range.

FLASH.SECTOR.SIZE() Size in bytes

Syntax: **FLASH.SECTOR.SIZE(***<address>***)**

Returns the size of the flash sector in bytes. Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: Hex value.

FLASH.SECTOR.STATE() FLASH programming state

Syntax: **FLASH.SECTOR.STATE(***<address>***)**

Returns the FLASH programming state of a sector. The state is displayed in the **state** column of the **FLASH.List** window.

Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: String. An empty string is returned if the specified address is not inside any sector or if the **state** column is empty.

Syntax: **FLASH.SECTOR.TYPE(**<*address*>**)**

Returns the FLASH family code of a sector. The code is displayed in the **type** column of the **FLASH.List** window.

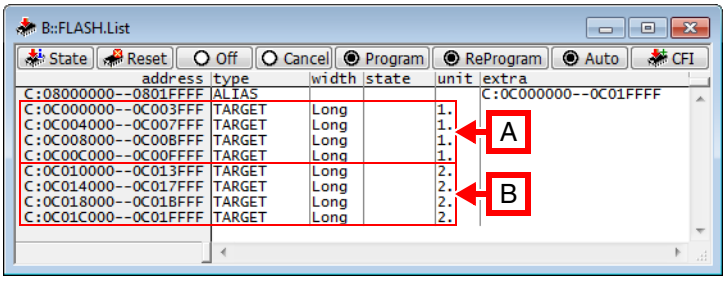
Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: String. An empty string is returned if the specified address is not inside any sector.

Example:

```
PRINT FLASH.SECTOR.TYPE(C:0x08000001) ;returns an empty string because
                                      ;the address is neither in the
                                      ;sectors of unit 1. nor unit 2.

PRINT FLASH.SECTOR.TYPE(C:0x0C004003) ;returns TARGET
```



A Sectors of unit 1. **B** Sectors of unit 2.

FLASH.SECTOR.WIDTH() Width of FLASH sector

[build 50663]

Syntax: **FLASH.SECTOR.WIDTH(***<address>***)**

Returns the width of a FLASH sector. The information is displayed in the **width** column of the **FLASH.List** window.

Please read the **NOTE** regarding the **FLASH.SECTOR.*()** functions.

Parameter Type: Address.

Return Value Type: String. Returns an empty string if the address is not inside any sector.

FLASH.TARGET.BUILD() Build number of FLASH algorithm file

[build 33191 - DVD 02/2012]

Syntax: **FLASH.TARGET.BUILD(***<file>***)**

Returns the build number of the FLASH algorithm binary file or 0 if the file has no build number. The build number of the FLASH algorithm binary is independent of the TRACE32 build number.

Parameter Type: String.

Return Value Type: Decimal value.

FLASH.TARGET.CODERANGE() Code range of FLASH algorithm

[build 89348 - DVD 02/2018]

Syntax: **FLASH.TARGET.CODERANGE()**

FLASH.TARGET2.CODERANGE()

Returns the address range where the code of the FLASH algorithm binary file and the software breakpoint is located. The address range is declared with the **FLASH.TARGET** or **FLASH.TARGET2** command.

Return Value Type: Address range.

FLASH.TARGET.DATARANGE() Data range of FLASH algorithm

[build 89348 - DVD 02/2018]

Syntax: **FLASH.TARGET.DATARANGE()**

FLASH.TARGET2.DATARANGE()

Returns the address range where the argument buffer, the data buffer, and the stack used by the FLASH algorithm binary file are located. The address range is declared with the **FLASH.TARGET** or **FLASH.TARGET2** command.

Return Value Type: Address range.

FLASH.TARGET.FILE() Name of FLASH algorithm file

Syntax: **FLASH.TARGET.FILE()**
[build 48531]

FLASH.TARGET2.FILE()
[build 67392 - DVD 02/2016]

Returns the file name of the FLASH algorithm binary declared with the **FLASH.TARGET** or **FLASH.TARGET2** command.

Return Value Type: String.

FLASH.UNIT() Unit number of FLASH sector

[build 41711]

Syntax: **FLASH.UNIT(**<*address*>**)**

Returns the unit number of a FLASH sector. Returns 0 if the sector does not exist.

Parameter Type: Address.

Return Value Type: Decimal value.

FLASH.UNIT.BEGIN() Unit start address

[build 41711]

Syntax: **FLASH.UNIT.BEGIN(**<*unit*>**)**

Returns the start address of the unit or displays the error #emu_flashnfnd if the unit does not exist.

Parameter Type: Decimal value.

Return Value Type: Address.

FLASH.UNIT.END() Unit end address

[build 41711]

Syntax: **FLASH.UNIT.END(**<*unit*>**)**

Returns the end address of the unit or displays the error #emu_flashnfnd if the unit does not exist.

Parameter Type: Decimal value.

Return Value Type: Address.

FLASH.UNIT.EXIST() TRUE if unit exists

[build 41711] [Go to figure]

Syntax: **FLASH.UNIT.EXIST(**<*unit*>**)**

Returns TRUE if the unit exists, otherwise FALSE.

Parameter Type: Decimal value.

Return Value Type: Boolean.

FLASH.UNIT.NEXT() Number of next unit

[build 41711]

Syntax: **FLASH.UNIT.NEXT(**<*unit*>**)**

Returns the number of the next unit. Returns 0 if there is no next unit.

Parameter Type: Decimal value.

Return Value Type: Decimal value.

In This Section

See also

- FLASHFILE
- FLASHFILE.GETBADBLOCK.NEXT()
- FLASHFILE.GETBADBLOCK.COUNT()
- FLASHFILE.SPAREADDRESS()

FLASHFILE.GETBADBLOCK.COUNT() Number of bad blocks

Syntax: **FLASHFILE.GETBADBLOCK.COUNT()**

After the **FLASHFILE.GETBADBLOCK** command has been executed, this function returns the number of the bad blocks on the NAND flash memory.

Return Value Type: Hex value.

Example:

```
FLASHFILE.GETBADBLOCK 0x0--0xFFFFF      ; test range 0x0--0xFFFFF for bad
                                        ; blocks

&badblockcount=FLASHFILE.GETBADBLOCK.COUNT()

PRINT "Number of bad blocks: " &badblockcount
```

FLASHFILE.GETBADBLOCK.NEXT() Address of bad block

[Example]

Syntax: **FLASHFILE.GETBADBLOCK.NEXT()**

After the **FLASHFILE.GETBADBLOCK** command has been executed, this function returns the addresses of the bad blocks on the NAND flash memory.

Return Value Type: Hex value.

Example:

```
FLASHFILE.GETBADBLOCK 0x0--0xFFFFF      ; test range 0x0--0xFFFFF for bad
                                        ; blocks

&badblockcount=FLASHFILE.GETBADBLOCK.COUNT()
&loopNum=&badblockcount

WHILE (&loopNum>0)
(
   PRINT %Hex "Bad block address: 0x" FLASHFILE.GETBADBLOCK.NEXT()
   &loopNum=&loopNum-1.
)
```

FLASHFILE.SPAREADDRESS() Address of spare area

Syntax: **FLASHFILE.SPAREADDRESS(***<address>***)**

Returns the address of a NAND flash spare area (SP) based on the specified main area address. *<address>* is the address of the main area.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

Example: The page size of our example NAND flash main / spare area is 2048 byte / 64 byte, and each row stands for one page.

address	**Main Area**	**SP**	address
0x0			0x0
0x800			0x40
0x1000			0x80

```
;                          <main_area_address>
PRINT FLASHFILE.SPAREADDRESS(0x800)        ;returns the corresponding
                                           ;spare area address, here:
                                           ;0x40
```

FPU Functions (Floating Point Unit)

In This Section

See also

■ FPU ❑ FPU() ❑ FPU.RAW() ❑ FPUCR()

FPU() FPU register contents

Syntax: **FPU(**<*name*>**)**

Returns the FPU register contents.

Parameter Type: String.

Return Value Type: Float.

FPUCR() FPU control register contents

Syntax: **FPUCR(**<*name*>**)**

Returns the FPU control register contents.

Parameter Type: String.

Return Value Type: Hex value.

FPU.RAW() FPU register raw contents

AndeStar, ARM, C2000, ColdFire, MIPS32, MMDSP, PowerPC, SH

Syntax: **FPU.RAW(**<*name*>**)**

Returns the hexadecimal raw content of the given FPU control register.

Parameter Type: String.

Return Value Type: Hex value.

FXU Function

FXU() Content of FXU register

RH850 [build 81540 - DVD 09/2017]

Syntax: **FXU(**<*register_name*>**)**

Returns the content of the selected FXU register.

Parameter Type: String.

Return Value Type: String.

See also: **FXU** command group.

GROUP Function

GROUP.EXIST() TRUE if group exists

Syntax: **GROUP.EXIST(**<*group_name*>**)**

Returns TRUE if a group name already exists, FALSE otherwise.

Parameter Type: String.

Return Value Type: Boolean.

See also: **GROUP** command group.

Hardware Functions

In This Section

The hardware function allows to check which TRACE32 hardware is plugged in.

See also

- hardware.COMBIPROBE()
- hardware.ICD()
- hardware.POWERINTEGRATOR2()
- hardware.POWERPROBE()
- hardware.POWERTRACE2()
- hardware.POWERTRACE3()
- hardware.POWERTRACESERIAL()
- hardware.UTRACE()
- hardware.ESI()
- hardware.POWERINTEGRATOR()
- hardware.POWERNEXUS()
- hardware.POWERTRACE()
- hardware.POWERTRACE2LITE()
- hardware.POWERTRACEPX()
- hardware.QUADPROBE()

hardware.COMBIPROBE() TRUE if CombiProbe

Syntax: **hardware.COMBIPROBE()**

Returns TRUE if a COMBIPROBE hardware is connected.

Return Value Type: Boolean.

hardware.ESI() TRUE if EPROM Simulator

Syntax: **hardware.ESI()**
ESI() (deprecated)

Returns TRUE if the debugger is running via the ESI (EPROM Simulator) hardware.

Return Value Type: Boolean.

hardware.ICD() TRUE if TRACE32 debug hardware

Syntax: **hardware.ICD()**
BDM() (deprecated)
DEBUGger() (deprecated)
ICD() (deprecated)

Returns TRUE if TRACE32 PowerView communicates with the target via a TRACE32 debug hardware. Unlike the **hardware.POWERDEBUG()** function, this also includes legacy debug modules.

Return Value Type: Boolean.

hardware.POWERDEBUG() TRUE if TRACE32 PowerDebug hardware

Syntax: **hardware.POWERDEBUG()**
POWERDEBUG() (deprecated)

Returns TRUE if TRACE32 PowerView communicates with the target via a µTrace (MicroTrace), POWER DEBUG or POWERTRACE / ETHERNET hardware module.

Return Value Type: Boolean.

hardware.POWERINTEGRATOR() TRUE if a PowerIntergrator

Syntax: **hardware.POWERINTEGRATOR()**
POWERINTEGRATOR() (deprecated)
INTEGRATOR() (deprecated)

Returns TRUE if a POWER INTEGRATOR hardware module is connected.

Return Value Type: Boolean.

hardware.POWERINTEGRATOR2() TRUE if a PowerIntegrator II

Syntax: **hardware.POWERINTEGRATOR2()**
POWERINTEGRATOR2() (deprecated)

Returns TRUE if a POWERINTEGRATOR II hardware module is connected.

Return Value Type: Boolean.

hardware.POWERNEXUS() TRUE is a NEXUS Adapter

Syntax: **hardware.POWERNEXUS()**
POWERNEXUS() (deprecated)

Returns TRUE if a NEXUS ADAPTER is connected.

Return Value Type: Boolean.

hardware.POWERPROBE() TRUE is a PowerProbe

Syntax: **hardware.POWERPROBE()**
POWERPROBE() (deprecated)
PROBE() (deprecated)

Returns TRUE if a POWERPROBE hardware module is connected.

Return Value Type: Boolean.

hardware.POWERTRACE() TRUE if a PowerTrace Module

Syntax: **hardware.POWERTRACE()**
POWERTRACE() (deprecated)

Returns TRUE if a POWERTRACE module is connected.

Additional prerequisite is that a PREPROCESSOR / NEXUS ADAPTER is also plugged in. This is not required for POWER TRACE SERIAL.

Return Value Type: Boolean.

hardware.POWERTRACE2() TRUE if a PowerTrace II

[build 14028 - DVD 10/2008]

Syntax: **hardware.POWERTRACE2()**
POWERTRACE2() (deprecated)

Returns TRUE if a POWER TRACE II or POWER TRACE II LITE hardware module is connected. Additional prerequisite is that a PREPROCESSOR / NEXUS ADAPTER is also plugged in.

Return Value Type: Boolean.

hardware.POWERTRACE2LITE() TRUE if a PowerTrace II LITE

[build 117875 - DVD 09/2020]

Syntax: **hardware.POWERTRACE2LITE()**

Returns TRUE if a POWER TRACE II LITE hardware module is connected. Additional prerequisite is that a PREPROCESSOR / NEXUS ADAPTER is also plugged in.

Return Value Type: Boolean.

hardware.POWERTRACE3() TRUE if a PowerTrace III

[build 124115 - DVD 09/2021]

Syntax: **hardware.POWERTRACE3()**

Returns TRUE if a POWER TRACE III hardware module is connected. Additional prerequisite is that a PREPROCESSOR / NEXUS ADAPTER is also plugged in.

Return Value Type: Boolean.

hardware.POWERTRACEPX() TRUE if a PowerTrace PX

[build 79974 - DVD 02/2017]

Syntax: **hardware.POWERTRACEPX()**

Returns TRUE if a POWER TRACE PX hardware module is connected. Additional prerequisite is that a PREPROCESSOR / NEXUS ADAPTER is also plugged in.

Return Value Type: Boolean.

hardware.POWERTRACESERIAL() TRUE if a PowerTrace Serial

[build 78458 - DVD 02/2017]

Syntax: **hardware.POWERTRACESERIAL()**

Returns TRUE if a POWER TRACE SERIAL hardware module is connected.

Return Value Type: Boolean.

hardware.POWERTRACESERIAL2() TRUE if a PowerTrace Serial II

[build 158903 - DVD 03/2023]

Syntax: **hardware.POWERTRACESERIAL2()**

Returns TRUE if a POWER TRACE SERIAL II hardware module is connected.

Return Value Type: Boolean.

hardware.QUADPROBE() TRUE if QuadProbe

[build 63955 - DVD 09/2016]

Syntax: **hardware.QUADPROBE()**

Returns TRUE if a QUADPROBE hardware is plugged.

Return Value Type: Boolean.

hardware.UTRACE() TRUE if µTrace

[build 46956 - DVD 2013/08]

Syntax: **hardware.UTRACE()**

Returns TRUE if TRACE32 PowerView communicates with the target via a µTrace (MicroTrace),

Return Value Type: Boolean.

HVX() Content of HVX register

[build 61364 - DVD 09/2015]

Syntax: **HVX(**<*register_name*>**)**

Returns the content of the selected HVX register.

Parameter Type: String.

Return Value Type: String.

See also: **HVX** command group.

I2C Functions

In This Section

See also

■ I2C ❑ I2C.DATA() ❑ I2C.PIN()

I2C.DATA() Data read by I2C.TRANSFER

[build 62704 - DVD 09/2015]

Syntax: **I2C.DATA(**<*index*>**)**
CAnalyzer.I2C.DATA(<*index*>**)** (deprecated)

Returns byte <*index*> read with the command **I2C.TRANSFER**.

Parameter Type: Decimal value.

Return Value Type: Decimal value.

I2C.PIN() Pin status

[build 62704 - DVD 09/2015]

Syntax: **I2C.PIN(**<*pin_name*>**)**

<*pin_name*>: **SCL** | **SDA**

Returns the pin status 0 or 1.

Parameter Type: String.

Return Value Type: Binary value.

See also: **I2C.PIN**

ID Functions

In This Section

See also

- ID.CABLE()
- ID.PREPROcessor()
- ID.SERialPort1()
- ID.WHISKER()
- IDCODE()
- IDCODENUMBER()

ID.CABLE() Hardware ID of debug cable

Syntax: **ID.CABLE()**

Returns the hardware ID of the debug cable (not the serial number).

Return Value Type: Hex value.

ID.PREPROcessor() Hardware ID of preprocessor

[build 12440 - DVD 10/2008]

Syntax: **ID.PREPROcessor()**

HEADID() (deprecated)
[build 07808 - DVD 09/2007]

Returns the hardware ID of the preprocessor (not the serial number).

Return Value Type: Hex value.

To return the serial number, use the function **VERSION.SERIAL.PREPROcessor()**.

Syntax: **ID.SERialPort1()**

Returns a 16-bit ID specifying the type of the Lauterbach device plugged to Serial Port 1 of a PowerTrace Serial.

Return Value Type: Hex value.

Return Value and Description:

0x2082	Aurora 2 Preprocessor
0x208A	PCIe Gen 4 Preprocessor
0x2002	PCIe Gen 3 x8 Slot-Card-Adapter
0x2009	PCIe Gen 3 x4 Slot-Card-Adapter
0x200A	PCIe Gen 3 x1 Slot-Card-Adapter
0x2004	Adapter for AGBT
0x200B	Adapter for AGBT/SGBT via HSTCU
0x2003	Universal Trace Adapter
0x2006	Adapter for RH850-34pin
0x2007	Adapter for RH850-40pin

Syntax: **ID.WHISKER(**<*int*>**)**

Returns the ID of the whisker cable connected to the connector specified by <*int*>.

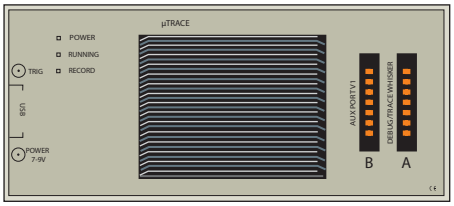
Parameter Type: Decimal value.

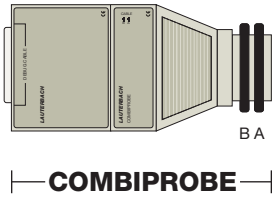
Return Value Type: Hex value.

Use cases for the function **ID.WHISKER(**<*int*>**)**:

- Scripts that use TRACE32 commands that only work if a specific whisker is connected. E.g. using the **ETA** command group requires that a **Conv. CombiProbe/µTrace (MicroTrace) to PI-Analog Probe** plus an **Analog Probe for PI/PT-II/CombiProbe/µTrace (MicroTrace)** is connected.
- A script designed for different tool configurations can use this function to parenthesize commands applicable only for a individual configuration.

µTRACE® FOR CORTEX®-M / USB 3



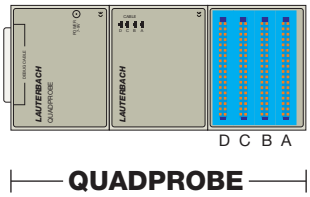


<int> Parameter	
0	For connector A
1	For connector B

Return Values	
0x00	No whisker
0x01	LA-4505 MIPI34 Whisker for CombiProbe/MicroTrace
0x02	LA-4515 DCI OOB Whisker for CombiProbe (Version 1)
0x04	LA-4551 Whisker Cable TriCore DAP for CombiProbe (no longer supported)
0x05	LA-4509 CombiProbe Intel x86/x64 MIPI34 (no series termination).
0x08	LA-4508 Conv. CombiProbe/µTrace (MicroTrace) to PI-Analog Probe plus LA-7949 Analog Probe for PI/PT-II/CombiProbe/µTrace (MicroTrace)
0x09	LA-4515 DCI OOB Whisker for CombiProbe (Version 2)
0x0B	LA-4400 USB-C Breakout Module (Type-C Port)
0x0E	LA-4553 AUTO26 Whisker for CombiProbe or LA-2763 Whisker dsPIC Dual Core for CombiProbe
0x10	LA-4511 Whisker Cable MIPI60-C for CombiProbe or LA-4517 Whisker Cable MIPI60-C for CombiProbe Long
0x11	LA-4571 Whisker Cable MIPI60-Cv2 for CombiProbe (Version 2)
0x12	LA-4513 MIPI20T-HS Whisker for CombiProbe/MicroTrace
0x26	LA-4571 Whisker Cable MIPI60-Cv2 for CombiProbe (Version 2.1)
0x27	LA-2500 Mixed-Signal Probe COB 2/MicroTrace/PT-III
0xFF	Unknown whisker, please update the TRACE32 software.

Not all whiskers are supported on both ports. For whiskers that occupy both ports of the CombiProbe, the ID is returned by ID.WHISKER(0.) and ID.WHISKER(1.) returns 0x00.

The function **hardware.UTRACE()** returns TRUE if a µTrace (MicroTrace) is connected. The function **hardware.COMBIPROBE()** returns TRUE if a CombiProbe or CombiProbe 2 is connected.



<int> Parameter	
0	For connector A
1	For connector B
2	For connector C
3	For connector D

Return Values	
0x32	LA-4611 Whisker MIPI60-Q QuadProbe Intel® x86/x64 connected.
0x00	No whisker connected.
0xff	Unknown whisker, please update the TRACE32 software.

The function **hardware.QUADPROBE()** returns TRUE if a TRACE32 QuadProbe is connected.

[build 17669 - DVD 12/2009]

Syntax: **IDCODE(***<n>***)**

Returns the JTAG ID code of the n-th TAP in your JTAG chain after executing **SYStem.DETECT IDCode**.

Parameter Type: Decimal value.

Return Value Type: Hex value.

Example:

```
SYStem.DETECT IDCode
PRINT "1st device in JTAG daisy chain is 0x" %Hex IDCODE(0)
```

IDCODENUMBER() Number of detected TAPs

[build 17669 - DVD 12/2009]

Syntax: **IDCODENUMBER()**

Returns the number of TAPs detected by **SYStem.DETECT IDCode**.

Return Value Type: Decimal value.

Integrator Functions

In This Section

See also

- Integrator()
- Integrator.ADC.ENABLE()
- Integrator.ADC.SHUNT()
- Integrator.COUNTER.EVENT()
- Integrator.COUNTER.EXTERN()
- Integrator.COUNTER.TIME()
- Integrator.DIALOGDSEL()
- Integrator.DIALOGDSELGET()
- Integrator.DSEL()
- Integrator.FIND.PI_CHANNEL()
- Integrator.FIND.PI_WORD()
- Integrator.FIRST()
- Integrator.FLAG()
- Integrator.GET()
- Integrator.MAXSIZE()
- Integrator.PROBE()
- Integrator.PROGRAMFILENAME()
- Integrator.RECORD.DATA()
- Integrator.RECORD.TIME()
- Integrator.RECORDS()
- Integrator.REF()
- Integrator.SIZE()
- Integrator.STATE()
- Integrator.TRACK.RECORD()
- Integrator.USB()

Integrator() TRUE if PowerIntegrator

Syntax: **Integrator()**

Returns TRUE if a PowerIntegrator is connected.

Return Value Type: Boolean.

Integrator.FIRST() Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **Integrator.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

Integrator.ADC.ENABLE() Bitmask of enabled analog channels

Syntax: **Integrator.ADC.ENABLE(***<channel>***)**

Returns the bitmask of enabled analog channels of the Analog Probe.

Parameter Type: String.

Return Value Type: Boolean.

Integrator.ADC.SHUNT() Shunt-resistor value

Syntax: **Integrator.ADC.SHUNT(***<channel>***)**

Returns the shunt-resistor value of the specified current measurement *<channel>* of the Analog Probe.

Parameter Type: String.

Return Value Type: Float.

Integrator.ANALOG()

Syntax: **Integrator.ANALOG()**

Returns a value different from zero if analog probes are plugged in a PowerIntegrator hardware. For connector A..F, J..O the values are: 0x0001..0x0020, 0x0100..0x2000.

Return Value Type: Hex value.

Integrator.COUNTER.EVENT() Get value of trigger program event counter

Syntax: **Integrator.COUNTER.EVENT(***<counter_name>***)**

Returns the value of an event counter of the PowerIntegrator CTU.

Parameter Type: String.

Return Value Type: Decimal value.

Integrator.COUNTER.EXTERN() Value of trigger program external counter

Syntax: **Integrator.COUNTER.EXTERN(***<counter_name>***)**

Returns the value of an extern counter of the PowerIntegrator CTU.

Parameter Type: String.

Return Value Type: Decimal value.

Integrator.COUNTER.TIME() Get value of trigger program time counter

Syntax: **Integrator.COUNTER.TIME(***<counter_name>***)**

Returns the value of an time counter of the PowerIntegrator CTU.

Parameter Type: String.

Return Value Type: Time value.

Integrator.DIALOGDSEL() For internal usage only

Syntax: **Integrator.DIALOGDSEL(***<string>***)**

For internal usage only.

Parameter Type: String.

Return Value Type: Hex value.

Integrator.DIALOGDSELGET() For internal usage only

Syntax: **Integrator.DIALOGDSELGET()**

For internal usage only.

Return Value Type: String.

Integrator.DSEL() — For internal usage only

Syntax: **Integrator.DSEL()**

For internal usage only.

Return Value Type: String.

Integrator.FIND.PI_CHANNEL() — For internal usage only

Syntax: **Integrator.FIND.PI_CHANNEL(***<signal_name>***)**

Returns if the specified signal-name is defined (internal use only).

Parameter Type: String.

Return Value Type: Hex value.

Integrator.FIND.PI_WORD() — TRUE if signal word is defined

Syntax: **Integrator.FIND.PI_WORD(***<signal_word>***)**

Returns TRUE if the specified signal-word is defined.

Parameter Type: String.

Return Value Type: Boolean.

Integrator.FLAG() — Check state of trigger program FLAG

Syntax: **Integrator.FLAG(***<flag_name>***)**

Returns the value of a flag of the PowerIntegrator CTU.

Parameter Type: String.

Return Value Type: Boolean.

Integrator.GET() Value of channel

Syntax: **Integrator.GET(**<*channel_name*>**)**

Returns the current value of the channel.

Parameter Type: String.

Return Value Type: Hex value.

0	High
1	Low

Integrator.MAXSIZE() Get max. size of trace buffer in records

Syntax: **Integrator.MAXSIZE()**

Returns the maximum size of the PowerIntegrator trace buffer in records (value depends on the actual selected tracing mode too).

Return Value Type: Decimal value.

Integrator.PROBE() For internal usage only

Syntax: **Integrator.PROBE()**

Returns the bitmask of connected probes (internal use only).

Return Value Type: Hex value.

Integrator.PROGRAMFILENAME() File name of trigger program

Syntax: **Integrator.PROGRAMFILENAME()**

Returns the file name of the active trigger program.

Return Value Type: String.

Integrator.RECORD.DATA() Get data recorded in trace record

Syntax: **Integrator.RECORD.DATA(***<record_number>***,***<channel>***)**

Returns the sampled data from the specified record.

Parameter and Description:

<record_number>	**Parameter Type**: Decimal value.
<channel>	**Parameter Type**: String.

Return Value Type: Hex value.

Integrator.RECORD.TIME() Get timestamp of trace record

Syntax: **Integrator.RECORD.TIME(***<record_number>***)**

Returns the timestamp from the specified record. For an example, see **Analyzer.RECORD.TIME()**.

Parameter Type: Decimal value.

Return Value Type: Time value.

Integrator.RECORDS() Get number of used trace records

Syntax: **Integrator.RECORDS()**

Returns the number of records.

Return Value Type: Decimal value.

Integrator.REF() Get record number of reference record

Syntax: **Integrator.REF()**

The command **Integrator.REF** allows to mark a trace record as reference record. The function returns the record number of the reference record.

Return Value Type: Decimal value.

Integrator.SIZE() Get current trace buffer size in records

Syntax: **Integrator.SIZE()**

Returns the actual defined logical size of the Power Integrator trace buffer in records.

Return Value Type: Decimal value.

Integrator.STATE() Get state of the Integrator

Syntax: **Integrator.STATE()**

Returns the state of the Integrator.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state

Integrator.TRACK.RECORD() Get record number matching search

Syntax: **Integrator.TRACK.RECORD()**

After a successful search operation, this function returns the record number. For an example, see **Analyzer.TRACK.RECORD()**.

Return Value Type: Decimal value.

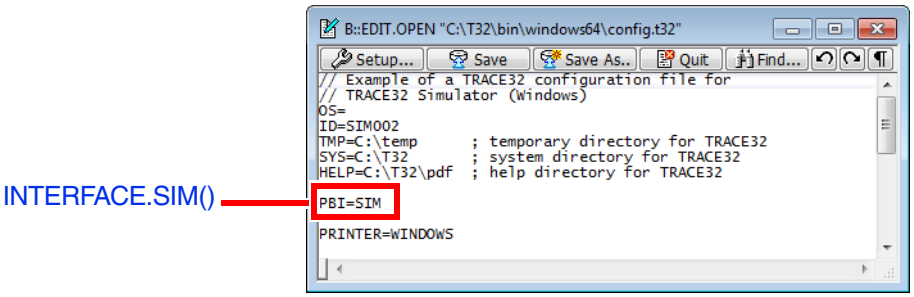
Syntax: **Integrator.USB()**

Returns a value not equal to 0 if an USB2 probe is connected (internal use only).

Return Value Type: Hex value.

INTERFACE Functions

This figure provides an overview of the return values of some of the functions. For descriptions of the illustrated functions and the functions not shown here, see below.



The **INTERFACE** functions return the `PBI=` setting in the configuration file (by default config.t32) irrespective of the current system mode setting. The system mode settings, such as **SYStem.Mode Up** or **SYStem.Mode Down**, are shown in the **SYStem.state** window. They can be returned with the **SYStem.Mode()** function.

To open the configuration file of a TRACE32 instance:

1. Choose **Help** menu > **About TRACE32**.
2. In the **VERSION** window, click the **edit** button.

In This Section

See also

- INTERFACE.CADI()
- INTERFACE.GDB()
- INTERFACE.GDI()
- INTERFACE.HOST()
- interface.HOSTMCI()
- INTERFACE.IRIS()
- INTERFACE.MCD()
- INTERFACE.NAME()
- INTERFACE.QNX()
- INTERFACE.SIM()

INTERFACE.CADI() TRUE if connection to target is via CADI interface

Syntax: **INTERFACE.CADI()**

Returns TRUE if the debugger is connected to the target via the CADI interface (`PBI=CADI`).

Return Value Type: Boolean.

INTERFACE.GDB() TRUE if connection to target is via GDB interface

Syntax: **INTERFACE.GDB()**

Returns TRUE if the debugger is connected to the target via the GDB interface (`PBI=GDB`).

Return Value Type: Boolean.

INTERFACE.GDI() TRUE if connection to target via GDI interface

Syntax: **INTERFACE.GDI()**
GDI() (deprecated)

Returns TRUE if the debugger is connected to the target via the GDI interface (`PBI=GDI`).

Return Value Type: Boolean.

INTERFACE.HOST() TRUE if application is debugged on host

Syntax: **INTERFACE.HOST()**

Returns TRUE if the debugger is connected to the application via the Native Host Debugging interface (`PBI=HOST`).

Return Value Type: Boolean.

interface.HOSTMCI() TRUE if TRACE32 debug driver runs on host

[build 38565 - DVD 08/2012]

Syntax: **interface.HOSTMCI()**

Returns TRUE if the TRACE32 debug driver runs on the host to access the target CPU by native connections or artificial back-end interfaces. The result reflects the settings `PBI=MCILIB` or `PBI=MCISERVER` of the config file.

Return Value Type: Boolean.

INTERFACE.IRIS() TRUE if connection to target is via IRIS interface

Syntax: **INTERFACE.IRIS()**

Returns TRUE if the debugger is connected to the target via the IRIS interface (`PBI=IRIS`).

Return Value Type: Boolean.

INTERFACE.MCD() TRUE if connection to target via MCD interface

Syntax: **INTERFACE.MCD()**

Returns TRUE if the debugger is connected to the target via the MCD interface (`PBI=MCD`).

Return Value Type: Boolean.

INTERFACE.NAME() Name of debugger

[build 76305 - DVD 09/2016]

Syntax: **INTERFACE.NAME()**

Returns the name of the debugger this GUI is connected to. This is the same name as printed in the **VERSION.view** window, e.g. “Power Debug USB 3.0”, “Simulator” or “GDI”.

Return Value Type: String.

INTERFACE.QNX() TRUE if PBI=QNX

Syntax: **INTERFACE.QNX()**

Returns TRUE if TRACE32 is connected to a process-level debugger for QNX (`PBI=QNX`).

Return Value Type: Boolean.

[Go to figure]

Syntax: **INTERFACE.SIM()**
SIMULATOR() (deprecated)

Returns TRUE if the debugger is running on the Lauterbach Instruction Set Simulator (`PBI=SIM`).

Return Value Type: Boolean.

IOBASE Functions

In This Section

See also

❏ IOBASE() ❏ IOBASE.ADDRESS() ❏ IOBASE2()

IOBASE() Base address of internal I/O's

Syntax: **IOBASE()**

Gets the base address of the internal I/O's without access class.

Return Value Type: Hex value.

IOBASE.ADDRESS() Base address of internal I/O's with access class

Syntax: **IOBASE.ADDRESS()**

Gets the base address of the internal I/O's with access class.

Return Value Type: Address.

IOBASE2() Second base address of internal I/O's

Syntax: **IOBASE2()**

Gets the second base address of the internal I/O's without access class.

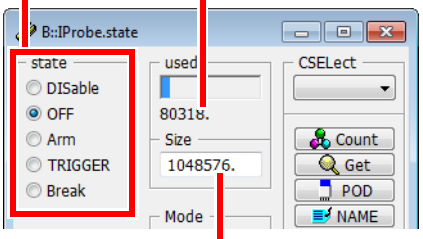
Return Value Type: Hex value.

IProbe Functions

These figures provide an overview of the return values of some of the functions. For descriptions of the illustrated functions and the functions not shown here, see below.

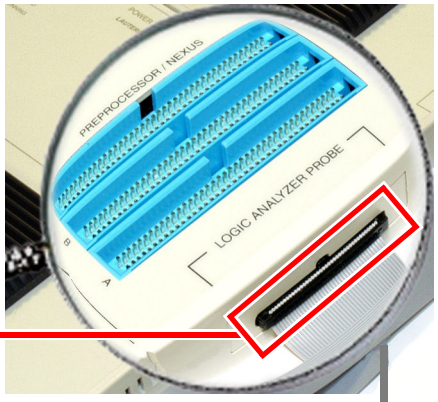
IProbe.STATE()

IProbe.RECORDS()

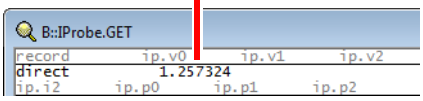


IProbe.SIZE()
IProbe.MAXSIZE()

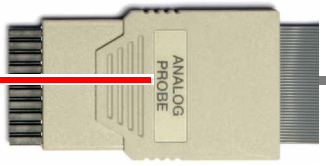
IProbe.PROBE()

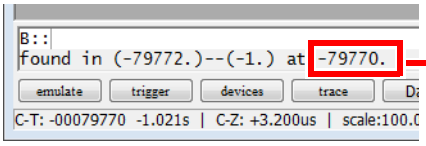


IProbe.GET()



IProbe.ANALOG()

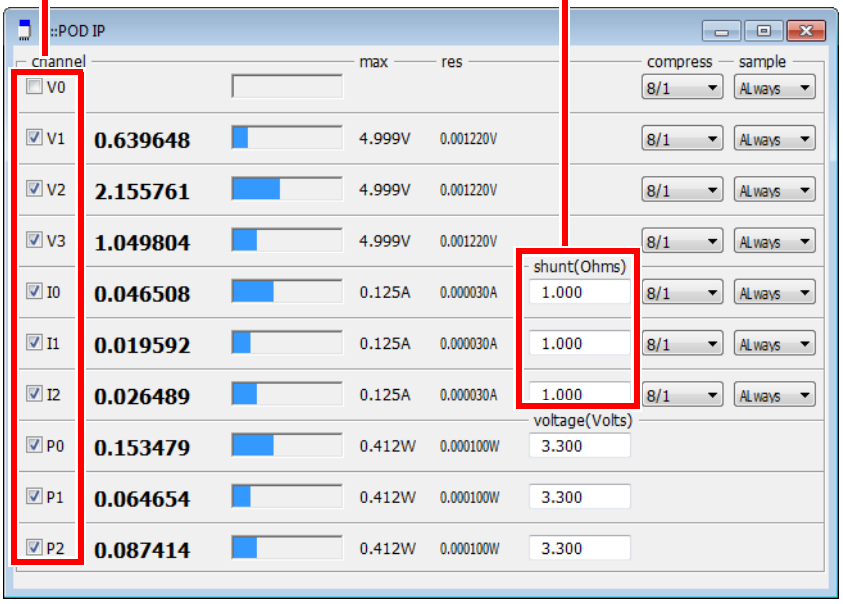




IProbe.TRACK.RECORD()

IProbe.ADC.ENABLE()

IProbe.ADC.SHUNT()



In This Section

See also

- IProbe
- IProbe()
- IProbe.ADC.ENABLE()
- IProbe.ADC.SHUNT()
- IProbe.ANALOG()
- IProbe.FIRST()
- IProbe.GET()
- IProbe.MAXSIZE()
- IProbe.PROBE()
- IProbe.RECORD.DATA()
- IProbe.RECORD.TIME()
- IProbe.RECORDS()
- IProbe.REF()
- IProbe.SIZE()
- IProbe.STATE()
- IProbe.TRACK.RECORD()

IProbe() TRUE if IPROBE

Syntax: **IProbe()**

Returns TRUE if the debugger is running on a TRACE32-IPROBE hardware.

Return Value Type: Boolean.

IProbe.ADC.ENABLE() TRUE if channel is enabled

[Go to figure]

Syntax: **IProbe.ADC.ENABLE(***<channel>***)**

Returns TRUE if the specified analog channel of the IProbe is enabled.

Parameter Type: String.

Return Value Type: Boolean.

Example 1:

```
; The prefix ip stands for the IProbe of the Analog Probe.

; Returns FALSE if the V0 check box on the POD IP window is cleared.
PRINT IProbe.ADC.ENABLE(ip.V0)

; Returns TRUE if the V1 check box on the POD IP window is selected.
PRINT IProbe.ADC.ENABLE(ip.V1)
```

Example 2: To programmatically clear or select a checkbox on the **POD IP** window, use the **POD.ADC** command in a PRACTICE script file (*.cmm).

```
; Opens the POD IP window.
POD IP

; The keyword ON is used to select the V0 check box.
POD.ADC IP V0.ON
; Returns TRUE because the V0 check box is selected.
PRINT IProbe.ADC.ENABLE(ip.V0)  ; The prefix ip stands for the
                                ; IProbe of the Analog Probe.

; The keyword OFF is used to clear the V0 check box.
POD.ADC IP V0.OFF
; Returns FALSE because the V0 check box is cleared.
PRINT IProbe.ADC.ENABLE(ip.V0)
```

IProbe.ADC.SHUNT() Shunt resistor value of channel

[Go to figure]

Syntax: **IProbe.ADC.SHUNT(***<channel>***)**

Returns the shunt-resistor value of the specified current measurement *<channel>* of the IProbe.

Parameter Type: String.

Return Value Type: Float.

Example:

```
; The prefix ip stands for the IProbe of the Analog Probe.

; Returns the shunt-resistor value of the current channel I0.
PRINT IProbe.ADC.SHUNT(ip.I0)
```

IProbe.ANALOG() TRUE if Analog Probe is plugged

[Go to figure]

Syntax: **IProbe.ANALOG()**

When the PowerTrace II / PowerTrace III is powered up, this function returns if the connector of an Analog Probe is plugged.

Return Value Type: Boolean.

Return Value and Description:

TRUE	The connector of an Analog Probe is plugged in at the socket labeled LOGIC ANALYZER PROBE.
FALSE	The connector of the Analog Probe is *not* plugged in at the socket labeled LOGIC ANALYZER PROBE.

IProbe.FIRST() Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **IProbe.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

IProbe.GET() Value of channel

[Go to figure]

Syntax: **IProbe.GET(**<*channel_name*>**)**

Returns the current value of the channel. This function is primarily intended for the digital channels of the Standard Probe.

Parameter Type: String.

Return Value Type: Hex value. If there is no current value, the function returns 0FFFFFFFFFFFFFFF.

Example:

```
PRINT IProbe.GET(ip.V1)    ; Returns a hexadecimal value.

PRINT IProbe.GET(ip.V1)+0. ; Convert hex value to a decimal value.
```

[Go to figure]

Syntax: **IProbe.MAXSIZE()**

Returns the maximum size of the IProbe trace buffer in records. To return a user-defined trace buffer size, use the function **IProbe.SIZE()**.

Return Value Type: Decimal value.

Example: The command **IProbe.SIZE** is used to reduce the trace buffer size, e.g. to 600000 records, to speed up analysis. Using **IProbe.MAXSIZE()**, you can easily restore the maximum trace buffer size.

```
; Reduce the trace buffer size to the specified number of records.
IProbe.SIZE 600000.

; Restore the maximum trace buffer size.
IProbe.SIZE IProbe.MAXSIZE()
```

IProbe.PROBE()

[Go to figure]

Syntax: **IProbe.PROBE()**

When the PowerTrace II / PowerTrace III is powered up, this function returns if a connector is plugged.

Return Value Type: Boolean.

Return Value and Description:

TRUE	A connector is plugged in at the socket labeled LOGIC ANALYZER PROBE.
FALSE	No connector plugged in at the socket labeled LOGIC ANALYZER PROBE.

To check if the connector belongs to an Analog Probe, use the function **IProbe.ANALOG()**.

IProbe.RECORD.DATA() Get data recorded in trace record

Syntax: **IProbe.RECORD.DATA(***<record_number>***,***<channel>***)**

Returns the sampled data from the specified record.

Parameter and Description:

<record_number>	**Parameter Type**: Decimal value.
<channel>	**Parameter Type**: String.

Return Value Type: Hex value.

IProbe.RECORD.TIME() Get timestamp of trace record

Syntax: **IProbe.RECORD.TIME(***<record_number>***)**

Returns the timestamp from the specified record.

Parameter Type: Decimal value.

Return Value Type: Time value.

Example: For an example, see **Analyzer.RECORD.TIME()**.

[Go to figure]

Syntax: **IProbe.RECORDS()**

Returns the number of records in the IProbe trace buffer.

Return Value Type: Decimal value.

Examples:

```
; Print the number of records to the message bar below the command line.
PRINT IProbe.RECORDS()
```

```
; Opens an IProbe.List window.
IProbe.List /Track

; Go to the first record of the analog trace data.
IProbe.GOTO -IProbe.Records() ; Please note the minus sign.

; Go to the last record of the analog trace data.
IProbe.GOTO IProbe.RECORDS()  ; No minus sign in this case.
```

IProbe.REF() Get record number of reference record

Syntax: **IProbe.REF()**

Returns the record number of the reference record.

Return Value Type: Decimal value.

Example 1: The **IProbe.REF()** function is used as an argument in the **IProbe.GOTO** command to jump back to the reference record.

```
IProbe.GOTO IProbe.REF() ; Jump back to the reference record.
```

Example 2: The **IProbe.View** window is opened to display more information about the reference record.

```
LOCAL &refNo

; Get the number of the reference record.
&refNo=IProbe.Ref()

; Open the window with the reference record number displayed.
IProbe.View &refNo
```

IProbe.SIZE() Get current trace buffer size in records

[Go to figure]

Syntax: **IProbe.SIZE()**

Returns the *user-defined* logical size of the TRACE32-IPROBE trace buffer in records. To return the maximum size, use the function **IProbe.MAXSIZE()**. For more information about the IProbe trace buffer size, refer to the command **IProbe.SIZE**.

Return Value Type: Decimal value.

IProbe.STATE() Get state of IProbe

[Go to figure]

Syntax: **IProbe.STATE()**

Returns the state of the IProbe.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state

IProbe.TRACK.RECORD() Get record number matching search

[Go to figure]

Syntax: **IProbe.TRACK.RECORD()**

After a successful search operation, this function returns the record number. The record number returned is the number of the *first record* that matches the search criterion.

Return Value Type: Decimal value.

Example:

```
; Prerequisite: Analog trace data was recorded using the IProbe.

; In the power channel p0, search for 0.17xxxx.
IProbe.Find , IProbe.p0 0.17

IF FOUND()
(   ; Returns the first record matching the search criterion.
    AREA ; Print the record number to the AREA window.
    PRINT IProbe.TRACK.RECORD()

    ; Lists all matching records in a FindALL window.
    IProbe.FindALL , IProbe.p0 0.17 /Track
)
```

JTAG Functions

In This Section

See also

- JTAG
- JTAG.X7EFUSE
- JTAG.MIPI34()
- JTAG.SEQuence.EXIST()
- JTAG.SEQuence.RESULT()
- JTAG.X7EFUSE.CNTL()
- JTAG.X7EFUSE.KEY()
- JTAG.X7EFUSE.USER()
- JTAG.XUSEFUSE.DNA()
- JTAG.XUSEFUSE.RESULT()
- JTAG.XUSEFUSE.SEC()
- JTAG.XUSEFUSE.USER128()
- JTAG.SEQuence
- JTAG.XUSEFUSE
- JTAG.PIN()
- JTAG.SEQuence.LOCKED()
- JTAG.SHIFT()
- JTAG.X7EFUSE.DNA()
- JTAG.X7EFUSE.RESULT()
- JTAG.XUSEFUSE.CNTL()
- JTAG.XUSEFUSE.KEY()
- JTAG.XUSEFUSE.RSA()
- JTAG.XUSEFUSE.USER()

JTAG.MIPI34() Query special MIPI34 pins

[build 112622 - DVD 02/2020]

Syntax: **JTAG.MIPI34(**<*pin*>**)**

Query special pins on MIPI34 connector. Only works in conjunction with the CombiProbe/µTrace (MicroTrace) MIPI34 whisker.

The parameter must be a valid <*pin*> argument of the command **JTAG.MIPI34**, e. g. **PIN12**.

Parameter Type: String.

Return Value Type: Decimal value.

Return Value and Description:

0	The pin was measured at a low state.
1	The pin was measured at a high state.
-1	The pin could not be read (incorrect parameter or not a MIPI-34 whisker).

Syntax: **JTAG.PIN(***<signal_name>***)**

Reads the level of a JTAG signal. See command **JTAG.PIN**.

Parameter Type: String.

Return Value Type: Hex value.

JTAG.SEQuence.RESULT() Get result of JTAG sequence

ARC, ARM, TeakLite, Xtensa [build 97116 - DVD 09/2018]

Syntax: **JTAG.SEQuence.RESULT(***<global_seq_variable>***)**

<global_seq_variable>: **0** | **1**

Any JTAG sequence can assign values to the global sequence variables **Result0** and **Result1**.
The function **JTAG.SEQuence.RESULT()** returns the value of these two global sequence variables.

Parameter Type: Decimal value.

Return Value Type: Hex value.

Example:

```
SILENT.JTAG.SEQuence.Execute PowerCheck
IF JTAG.SEQuence.RESULT(0)==1
    ECHO "Power OK"
```

JTAG.SEQuence.EXIST() Check if a JTAG sequence exists

ARC, ARM, TeakLite, Xtensa [build 93345 - DVD 09/2018]

Syntax: **JTAG.SEQuence.EXIST(***<seq_name>***)**

Returns TRUE if the specified JTAG sequence name exists.

Parameter Type: String. For a description of *<seq_name>*, see **JTAG.SEQuence.ADD**.

Return Value Type: Boolean.

Syntax: **JTAG.SEQuence.LOCKED(***<seq_name>***)**

Returns TRUE if the specified JTAG sequence exists but is locked. A sequence is locked if it is assigned to an event (e.g. with **SYStem.CONFIG.MULTITAP.JtagSEQuence**) or if the sequence is an internal one which was created by TRACE32 after using **SYStem.CPU**.

Parameter Type: String. For a description of *<seq_name>*, see **JTAG.SEQuence.ADD**.

Return Value Type: Boolean.

JTAG.SHIFT() TDO output of JTAG shift

Syntax: **JTAG.SHIFT()**

Reads the TDO output of a JTAG shift. LSB first. Limited to 64 bit. See commands **JTAG.SHIFTREG** and **JTAG.SHIFTTDI**.

If you shift more than 64 bits with **JTAG.SHIFTREG** or **JTAG.SHIFTTDI** only *the first* 64-bits received by the debugger, are returned by **JTAG.SHIFT()**.

Return Value Type: Hex value.

Syntax: **JTAG.X7EFUSE.RESULT()**

Returns information about the success of a previous **JTAG.X7EFUSE** command.

Return Value Type: Decimal value.

Return Value and Description:

0.	OK, no fuses were blown.
1.	OK, fuses were blown as requested.
2.	Error, but no fuses were blown.
3.	Error, at least one fuse was blown, but there was not yet any attempt to program sensitive data (KEY or USER).
4.	Error, programming or verifying KEY or USER failed. However, all requested CNTL flags (if any) were programmed successfully.
5.	Error while programming or verifying CNTL flags. The specified security settings may not be in effect. It could be possible to extract secret KEY or USER data that was previously programmed.

Example: See **JTAG.X7EFUSE** command.

Syntax: **JTAG.X7EFUSE.CNTL()**

Returns CNTL flags read by the previous **JTAG.X7EFUSE** command. See Xilinx application note XAPP 1239, *Using Encryption to Secure a 7 Series FPGA Bitstream*, for a description of the values.

Return Value Type: Hex value.

Example:

```
JTAG.X7EFUSE /DEVICE XC7K325T

IF JTAG.X7EFUSE.RESULT()==0.
(
    IF (JTAG.X7EFUSE.CNTL()&1.<<0.)!=0
        PRINT "CFG_AES_ONLY is set!"
    IF (JTAG.X7EFUSE.CNTL()&1.<<1.)!=0
        PRINT "AES_EXCLUSIVE is set!"
    IF (JTAG.X7EFUSE.CNTL()&1.<<2.)!=0
        PRINT "W_EN_B_KEY_USER is set!"
    IF (JTAG.X7EFUSE.CNTL()&1.<<3.)!=0
        PRINT "R_EN_B_KEY is set!"
    IF (JTAG.X7EFUSE.CNTL()&1.<<4.)!=0
        PRINT "R_EN_B_USER is set!"
    IF (JTAG.X7EFUSE.CNTL()&1.<<5.)!=0
        PRINT "W_EN_B_CNTL is set!"
)
```

JTAG.X7EFUSE.DNA() DNA value read by JTAG.X7EFUSE command

Syntax: **JTAG.X7EFUSE.DNA()**

Returns the unique DNA value stored in every Xilinx 7-series device, as read by the previous **JTAG.X7EFUSE** command.

Return Value Type: Hex value.

Example: See **JTAG.X7EFUSE** command.

JTAG.X7EFUSE.KEY() AES key read by JTAG.X7EFUSE command

[build 76023 - DVD 09/2016]

Syntax: **JTAG.X7EFUSE.KEY()**

Returns the KEY value as read by the previous **JTAG.X7EFUSE** command. If prohibited by the R_EN_B_KEY flag, return zeros instead. The returned value is encoded as a hexadecimal string with the same bit order as the one stored in the.nky file.

Return Value Type: String.

Example: See **JTAG.X7EFUSE**.

JTAG.X7EFUSE.USER() User code read by JTAG.X7EFUSE command

[build 76023 - DVD 09/2016]

Syntax: **JTAG.X7EFUSE.USER()**

Returns the USER value as read by the previous **JTAG.X7EFUSE** command. If prohibited by the R_EN_B_USER flag, return zeros instead.

Return Value Type: Hex value.

Example: See **JTAG.X7EFUSE**.

[build 109218 - DVD 09/2019]

Syntax: **JTAG.XUSEFUSE.RESULT()**

Returns information about the success of a previous **JTAG.XUSEFUSE** command.

Return Value Type: Decimal value.

Return Value and Description:

0	OK, the command was executed successfully.
1	Error, but no fuses were blown.
2	Error, fuses might be blown. Verify the current values by using the command **JTAG.XUSEFUSE** with the **/READ** option.
3	Error while verifying the AES key. The CRCs of the given key and stored key are not the same.

Example: See **JTAG.XUSEFUSE** command.

JTAG.XUSEFUSE.CNTL() CNTL value read by JTAG.XUSEFUSE command

[build 109218 - DVD 09/2019]

Syntax: **JTAG.XUSEFUSE.CNTL()**

Returns the register value of the CNTL register read by a previous **JTAG.XUSEFUSE** command. See Xilinx user guide UG570, *UltraScale Architecture Configuration*, for a description of the value.

Return Value Type: Hex value.

Example:

```
SYStem.JtagClock 1.0MHz

JTAG.XUSEFUSE /READ 0x0

IF ((JTAG.XUSEFUSE.RESULT()!=1.)&&(JTAG.XUSEFUSE.RESULT()!=2.))
(
    IF (JTAG.XUSEFUSE.CNTL()&1.<<5.)!=0
        PRINT "W_DIS_CNTL is set!"
    IF (JTAG.XUSEFUSE.CNTL()&1.<<7.)!=0
        PRINT "W_DIS_KEY is set!"
)
```

JTAG.XUSEFUSE.DNA() DNA value read by JTAG.XUSEFUSE command

[build 109218 - DVD 09/2019]

Syntax: **JTAG.XUSEFUSE.DNA()**

Returns the unique DNA value stored in every Xilinx UltraScale series device, as read by the previous **JTAG.XUSEFUSE** command.

Return Value Type: String.

Example:

```
SYStem.JtagClock 1.0MHz

JTAG.XUSEFUSE /READ 0x0

PRINT JTAG.XUSEFUSE.DNA()
```

JTAG.XUSEFUSE.KEY() AES key read by JTAG.XUSEFUSE command

[build 109218 - DVD 09/2019]

Syntax: **JTAG.XUSEFUSE.KEY()**

Returns the KEY value as stated by the previous **JTAG.XUSEFUSE** command. If the key stored on the FPGA is not the same as the one provided by the user, the function **JTAG.XUSEFUSE.RESULT()** will return the value 3. The returned value is encoded as a hexadecimal string with the same bit order as the one stored in the *.nkz file.

Return Value Type: String.

Example:

```
SYStem.JtagClock 1.0MHz

JTAG.XUSEFUSE /READ \
0x0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF

IF JTAG.XUSEFUSE.RESULT()==0.
    PRINT "AES KEY matches the value: "
ELSE IF JTAG.XUSEFUSE.RESULT()==3.
    PRINT "AES KEY does not match the value: "
ELSE
    PRINT "Error while verifying the key: "

PRINT %CONTinue JTAG.XUSEFUSE.KEY()
```

JTAG.XUSEFUSE.RSA() RSA hash read by JTAG.XUSEFUSE command

[build 109218 - DVD 09/2019]

Syntax: **JTAG.XUSEFUSE.RSA()**

Returns the value of the RSA hash value as read by the previous **JTAG.XUSEFUSE** command. If prohibited by the flags in the CNTL register, the return value is 0xFFF...FF. The retuned value is encoded as a hexadecimal string with the same bit order as the one stored in the *.nkz file.

Return Value Type: String.

Example: See **JTAG.XUSEFUSE**

JTAG.XUSEFUSE.SEC() SEC value read by JTAG.XUSEFUSE command

[build 109218 - DVD 09/2019]

Syntax: **JTAG.XUSEFUSE.SEC()**

Returns the register value of the SEC register read by a previous **JTAG.XUSEFUSE** command. See Xilinx user guide UG570, *UltraScale Architecture Configuration*, for a description of the value.

Return Value Type: Hex value.

Example:

```
SYStem.JtagClock 1.0MHz

JTAG.XUSEFUSE /READ 0x0

IF ((JTAG.XUSEFUSE.RESULT()!=1.)&&(JTAG.XUSEFUSE.RESULT()!=2.))
(
    IF (JTAG.XUSEFUSE.SEC()&1.<<2.)!=0
        PRINT "RSA authentication is forced!"
    IF (JTAG.XUSEFUSE.CNTL()&1.<<4.)!=0
        PRINT "Xilinx test access is disabled!"
)
```

JTAG.XUSEFUSE.USER() User code read by JTAG.XUSEFUSE command

[build 109218 - DVD 09/2019]

Syntax: **JTAG.XUSEFUSE.USER()**

Returns the register value as read by a previous **JTAG.XUSEFUSE** command. If prohibited by the flags in the CNTL register, the return value is 0xFFF...FF.

Return Value Type: Hex value.

Example: See **JTAG.XUSEFUSE**

JTAG.XUSEFUSE.USER128() 128 bit User code read by JTAG.XUSEFUSE

[build 109218 - DVD 09/2019]

Syntax: **JTAG.XUSEFUSE.USER128()**

Returns the 128 bit USER value as read by a previous **JTAG.XUSEFUSE** command. If prohibited by the flags in the CNTL register, the return value is 0xFFF...FF. The returned value is encoded as a hexadecimal string with the same bit order as the one stored in the *.nkz file.

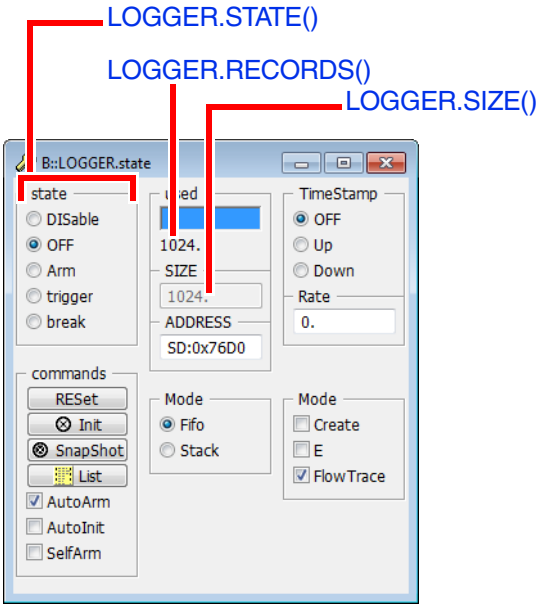
Return Value Type: String.

Example:

```
SYStem.JtagClock 1.0MHz

JTAG.XUSEFUSE /READ 0x0

IF ((JTAG.XUSEFUSE.RESULT()!=1.)&&(JTAG.XUSEFUSE.RESULT()!=2.))
    PRINT JTAG.XUSEFUSE.USER128()
ELSE
    PRINT "Error while reading the eFUSE values!"
```

LOGGER Functions

This figure provides an overview of the return values of some of the LOGGER functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- LOGGER.FIRST()
- LOGGER.RECORD.DATA()
- LOGGER.RECORD.TIME()
- LOGGER.REF()
- LOGGER.STATE()
- LOGGER.RECORD.ADDRESS()
- LOGGER.RECORD.OFFSET()
- LOGGER.RECORDS()
- LOGGER.SIZE()

LOGGER.FIRST() Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **LOGGER.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

LOGGER.RECORD.ADDRESS() Get address recorded in trace record

[build 38764]

Syntax: **LOGGER.RECORD.ADDRESS(**<*record_number*>**)**

Returns the sampled address (access class and offset) from the specified record.

Parameter Type: Decimal value.

Return Value Type: Address.

LOGGER.RECORD.DATA() Get data recorded in trace record

[build 38764]

Syntax: **LOGGER.RECORD.DATA(**<*record_number*>**)**

Returns the sampled data of the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

LOGGER.RECORD.OFFSET() Get address in trace record as number

[build 38764]

Syntax: **LOGGER.RECORD.OFFSET(**<*record_number*>**)**

Returns the address-offset of the sampled address from the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

LOGGER.RECORD.TIME() Get timestamp of trace record

[[build 38764]

Syntax: **LOGGER.RECORD.TIME(**<*record_number*>**)**

Returns the timestamp of the specified record. For an example, see **Analyzer.RECORD.TIME()**.

Parameter Type: Decimal value.

Return Value Type: Time value.

LOGGER.RECORDS() Get number of used trace records

[Go to figure]

Syntax: **LOGGER.RECORDS()**

Returns the number of records used for Logger trace.

Return Value Type: Decimal value.

LOGGER.REF() Get record number of reference record

Syntax: **LOGGER.REF()**

Returns the number of the selected reference record in the Logger trace.

Return Value Type: Decimal value.

LOGGER.SIZE() Get current trace buffer size in records

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **LOGGER.SIZE()**

Returns the size of the Logger trace buffer. The size is given by the logger buffer size of the target application.

Return Value Type: Decimal value.

[Go to figure]

Syntax: **LOGGER.STATE()**

Returns the state of the Logger trace.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state

MACHO.LASTUUID() Universally unique identifier of MachO file

Syntax: **MACHO.LASTUUID()**

Returns the 128-bit universally unique identifier (UUID) of the file loaded with the last **Data.LOAD.MachO** command. With no MachO file was load yet or the last file did not contain an UUID the string “no UUID read” will be returned.

Return Value Type: String.

MAP Functions

In This Section

See also

■ MAP ❏ MAP.ROMSIZE()

MAP.ROMSIZE() Size of the defined ROM

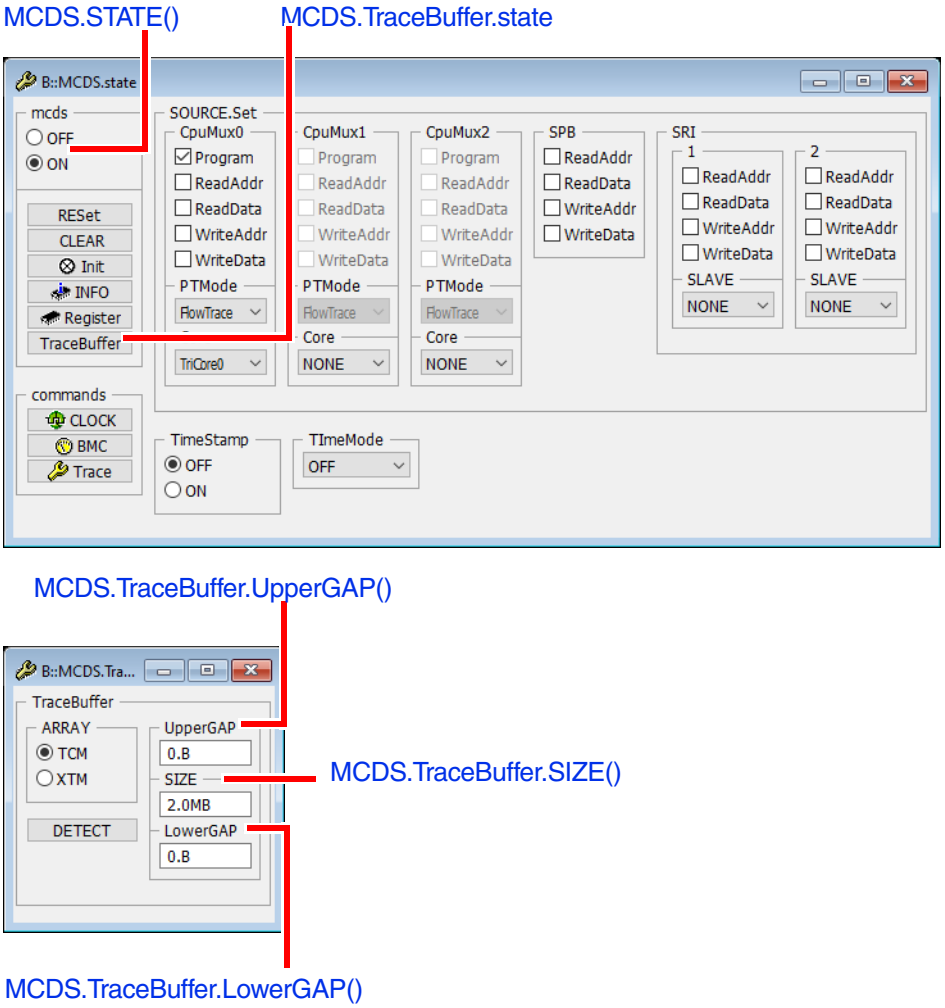
Syntax: **MAP.ROMSIZE()**

Returns the total size of the defined ROM.

Return Value Type: Decimal value.

MCDS Functions

This figure provides an overview of the return values of some of the functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- MCDS
- MCDS.GAP()
- MCDS.MODULE.NAME()
- MCDS.MODULE.NUMBER()
- MCDS.MODULE.REVision()
- MCDS.MODULE.TYPE()
- MCDS.SIZE()
- MCDS.STATE()
- MCDS.TraceBuffer.LowerGAP()
- MCDS.TraceBuffer.SIZE()
- MCDS.TraceBuffer.UpperGAP()

MCDS.MODULE.NAME() Name of MCDS module

[build 70152 - DVD 09/2016]

Syntax: **MCDS.MODULE.NAME()**

Returns the name of the MCDS module according to CPU selection.

Return Value Type: String.

Return Value and Description:

<undefined>	Unknown or undefined MCDS module. Contact technical support.
<none>	Devices does not feature an MCDS module.
MCDS	Standard MCDS module.
MCDSlight	MCDSlight module
miniMCDS	miniMCDS module.

MCDS.MODULE.NUMBER() Number-part of MCDS module ID

[build 70152 - DVD 09/2016]

Syntax: **MCDS.MODULE.NUMBER()**

Returns the number-part of the MCDS module ID of the attached chip.

Return Value Type: Hex value.

Return Value and Description:

0xFFFF	MCDS ID register could not be read. Switch to **SYStem.Mode Up** if necessary.
0x0000	Devices does not feature an MCDS module.
other	MCDS_ID.NUMBER (16 bit)

MCDS.MODULE.REVision() Revision-part of MCDS module ID

[build 70152 - DVD 09/2016]

Syntax: **MCDS.MODULE.REVision()**

Returns the revision-part of the MCDS module ID of the attached chip.

Return Value Type: Hex value.

Return Value and Description:

0xFF	MCDS ID register could not be read. Switch to **SYStem.Mode Up** if necessary.
0x00	Devices does not feature an MCDS module.
other	MCDS_ID.REVISION (8 bit)

MCDS.MODULE.TYPE() Type-part of MCDS module ID

[build 70152 - DVD 09/2016]

Syntax: **MCDS.MODULE.TYPE()**

Returns the type-part of the MCDS module ID of the attached chip.

Return Value Type: Hex value.

Return Value and Description:

0xFF	MCDS ID register could not be read. Switch to **SYStem.Mode Up** if necessary.
0x00	Devices does not feature an MCDS module.
(other)	MCDS_ID.TYPE (8 bit)

MCDS.STATE() MCDS module is switched on/off

[Go to figure]

Syntax: **MCDS.STATE()**

Returns the state of the MCDS module.

Return Value Type: Hex value.

Return Value and Description:

0	OFF
1	ON

See also: Command group **MCDS**

[Go to figure]

Syntax: **MCDS.TraceBuffer.LowerGAP()**

Returns the size of the lower EMEM tiles of the currently selected memory array not used as trace buffer.

Return Value Type: Decimal value.

MCDS.TraceBuffer.SIZE() Trace buffer size

[Go to figure]

Syntax: **MCDS.TraceBuffer.SIZE()**
MCDS.SIZE() (deprecated)

Returns the size of the EMEM tiles of the currently selected memory array used as trace buffer.

Return Value Type: Decimal value.

MCDS.TraceBuffer.UpperGAP() Trace buffer upper gap

[Go to figure]

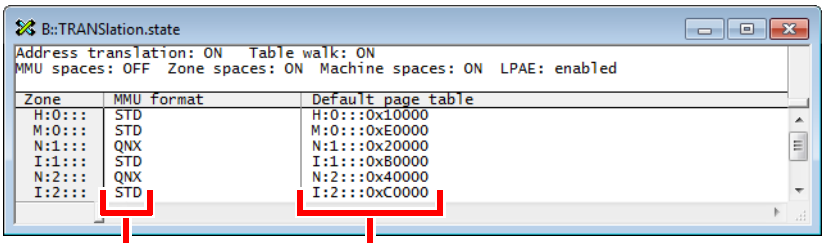
Syntax: **MCDS.TraceBuffer.UpperGAP()**
MCDS.GAP() (deprecated)

Returns the size of the upper EMEM tiles of the currently selected memory array not used as trace buffer.

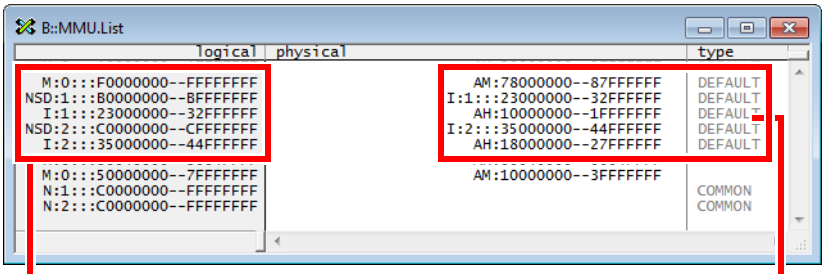
Return Value Type: Decimal value.

MMU Functions (Memory Management Unit)

This figure provides an overview of the return values of some of the functions. For descriptions of the illustrated functions and the functions not shown here, see below.



MMU.FORMAT.ZONE() MMU.DEFAULTPT.ZONE()



MMU.DEFAULTTRANS.LOGRANGE.ZONE() MMU.DEFAULTTRANS.PHYSADDR.ZONE()

In This Section

See also

- MMU
- MMU.DEFAULTPT()
- MMU.DEFAULTTRANS.LOGRANGE()
- MMU.FORMAT()
- MMU.FORMAT.ZONE()
- MMU.INTERMEDIATE.VALID()
- MMU.INTERMEDIATEEX.VALID()
- MMU.LINEAR.VALID()
- MMU.LINEAREX.VALID()
- MMU.LOGICAL.VALID()
- MMU.PHYSICAL.VALID()
- MMU.PHYSICALEX.VALID()
- MMU()
- MMU.DEFAULTPT.ZONE()
- MMU.DEFAULTTRANS.PHYSADDR()
- MMU.FORMAT.DETECTED()
- MMU.INTERMEDIATE()
- MMU.INTERMEDIATEEX()
- MMU.LINEAR()
- MMU.LINEAREX()
- MMU.LOGICAL()
- MMU.PHYSICAL()
- MMU.PHYSICALEX()

MMU()

Value of MMU register

Syntax: **MMU(***<register_name>***)**

Returns an MMU register value.

Parameter Type: String.
Return Value Type: Hex value.

MMU.DEFAULTPT() Base address of default page table

[Go to figure]

Syntax 1:	**MMU.DEFAULTPT()** [build 61046 - DVD 02/2015]
Syntax 2:	**MMU.DEFAULTPT.ZONE(***<address>***)** [build 106522 - DVD 09/2019]
<address>:	*<access_class>*:[*<machine_id>*:::]**0x0**

Both MMU functions return the base address of the default page table that has been set with the **MMU.FORMAT** command.
Syntax 1: The returned base address is located in the currently active zone of the core.
Syntax 2: The returned base address is located in the zone that is selected with *<address>*.

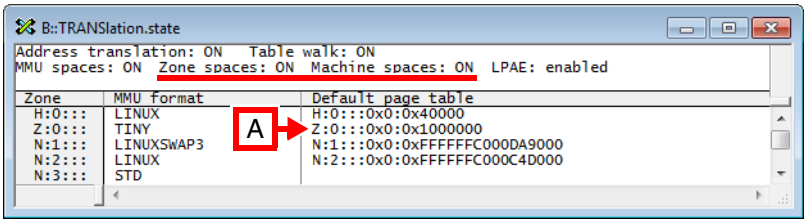
Return Value Type: Address. If the CPU has no MMU, a zero address is returned.

Parameter and Description:

<access_class>	Mandatory if **SYStem.Option.ZoneSPACES** is **ON**. See access class in the glossary.pdf.
<machine_id>	Mandatory if **SYStem.Option.MACHINESPACES** is **ON**. See machine ID in the glossary.pdf.
0x0	Fixed *<address>* suffix.

Example - Syntax 2:

```
;optional step: list the base addresses of the default page tables
;per zone
TRANSlation.state                          ;see column 'Default page table'

;let's return the base address of the default page table of
;machine '0:::' in the secure zone 'Z'
PRINT MMU.DEFAULTPT.ZONE(Z:0:::0x0)        ;result 'Z:0:::0x0:0x1000000'
                                           ;see [A] below
```



[Go to figure] [Examples]

Syntax 1:	**MMU.DEFAULTTRANS.***<range>***()**
Syntax 2:	**MMU.DEFAULTTRANS.***<range>***.ZONE(***<address>***)**
<range>:	**LOGRANGE** | **PHYSADDR**
<address>:	*<access_class>***:**[*<machine_id>***:::]0x0**
Full function name only required for HELP.Index:	**MMU.DEFAULTTRANS.LOGRANGE()** **MMU.DEFAULTTRANS.PHYSADDR()** [build 95143 - DVD 09/2018] **MMU.DEFAULTTRANS.LOGRANGE.ZONE(***<address>***)** **MMU.DEFAULTTRANS.PHYSADDR.ZONE(***<address>***)** [build 106522 - DVD 09/2019]

Both MMU functions return settings that have been made with the **MMU.FORMAT** command.
Syntax 1: The returned settings are located in the currently active zone of the core.
Syntax 2: The returned settings are located in the zone that is selected with *<address>*.

Return Value and Description:

<range>	Description
LOGRANGE	**Return Value Type**: Address range. Use this keyword to get the logical address range of the default translation.
PHYSADDR	**Return Value Type**: Address. Use this keyword to get the physical base address of the default translation.

Parameter and Description:

<access_class>	**Parameter Type**: Address. Mandatory if **SYStem.Option.ZoneSPACES** is **ON**. See access class in the glossary.pdf.
<machine_id>	**Parameter Type**: Address. Mandatory if **SYStem.Option.MACHINESPACES** is **ON**. See machine ID in the glossary.pdf.
0x0	Fixed *<address>* suffix.

Examples - Syntax 1

Example 1: The function returns the logical address range of the default translation that is currently active on the core.

```
MMU.FORMAT STD      0x88000000      0x80000000--0x9FFFFFFF      0x10000000

PRINT MMU.DEFAULTTRANS.LOGRANGE()
                          ;returns C:0x80000000--0x9FFFFFFF
```

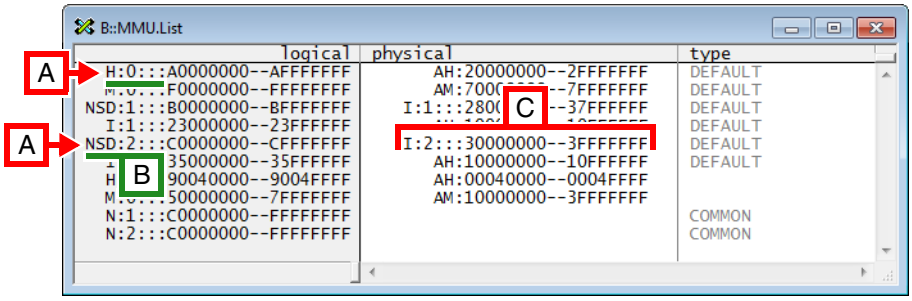
Example 2: The function returns the physical base address of the default translation that is currently active on the core.

```
MMU.FORMAT STD      0x88000000      0x80000000--0x9FFFFFFF      0x10000000
PRINT MMU.DEFAULTTRANS.PHYSADDR()                    ;returns A:0x10000000
```

Example - Syntax 2

Example 3: Based on the zones **N:0x2:::** and **H:0x0:::** passed as arguments, the function **MMU.DEFAULTTRANS.LOGRANGE.ZONE()** returns the logical address ranges of these zones.

```
MMU.FORMAT STD     H:0x0:::0x10000     H:0x0:::0xA0000000--0xAFFFFFFF \
                                                    A:0x20000000 /MACHINE 0
PRINT MMU.DEFAULTTRANS.LOGRANGE.ZONE(H:0x0:::0x0)
                                ;returns H:0:::0xA0000000--0xAFFFFFFF


MMU.FORMAT QNX     N:0x2:::0x40000     N:0x2:::0xC0000000--0xCFFFFFFF \
                                                    I:0x30000000 /MACHINE 2
PRINT MMU.DEFAULTTRANS.LOGRANGE.ZONE(N:0x2:::0x0)
                                ;returns NSD:2:::0xC0000000--0xCFFFFFFF
```



A Logical address range.

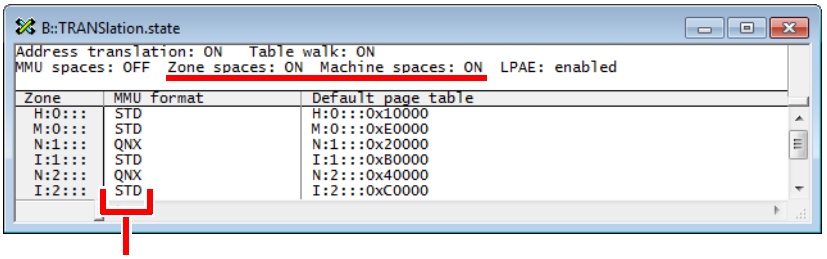
B Zone of the logical address range.

C Physical address range.

[Go to figure]

Syntax 1:	**MMU.FORMAT()** [build 53385 - DVD 08/2014]
Syntax 2:	**MMU.FORMAT.ZONE(***<address>***)** [build 106522 - DVD 09/2019]
<address>:	*<access_class>*:[*<machine_id>*:::]**0x0**

Both MMU functions return the MMU format of the default page table that has been set with the **MMU.FORMAT** command.
Syntax 1: The returned MMU format is the format of the currently active zone of the core.
Syntax 2: The returned MMU format is the format of the zone that is selected with *<address>*.



MMU.FORMAT()
MMU.FORMAT.ZONE()

Parameter and Description:

<access_class>	**Parameter Type**: Address. Mandatory if **SYStem.Option.ZoneSPACES** is **ON**. See access class in the glossary.pdf.
<machine_id>	**Parameter Type**: Address. Mandatory if **SYStem.Option.MACHINESPACES** is **ON**. See machine ID in the glossary.pdf.
0x0	Fixed *<address>* suffix.

Return Value Type: String. If the CPU has no MMU, an empty string is returned.

Example - Syntax 2:

```
PRINT MMU.FORMAT.ZONE(N:2:::0x0)     ;returns QNX
```

Syntax: **MMU.FORMAT.DETECTED()**

Auto-detects the page table format used by the kernel in the currently active zone of the core. Then the function returns the name of the page table format. The auto-detection works only if **MMU.FORMAT** is set to **STD**.

Return Value Type: String.

Return Value and Description: For a list of format names, see:

- "**<format> Options for x86**" (general_ref_m.pdf)
- "**<format> Options for RISC-V**" (general_ref_m.pdf)

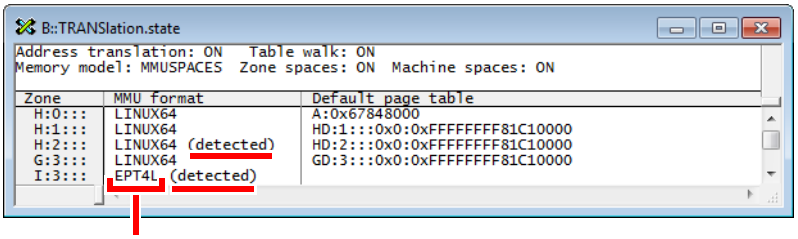
Example:

```
MMU.FORMAT STD
PRINT MMU.FORMAT.DETECTED()     ;returns the name of the standard format
                                ;derived from the CPU state, e.g.
                                ;the format name P32
```

Syntax: **MMU.FORMAT.DETECTED.ZONE(***<address>***)**

<address>: *<access_class>*:[*<machine_id>*:::]**0x0**

Auto-detects the page table format used by the kernel and returns its name. The returned page table format refers to the zone that is selected with *<address>*. The auto-detection works only if **MMU.FORMAT** is set to **STD**.



MMU.FORMAT.DETECTED.ZONE()

Parameter Type: Address.

Parameter and Description:

<access_class>	Mandatory if **SYStem.Option.ZoneSPACES** is **ON**. See access class in the glossary.pdf.
<machine_id>	Mandatory if **SYStem.Option.MACHINESPACES** is **ON**. See machine ID in the glossary.pdf.
0x0	Fixed *<address>* suffix.

Return Value Type: String.

Return Value and Description: For a list of format names, see:

- "**<format> Options for x86**" (general_ref_m.pdf)
- "**<format> Options for RISC-V**" (general_ref_m.pdf)

Example:

```
SYStem.Option.MACHINESPACES ON
MMU.FORMAT STD /MACHINE 2
PRINT MMU.FORMAT.DETECTED.ZONE(I:3:::0)
                                ;returns the name of the standard
                                ;page table format used in zone "I:3:::0"
                                ;and derived from the CPU state, e.g.
                                ;the format name "EPT4L"
```

MMX Function (MultiMedia eXtension)

MMX() Value of MMX register

Syntax: **MMX(***<register_name>***)**

Returns the content of the selected MMX register. See also **MMX** command group.

Parameter Type: String.

Return Value Type: Hex value.

MONITOR Function

MONITOR() TRUE if debugger is running as monitor

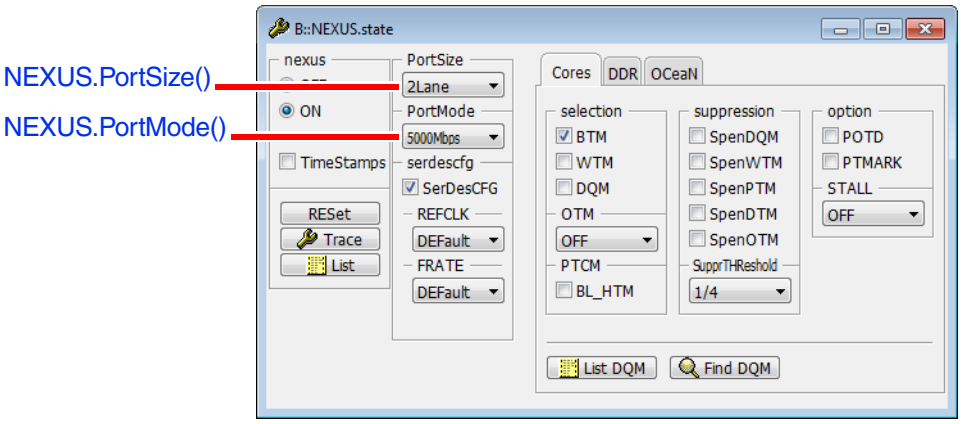
Syntax: **MONITOR()**

Returns TRUE if the debugger is running as monitor.

Return Value Type: Boolean.

NEXUS Functions

This figure provides an overview of the return values of some of the NEXUS functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- ❏ NEXUS()
- ❏ NEXUS.PortMode()
- ❏ NEXUS.PortSize()
- ❏ NEXUS.RTTBUILD()

NEXUS() TRUE if Nexus trace is supported

Syntax: **NEXUS()**

Returns TRUE if the selected CPU supports a Nexus trace.

Return Value Type: Boolean.

NEXUS.RTTBUILD() RTT build register

ARC [build 93248 - DVD 02/2018]

Syntax: **NEXUS.RTTBUILD(***<register_index>***)**

<register_index>: **0**

Returns the RTT build register.

Parameter Type: Decimal value.

Return Value Type: Hex value. Returns 0 if the command **NEXUS.RTTBUILD** was *not* used or if there is *no* RTT trace source.

NEXUS.PortMode() Current PortMode setting

[Go to figure]

Syntax: **NEXUS.PortMode()**

Returns the current Nexus **PortMode** setting.

Return Value Type: String.

NEXUS.PortSize() Current PortSize setting

[build 39453 - DVD 08/2012] [Go to figure]

Syntax: **NEXUS.PortSize()**

Returns the current Nexus **PortSize** setting.

Return Value Type: String.

Onchip Functions

In This Section

See also

- Onchip()
- Onchip.FLOW.ERRORS()
- Onchip.MAXSIZE()
- Onchip.RECORD.DATA()
- Onchip.RECORD.TIME()
- Onchip.REF()
- Onchip.STATE()
- Onchip.TRACK.RECORD()
- Onchip.FIRST()
- Onchip.FLOW.FIFOFULL()
- Onchip.RECORD.ADDRESS()
- Onchip.RECORD.OFFSET()
- Onchip.RECORDS()
- Onchip.SIZE()
- Onchip.TraceCONNECT()

Onchip() — TRUE if the onchip trace is available

Syntax: **Onchip()**

Returns TRUE if an onchip trace is available.

Return Value Type: Boolean.

Onchip.FIRST() — Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **Onchip.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

Onchip.FLOW.ERRORS() — Get number of flow errors / hard errors

Syntax: **Onchip.FLOW.ERRORS()**

Returns the number of flow errors / harderrors in the trace recording.

Return Value Type: Decimal value.

Please be aware that the return value of this function is the accumulated count of events that were encountered while processing the trace recording. All opened windows showing trace data contribute to this value. The value is reset when a new trace recording is made, or when the **Trace.FLOWSTART** or **Trace.FLOWPROCESS** command is executed.

The use of this function is only recommended if you want to find out if a specified part of a trace recording is error free. The part to be analyzed can be defined using **Trace.STATistic.FIRST** and **Trace.STATistic.LAST**. If the defined part is error free (and thus this function returns zero), the analysis results are reliable as well.

Example 1: This script shows how to return only the number of flow errors and hard errors in the trace that is currently visible within the **Onchip.List** window. If you now scroll up or down in the **Onchip.List** window or increase the window size, more trace data will be decoded, and thus the number of errors returned by the function may increase.

```
Onchip.List

PRINT Onchip.FLOW.ERRORS()

; scroll up or down in the window

PRINT Onchip.FLOW.ERRORS()
```

Example 2: This script shows how to obtain the exact number of flow errors in the whole trace recording.

```
Trace.Find FLOWERROR /ALL
PRINT FOUND.COUNT()
```

Onchip.FLOW.FIFOFULL() Get number of FIFO overflows

Syntax: **Onchip.FLOW.FIFOFULL()**

Returns the number of target FIFO overflows in the trace recording.

Return Value Type: Decimal value.

Please be aware that the return value of this function is the accumulated count of events that were encountered while processing the trace recording. All opened windows showing trace data contribute to this value. The value is reset when a new trace recording is made, or when the **Trace.FLOWSTART** or **Trace.FLOWPROCESS** command is executed.

The use of this function is only recommended if you want to find out if a specified part of a trace recording is error free. The part to be analyzed can be defined using **Trace.STATistic.FIRST** and **Trace.STATistic.LAST**. If the defined part is error free (and thus this function returns zero), the analysis results are reliable as well.

Onchip.MAXSIZE() Get max. size of trace buffer in records

[build 38323 - DVD 08/2012]

Syntax: **Onchip.MAXSIZE()**

Returns the maximum possible size of the Onchip trace buffer in records.

Return Value Type: Decimal value.

Onchip.RECORD.ADDRESS() Get address recorded in trace record

[build 38764]

Syntax: **Onchip.RECORD.ADDRESS(***<record_number>***)**

Returns the sampled address (access class and offset) from the specified record.

Parameter Type: Decimal value.

Return Value Type: Address.

Onchip.RECORD.DATA() Get data recorded in trace record

[build 38764]

Syntax: **Onchip.RECORD.DATA(***<record_number>***)**

Returns the sampled data of the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

Onchip.RECORD.OFFSET() Get address in trace record as number

[build 38764]

Syntax: **Onchip.RECORD.OFFSET(***<record_number>***)**

Returns the address-offset of the sampled address from the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

Onchip.RECORD.TIME() Get timestamp of trace record

Syntax: **Onchip.RECORD.TIME(**<*record_number*>**)**

Returns the timestamp of the specified record.

Parameter Type: Decimal value.

Return Value Type: Time value.

Example: For an example, see **Analyzer.RECORD.TIME()**.

Onchip.RECORDS() Get number of used trace records

Syntax: **Onchip.RECORDS()**

The number of records currently stored in the on-chip trace buffer.

Return Value Type: Decimal value.

Onchip.REF() Get record number of reference record

Syntax: **Onchip.REF()**

Returns the number of the selected reference record in the Onchip trace.

Return Value Type: Decimal value.

Onchip.SIZE() Get current trace buffer size in records

Syntax: **Onchip.SIZE()**

Size of on-chip memory to be used as on-chip trace buffer in trace records.

Return Value Type: Decimal value.

Onchip.STATE() Get state of Onchip trace

Syntax: **Onchip.STATE()**

Returns the state of the on-chip trace.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state

Onchip.TraceCONNECT() Name of trace sink of the SoC

[build 80222 - DVD 02/2017]

Syntax: **Onchip.TraceCONNECT()**

Returns the name of the currently selected trace sink of the SoC. In case no trace-sink is selected/available, the function returns **NONE**. The trace sink is selected with the **<trace>.TraceCONNECT** command.

Return Value Type: String.

Example: ARM Coresight system with two ETFs

```
SYStem.CONFIG ETF1 Base DAP:<address>
SYStem.CONFIG ETR2 Base DAP:<address>
Onchip.TraceCONNECT ETF1
PRINT Onchip.TraceCONNECT() ; returns "ETF1"
Onchip.TraceCONNECT ETF2
PRINT Onchip.TraceCONNECT() ; returns "ETF2"
```

Syntax: **Onchip.TRACK.RECORD()**

After a successful search operation, this function returns the record number.

Return Value Type: Decimal value.

Example: For an example, see **Analyzer.TRACK.RECORD()**.

PBI() Name of used debug back-end

[build 25366 - DVD 02/2011]

Syntax: **PBI()**

Returns the name of the used debug back-end. For further information, refer to **"Section PBI"** in TRACE32 Installation Guide, page 42 (installation.pdf).

Return Value Type: String.

PCI Functions

In This Section

See also

■ PCI ❑ PCI.Read.B() ❑ PCI.Read.L() ❑ PCI.Read.W()

PCI.Read.B() Byte from PCI register

Syntax: **PCI.Read.B(***<bus>***,***<device>***,***<function>***,***<register>***)**

Reads a byte from selected PCI register. For a description of the parameters, click **here**.

Return Value Type: Hex value.

PCI.Read.L() Long from PCI register

Syntax: **PCI.Read.L(***<bus>***,***<device>***,***<function>***,***<register>***)**

Reads a long from selected PCI register. For a description of the parameters, click **here**.

Return Value Type: Hex value.

PCI.Read.W() Word from PCI register

Syntax: **PCI.Read.W(***<bus>***,***<device>***,***<function>***,***<register>***)**

Reads a word from selected PCI register. For a description of the parameters, click **here**.

Parameter and Description:

<bus>	**Parameter Type**: Decimal or hex or binary value. PCI bus number.
<device>	**Parameter Type**: Decimal or hex or binary value. PCI device number.
<function>	**Parameter Type**: Decimal or hex or binary value. PCI function number.
<register>	**Parameter Type**: Decimal or hex or binary value. PCI register number.

Return Value Type: Hex value.

PER Functions

In This Section

See also

- PER
- PER.ADDRESS()
- PER.ARG()
- PER.BASE()
- PER.Buffer.Byte()
- PER.Buffer.Long()
- PER.Buffer.LongLong()
- PER.Buffer.Quad()
- PER.Buffer.Short()
- PER.Buffer.Word()
- PER.Byte()
- PER.EVAL()
- PER.HByte()
- PER.Long()
- PER.Long.BigEndian()
- PER.Long.LittleEndian()
- PER.LongLong()
- PER.LongLong.BigEndian()
- PER.LongLong.LittleEndian()
- PER.PByte()
- PER.Quad()
- PER.Quad.BigEndian()
- PER.Quad.LittleEndian()
- PER.SAVEINDEX()
- PER.SByte()
- PER.Short()
- PER.Short.BigEndian()
- PER.Short.LittleEndian()
- PER.SLong()
- PER.TByte()
- PER.VALUE()
- PER.VALUE.STRING()
- PER.Word()
- PER.Word.BigEndian()
- PER.Word.LittleEndian()

PER.<width>() Memory contents in default endianness

[build 76251 - DVD 09/2016]

Syntax:	**PER.**<*width*>**(**<*address*>**)**
<*width*>:	**Byte** | **Short** | **Word** | **TByte** | **Long** | **PByte** | **HByte** | **SByte** | **SLong** | **Quad** | **LongLong**
Full function name only required for HELP.Index:	**PER.Byte(**<*address*>**)** **PER.Short(**<*address*>**)** **PER.Word(**<*address*>**)** **PER.TByte(**<*address*>**)** **PER.Long(**<*address*>**)** **PER.PByte(**<*address*>**)** **PER.HByte(**<*address*>**)** **PER.SByte(**<*address*>**)** **PER.SLong(**<*address*>**)** **PER.Quad(**<*address*>**)** **PER.LongLong(**<*address*>**)**

The **PER.**<*width*>**()** functions return memory contents in the default endianness of the architecture.

Parameter Type: Address.

Return Value Type: Hex value.

The **PER.**<*width*>**()** functions are used in PER files and are synonyms for the **Data.**<*width*>**()** functions. However, the difference between the two function groups is that the return values of the **PER.**<*width*>**()** functions are based on dualport accesses, provided the respective PER file is opened with the **DualPort** option e.g.:

```
PER.view <file>.per /DualPort
```

<width>	Description of PER.<width>()
Byte	Returns a single byte from memory.
Short	Returns a word (16-bit) from memory.
Word	Returns a word (16-bit) from memory.
TByte	Returns a 3-byte value from memory.
Long	Returns a long value (32-bit) from memory.
PByte	Returns a 5-byte value from memory.
HByte	Returns a 6-byte value from memory.
SByte	Returns a 7-byte value from memory.
SLong	Reads signed long value from memory - sign extended internally to a 64-bit value.
Quad	Returns a 64-bit value from memory.
LongLong	Returns a 64-bit value from memory.

PER.<width>.<endianness>() Memory contents in specified endianness

[build 76251 - DVD 09/2016]

Syntax:	**PER.***<width>***.***<endianness>***(***<address>***)**
<width>:	**Byte** | **Short** | **Word** | **TByte** | **Long** | **PByte** | **HByte** | **SByte** | **SLong** | **Quad** | **LongLong**
<endianness>:	**LittleEndian** | **BigEndian**
Full function name only required for HELP.Index:	**PER.Short.BigEndian(***<address>***)** **PER.Short.LittleEndian(***<address>***)** **PER.Word.BigEndian(***<address>***)** **PER.Word.LittleEndian(***<address>***)** **PER.Long.BigEndian(***<address>***)** **PER.Long.LittleEndian(***<address>***)** **PER.LongLong.BigEndian(***<address>***)** **PER.LongLong.LittleEndian(***<address>***)** **PER.Quad.BigEndian(***<address>***)** **PER.Quad.LittleEndian(***<address>***)**

The **PER.***<width>***.***<endianness>***()** functions return memory contents in the specified endianness.

Parameter Type: Address.

Return Value Type: Hex value.

The **PER.**<*width*>**.**<*endianness*>**()** functions are used in PER files and are synonyms for the **Data.**<*width*>**.**<*endianness*>**()** functions. However, the difference between the two function groups is that the return values of the **PER.**<*width*>**.**<*endianness*>**()** functions are based on dualport accesses, provided the respective PER file is opened with the **DualPort** option e.g.:

```
PER.view <file>.per /DualPort
```

<width>.<endianness>	Description of PER.<width>.<endianness>()
Short.BigEndian	Returns a word (16-bit) from memory, while the byte order of the word is forced to big endian.
Short.LittleEndian	Returns a word (16-bit) from memory, while the byte order of the word is forced to little endian.
Word.BigEndian	Returns a word (16-bit) from memory, while the byte order of the word is forced to big endian.
Word.LittleEndian	Returns a word (16-bit) from memory, while the byte order of the word is forced to little endian.
Long.BigEndian	Returns a long value (32-bit) from memory, while the byte order of the word is forced to big endian.
Long.LittleEndian	Returns a long value (32-bit) from memory, while the byte order of the word is forced to endian endian.
LongLong.BigEndian	Returns a 64-bit value from memory, while the byte order of the word is forced to big endian.
LongLong.LittleEndian	Returns a 64-bit value from memory, while the byte order of the word is forced to little endian.
Quad.BigEndian	Returns a 64-bit value from memory, while the byte order of the word is forced to big endian.
Quad.LittleEndian	Returns a 64-bit value from memory, while the byte order of the word is forced to little endian.

PER.ADDRESS() Address of register(field)

[build 147535 - DVD 09/2022]

Syntax: **PER.ADDRESS("**<*path*>**")**

Returns the address of the register stated as <*path*>. Refer to **PER.Set.ByName** for a description of <*path*>.

Parameter Type: String.

Return Value Type: Address.

Syntax:	**PER.ADDRESS.**<*sub_cmd*>**(**<*address*>**)**
<*width*>:	**isNONSECURE** | **isNONSECUREEX** | **isSECURE** | **isSECUREEX**
Full function name only required for HELP.Index:	**PER.ADDRESS.isNONSECURE(**<*address*>**)** **PER.ADDRESS.isNONSECUREEX(**<*address*>**)** **PER.ADDRESS.isSECURE(**<*address*>**)** **PER.ADDRESS.isSECUREEX(**<*address*>**)**

The **PER.ADDRESS.**<*sub_cmd*>**()** functions are used in PER files and are synonyms respectively for the **ADDRESS.isNONSECURE()**, **ADDRESS.isNONSECUREEX()**, **ADDRESS.isSECURE()**, and **ADDRESS.isSECUREEX()** functions. However, the difference between the two function groups is that the return values of the **PER.ADDRESS.**<*sub_cmd*>**()** functions are based on the access class provided as parameter and can be overriden using the options Secure or NonSecure, e.g.:

```
PER.view <file>.per /Secure
PER.view <file>.per /NonSecure
```

Parameter Type: Address.

Return Value Type: Boolean.

<sub_cmd>	Description of PER.ADDRESS.<sub_cmd>()
isNONSECURE	Checks if the address, inside a peripheral definition file (PER file), passed as parameter will force a non-secure (TrustZone) access.
isNONSECUREEX	Checks if the address, inside a peripheral definition file (PER file), passed as parameter combined with the current CPU status will cause a non-secure (TrustZone) access.
isSECURE	Checks if the address, inside a peripheral definition file (PER file), passed as a parameter will force a Secure (TrustZone) access.
isSECUREEX	Checks if the address, inside a peripheral definition file (PER file), passed as a parameter combined with the current CPU status will cause an Secure (TrustZone) access.

PER.ARG() Argument of PER.view command

Syntax:	**PER.ARG()**

Returns the (optional) argument of the **PER.view** command. Only useful inside peripheral definition files.

Return Value Type: Hex value.

Syntax: **PER.ARG.ADDRESS()**

Returns the (optional) address argument of the **PER.view** command. Only useful inside peripheral definition files.

Return Value Type: Address.

PER.BASE() Last BASE address

[build 159996 - DVD 09/2023]

Syntax: **PER.BASE()**

Returns the address last passed to the **BASE** command. Only to be used within peripheral files.

Return Value Type: Address.

Syntax:	**PER.Buffer.***<width>***(***<index>***)**
<width>:	**Byte** | **Short** | **Word** | **Long** | **LongLong** | **Quad**
(Full function name only required for HELP.Index):	**PER.Buffer.Byte(***<index>***)** **PER.Buffer.Short(***<index>***)** **PER.Buffer.Word(***<index>***)** **PER.Buffer.Long(***<index>***)** **PER.Buffer.LongLong(***<index>***)** **PER.Buffer.Quad(***<index>***)**

<width>	**Description of PER.Buffer.<width>()**
Byte	Returns a byte from the SGROUP buffer. Only useful within a **SGROUP** of a PER-file.
Short **Word**	Returns a 16-bit word from the SGROUP buffer. Only useful within a **SGROUP** of a PER-file.
Long	Returns a 32-bit word from the SGROUP buffer. Only useful within a **SGROUP** of a PER-file.
LongLong **Quad**	Returns a 64-bit from the SGROUP buffer. Only useful within a **SGROUP** of a PER-file.

Parameter and Description:

<index>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

PER.EVAL() Value of expression in PER file

Syntax: **PER.EVAL(**<*integer*>**)**

Returns the value of a expression (defined with **BASE**) inside a peripheral definition file (PER file), which was defined after a **BASE**, **IF**, **ELIF** or **ELSE** command.

Parameter Type: Decimal or hex or binary value.

The parameter defines which expression is returned (0=first one).

NOTE 1:	The function returns only the last evaluated value of the expression. It will not evaluated the expression again. Expressions after **BASE**, will be evaluated by a **GROUP** command after the **BASE** command in a PER file.
NOTE 2:	The function must only be used in the context of **IF** or **ELIF**.

Return Value Type: Address.

PER.FILENAME() PER file name

Syntax: **PER.FILENAME()**

Returns the file name of the active peripheral definition file (PER file).

Return Value Type: String.

[build 142258 - DVD 02/2022]

Syntax: **PER.SAVEINDEX(***<address>***,***<width>***,***<index_address>***,***<index_width>***, **
***<index_value>*)**

Function returns memory contents at Address (*<address>*) while a value (*<index_value>*) is temporarily set in an index register (*<index_address>*). The access width can be set individually.

Sequence:

- Backup memory contents of memory location *<index_address>*
- Set value *<index_value>* to *<index_addr>*
- Read memory contents at memory location *<address>* using *<width>*
- Restore memory contents of memory location *<index_address>*

Parameter and Description:

<address>	Address to read. **Parameter Type**: Address.
<width>	Access width of *<address>*. **Parameter Type**: Decimal or hex or binary value.
<index_address>	Address of index register. **Parameter Type**: Address.
<index_width>	Access width of *<index_address>*. **Parameter Type**: Decimal or hex or binary value.
<index_value>	Index value. **Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

See also: per_prog.pdf - SAVEINDEX.

PER.VALUE() Value of register(field)

[build 147535 - DVD 09/2022]

Syntax: **PER.VALUE("***<path>***")**

Returns the value of the register or register field stated as *<path>*. Refer to **PER.Set.ByName** for a description of *<path>*.

Parameter Type: String.

Return Value Type: Hex value.

Syntax: **PER.VALUE.STRING("***<path>***")**

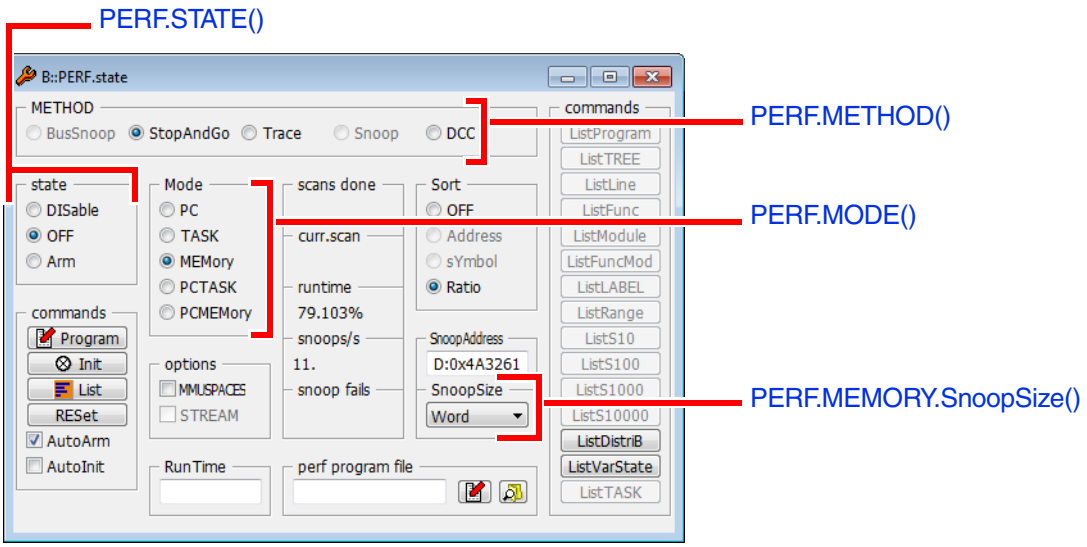
Returns the *<choice>* value of a **BITFLD.** Refer to **PER.Set.ByName** for a description of *<path>*.

Parameter Type: String.

Return Value Type: String.

PERF Functions (Performance)

This figure provides an overview of the return values of some of the PERF functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- PERF
- PERF.MEMORY.SnoopAddress()
- PERF.METHOD()
- PERF.PC.HITS()
- PERF.RunTime()
- PERF.STATE()
- PERF.MEMORY.HITS()
- PERF.MEMORY.SnoopSize()
- PERF.MODE()
- PERF.RATE()
- PERF.SNOOPFAILS()
- PERF.TASK.HITS()

PERF.MEMORY.HITS() Number of memory samples

Syntax: **PERF.MEMORY.HITS(***<value>*,*<core>***)**

Number of hits for the given memory value and core number.
Parameter and Description:

<value>	**Parameter Type**: Decimal or hex or binary value.
<core>	**Parameter Type**: Decimal value. If *<core>* is -1, the number of hits is returned for all cores.

Return Value Type: Decimal value.

PERF.MEMORY.SnoopAddress() Snoop memory address

Syntax: **PERF.MEMORY.SnoopAddress()**

Returns the snoop memory address.

Return Value Type: Address.

PERF.MEMORY.SnoopSize() Snoop size

[Go to figure]

Syntax: **PERF.MEMORY.SnoopSize()**

Returns the snoop size in bytes (1, 2, 4, or 8).

Return Value Type: Hex value.

PERF.METHOD() Recording method

[Go to figure]

Syntax: **PERF.METHOD()**

Returns the recording method of the Performance Analyzer.

Return Value Type: Hex value.

Return Value and Description:

0	**HARDWARE** method
1	**BusSnoop** method
2	**StopAndGo** method
3	**Trace** method
4	**Snoop** method
5	**DCC** method

PERF.MODE() Get Performance Analyzer recording mode

[Go to figure]

Syntax: **PERF.MODE()**

Returns the current recording mode of the Performance Analyzer.

Return Value Type: Hex value.

Return Value and Description:

1	**PC** mode
2	**FLAGs** mode
8	**TASK** mode
9	**PCTASK** mode
10	**MEMory** mode
11	**PCMEMory** mode
40	**LeVel** mode

PERF.PC.HITS() Number of PC samples

Syntax: **PERF.PC.HITS(***<address_range>***,***<core>***)**

Number of program counter hits for the given address range and core number.

Parameter and Description:

<address_range>	**Parameter Type**: Address range.
<core>	**Parameter Type**: Decimal value. If *<core>* is -1, the number of hits is returned for all cores.

Return Value Type: Decimal value.

PERF.RATE() Number of snoops per second

Syntax: **PERF.RATE()**

Returns the number of snoops per second.

Return Value Type: Decimal value.

PERF.RunTime() Retained time for program run

Syntax: **PERF.RunTime()**

Percentage of time retained for the actual program run when the **StopAndGo** method is used.

Return Value Type: String.

PERF.SNOOPFAILS() Number of snoop fails

Syntax: **PERF.SNOOPFAILS()**

Returns the number of snoop fails.

Return Value Type: Decimal value.

PERF.STATE() Get state of Performance Analyzer

[Go to figure]

Syntax: **PERF.STATE()**

Returns the state of the Performance Analyzer.

Return Value Type: Hex value.

Return Value and Description:

0	**DISable** state
1	**OFF** state
2	**Arm** state

Syntax: **PERF.TASK.HITS(***<task_magic>***,***<core>***)**

Returns the number of hits for the given task magic number and core number.

Parameter and Description:

<task_magic>	**Parameter Type**: Decimal or hex or binary value.
<core>	**Parameter Type**: Decimal value. If *<core>* is -1, the number of hits is returned for all cores.

Return Value Type: Decimal value.

Port Analyzer Functions

In This Section

See also

- PORT.GET()
- PORT.MAXSIZE()
- PORT.RECORDS()
- PORT.REF()
- PORT.SIZE()
- PORT.STATE()
- PORT.TRACK.RECORD()

PORT.GET() Value of channel

Syntax: **PORT.GET(**<*channel_name*>**)**

Returns the current value of the given port channel.

Parameter Type: String.

Return Value Type: Hex value.

PORT.MAXSIZE() Get max. size of trace buffer in records

[build 38323 - DVD 08/2012]

Syntax: **PORT.MAXSIZE()**

Returns the maximum possible trace buffer size of the port analyzer in records.

Return Value Type: Decimal value.

PORT.RECORDS() Get number of used trace records

Syntax: **PORT.RECORDS()**

Returns the number of records in the port analyzer.

Return Value Type: Decimal value.

PORT.REF() Get record number of reference record

Syntax: **PORT.REF()**

Returns the number of the selected reference record in the Port Analyzer.

Return Value Type: Decimal value.

PORT.SIZE() Get current trace buffer size in records

[build 38323 - DVD 08/2012]

Syntax: **PORT.SIZE()**

Returns the size of the Port Analyzer trace buffer to be used in trace records.

Return Value Type: Decimal value.

PORT.STATE() Get state of Port Analyzer

Syntax: **PORT.STATE()**

Returns the state of the Port Analyzer.

Return Value Type: Hex value.

Return Value and Description:

0	**Off** state
1	**Arm** state
2	**Break** state
3	**Trigger** state
6	**Slave** state

PORT.TRACK.RECORD() Get record number matching search

Syntax: **PORT.TRACK.RECORD()**

After a successful search operation, this function returns the record number. For an example, see **Analyzer.TRACK.RECORD()**.

Return Value Type: Decimal value.

PORTANALYZER()

Syntax: **PORTANALYZER()**

Returns TRUE if a port analyzer hardware is plugged.

Return Value Type: Boolean.

PORTSHARING Function

PORTSHARING() Current setting of PortSHaRing

ICD-TriCore, ICD-PCP, ICD-GTM, RH850

Syntax: **PORTSHARING()**
ETK() (deprecated)

Returns the current setting of **SYStem.CONFIG PortSHaRing**.

Return Value Type: Decimal value.

Return Value and Description:

0	OFF
1	ON
2	AUTO

POWER Functions

In This Section

See also

- hardware.POWERDEBUG()
- hardware.POWERINTEGRATOR2()
- hardware.POWERTRACE()
- hardware.POWERTRACE2LITE()
- hardware.POWERTRACEPX()
- hardware.POWERINTEGRATOR()
- hardware.POWERNEXUS()
- hardware.POWERTRACE2()
- hardware.POWERTRACE3()
- hardware.POWERTRACESERIAL()

PowerProbe Functions

In This Section

See also

- hardware.POWERPROBE()
- PROBE.GET()
- PROBE.MAXSIZE()
- PROBE.RECORD.DATA()
- PROBE.RECORD.TIME()
- PROBE.RECORDS()
- PROBE.REF()
- PROBE.SIZE()
- PROBE.STATE()
- PROBE.TRACK.RECORD()

PROBE.COUNTER.EVENT() Get value of trigger program event counter

Syntax: **PROBE.COUNTER.EVENT(***<counter_name>***)**

Returns the value of an event counter of the PowerProbe CTU.

Parameter Type: String.

Return Value Type: Decimal value.

PROBE.COUNTER.EXTERN() Get value of trigger program external counter

Syntax: **PROBE.COUNTER.EXTERN(***<counter_name>***)**

Returns the value of an extern counter of the PowerProbe CTU.

Parameter Type: String.

Return Value Type: Decimal value.

PROBE.COUNTER.TIME() Get value of trigger program time counter

Syntax: **PROBE.COUNTER.TIME(***<counter_name>***)**

Returns the value of a time counter of the PowerProbe CTU.

Parameter Type: String.

Return Value Type: Time value.

Probe.FIRST() Get record number of first trace record

Syntax: **Probe.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

PROBE.FLAG() Check state of trigger program FLAG

Syntax: **PROBE.FLAG(**<*flag_name*>**)**

Returns the value of a flag of the PowerProbe CTU.

Parameter Type: String.

Return Value Type: Boolean.

PROBE.GET() Value of channel

Syntax: **PROBE.GET(**<*channel_name*>**)**

Returns the current value of the given channel.

Parameter Type: String.

Return Value Type: Hex value.

PROBE.MAXSIZE() Get max. size of trace buffer in records

Syntax: **PROBE.MAXSIZE()**

The maximum number of records, which can be recorded by the PowerProbe (value depends on the actual selected tracing mode too).

Return Value Type: Decimal value.

PROBE.RECORD.DATA() Get data recorded in trace record

Syntax: **PROBE.RECORD.DATA(***<record_number>***,***<channel>***)**

Returns the sampled data from the specified record.

Parameter and Description:

<record_number>	**Parameter Type**: Decimal value.
<channel>	**Parameter Type**: String.

Return Value Type: Hex value.

PROBE.RECORD.TIME() Get timestamp of trace record

Syntax: **PROBE.RECORD.TIME(***<record_number>***)**

Returns the timestamp from the specified record. For an example, see **Analyzer.RECORD.TIME()**.

Parameter Type: Decimal value.

Return Value Type: Time value.

PROBE.RECORDS() Get number of used trace records

Syntax: **PROBE.RECORDS()**

Returns the number of records recorded by the PowerProbe.

Return Value Type: Decimal value.

PROBE.REF() Get record number of reference record

Syntax: **PROBE.REF()**

Returns the number of the selected reference record in the Onchip trace.

Return Value Type: Decimal value.

PROBE.SIZE() Get current trace buffer size in records

Syntax: **PROBE.SIZE()**

Returns the actual defined logical size of the PowerProbe trace buffer in records.

Return Value Type: Decimal value.

PROBE.STATE() Get state of PowerProbe

Syntax: **PROBE.STATE()**

Returns the state of the PowerProbe.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state

Syntax: **PROBE.TRACK.RECORD()**

After a successful search operation, this function returns the record number. For an example, see **Analyzer.TRACK.RECORD()**.

Return Value Type: Decimal value.

PP() Address of program pointer (access class, space ID, program counter)

Syntax: **PP()**

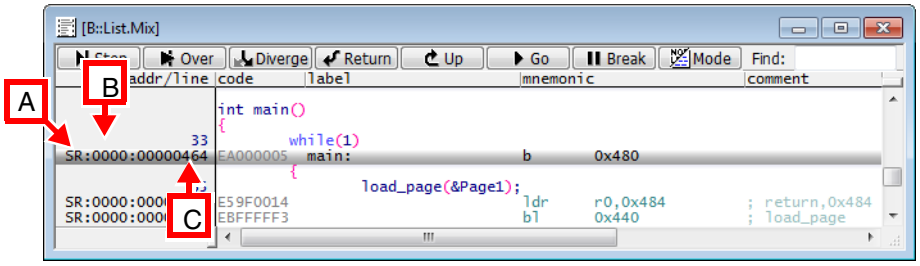
Returns the address of the *program pointer*, which consists of the access class + the CPU program counter (PC). See [**A**] and [**C**] in the example.

Additionally, the function **PP()** returns the space ID as a hex value if **SYStem.Option.MMUSPACES** is set to **ON**. See [**B**] in the example.

Return Value Type: Address.

Example:

```
PRINT PP()      ;prints to the TRACE32 message line: SR:0x0:0x464
                ;see screenshot
```



A Access class.

B Space ID.

C Content of CPU program counter (PC). See also **Register.view** window.

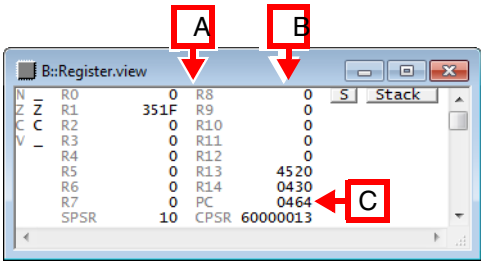
See also: **Register()**.

Register()

Content of register

Syntax: **Register(**<*register_name*> | **PP)**

Returns the content of a register.



A Register names.

B Register contents.

C Register content of the program counter (PC).

Parameter and Description:

<*register_name*>	**Parameter Type**: String. The architecture-specific register names are displayed in the **Register.view** window.
PP	**Parameter Type**: String. A TRACE32 built-in keyword that can be used to return the register content of the prograg249m counter (PC) regardless of its real <*register_name*>. That is, the keyword **PP** is an *architecture-independent register name* for the program counter.

Return Value Type: Hex value.

Example:

```
Register.view          ;opens the Register.view window

PRINT Register(R13)    ;prints the content of the register named R13 to
                       ;the TRACE32 message line

PRINT Register(PP)     ;returns the content of the program counter (PC)
                       ;- regardless of its real <register_name>
```

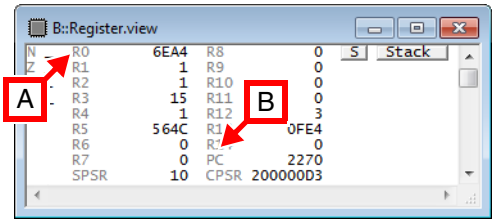
See also: **PP()**.

Syntax: **Register.LIST("" | "*<register_name>*")**

Returns the first / next valid register name.

Parameter and Description: You can pass an empty string as a register name or a real register name.

Empty string	**Parameter Type**: String. Passing an empty string returns the *first* register name. For example, the register name **`R0`** for the ARM architecture. See [**A**] in figure below.
<register_name>	**Parameter Type**: String. Passing a real register name returns the name of the *next* register. For example, if you pass **`PC`** as register name, the function returns **`CPSR`** as the next register name for the ARM architecture. See [**B**] in figure below.



Return Value Type: String.

Example: The function **Register.LIST()** is used to loop through all register names and print them to the **AREA.view** window.

```
Register.view
AREA.view

&reg=Register.LIST("")                ;returns the first register name
PRINT "&reg"                          ;and prints it to the AREA window

WHILE "&reg">""                       ;loop through all register names
(
    PRINT Register.LIST("&reg")       ;print register name
    &reg=Register.LIST("&reg")        ;get next register name
)
```

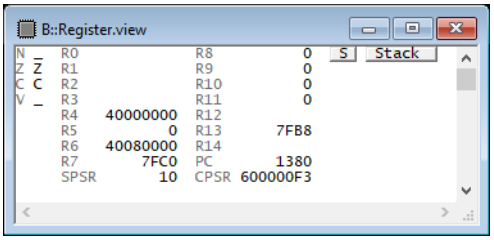
Syntax: **Register.Valid(**<*register_name*>**)**

Returns TRUE if the register value is available.

Parameter and Description:

<*register_name*>	**Parameter Type**: String. The architecture-specific register names are displayed in the **Register.view** window.

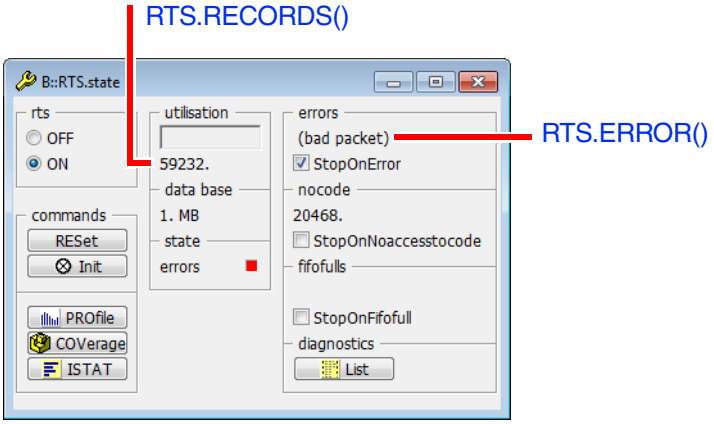
Return Value Type: Boolean.




```
PRINT Register.Valid(R0)        ; returns FALSE as the R0 value is not
                                ; available
PRINT Register.Valid(R5)        ; returns TRUE
```

RTS Functions

This figure provides an overview of the return values the RTS functions. For descriptions of the illustrated functions, see below.



In This Section

See also

- ■ RTS
- ❑ RTS.BUSY()
- ❑ RTS.ERROR()
- ❑ RTS.FIFOFULL()
- ❑ RTS.NOCODE()
- ❑ RTS.RECORD()
- ❑ RTS.RECORDS()

RTS.ERROR() Check for flowerrors during RTS processing

[Go to figure]

Syntax: **RTS.ERROR()**

Returns TRUE if **flowerrors** where detected during RTS processing.

Return Value Type: Boolean.

RTS.NOCODE() Check for RTS NOCODE error

[build 78366 - DVD 02/2017]

Syntax: **RTS.NOCODE()**

Returns TRUE if code was missing in VM for decoding.

Return Value Type: Boolean.

RTS.FIFOFULL() Check for FIFO full error in RTS

[build 78366 - DVD 02/2017]]

Syntax: **RTS.FIFOFULL()**

Returns TRUE if the target trace FIFO buffer overflowed during the recording.

Return Value Type: Boolean.

RTS.RECORD() Find record causing an error in RTS

[build 78366 - DVD 02/2017]

Syntax: **RTS.RECORD()**

Returns the record number of the error that stopped RTS.

Return Value Type: Decimal value.

RTS.RECORDS() Get number of trace records transferred to RTS

[Go to figure [build 47027 - DVD 08/2013]

Syntax: **RTS.RECORDS()**

Returns the number of trace records transferred to the host for RTS processing.

Return Value Type: Decimal value.

RTS.BUSY() Check if RTS is busy

[build 115124 - DVD 02/2020]

Syntax: **RTS.BUSY()**

Returns TRUE when RTS trace is off and processing is catching up.

Return Value Type: Boolean.

RunTime Functions

In This Section

See also

- RunTime
- RunTime.ACCURACY()
- RunTime.ACTUAL()
- RunTime.LAST()
- RunTime.LASTRUN()
- RunTime.REFA()
- RunTime.REFB()

RunTime.ACCURACY() Accuracy of run-time counter

Syntax: **RunTime.ACCURACY()**

Returns the measurement error of the RunTime counter in seconds.

Return Value Type: Time value.

RunTime.ACTUAL()

Syntax: **RunTime.ACTUAL()**

Returns the value displayed in the **actual** column of the **RunTime.state** window (as time from zero).

Return Value Type: Time value.

RunTime.LAST()

Syntax: **RunTime.LAST()**

Returns the value displayed in the **laststart** column of the **RunTime.state** window (as time from zero).

Return Value Type: Time value.

RunTime.LASTRUN()

Syntax: **RunTime.LASTRUN()**

Returns the time of the last single step or the time between the last go and break of the program.

Return Value Type: Time value.

RunTime.REFA()

Syntax: **RunTime.REFA()**

Returns the value displayed in the **ref A** column of the **RunTime.state** window (as time from zero).

Return Value Type: Time value.

RunTime.REFB()

Syntax: **RunTime.REFB()**

Returns the value displayed in the **ref B** column of the **RunTime.state** window (as time from zero).

Return Value Type: Time value.

SMMU.BaseADDRESS() Base address of SMMU

ARM, ARMv8-A [build 64463 - DVD 09/2015]

Syntax: **SMMU.BaseADDRESS("***<smmu_name>***")**

Returns the base address of hardware system MMU (SMMU). SMMUs are created with the **SMMU.ADD** command.

Parameter Type: String.

Return Value Type: Address.

Example:

```
;define a new SMMU named "myGPU" for a graphics processing unit
;         <smmu_name>             <base_address>
SMMU.ADD    "myGPU"     MMU500     A:0x50000000

;returns AZSD:0x0:0x50000000 as <base_address>
PRINT SMMU.BaseADDRESS("myGPU)
```

SMMU.StreamID2SMRG() Find match for stream ID

ARM, ARMv8-A [build 64463 - DVD 09/2015] [Example]

Syntax: **SMMU.StreamID2SMRG("***<name>***",***<stream_id>***)**

Finds a matching stream mapping register group (SMRG) for a given *<stream_id>* using the SMMU stream ID matching algorithm.

Parameter and Description:

<name>	**Parameter Type**: String. Specifies the SMMU to be searched. The SMMU *<name>* must be quoted.
<stream_id>	**Parameter Type**: Decimal or hex or binary value. Stream ID of a memory transaction stream.

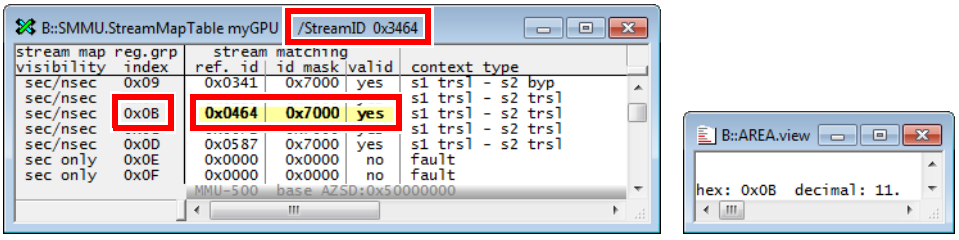
Return Value and Description:

Range: 0 to (max. number of index entries-1).	**Return Value Type**: Decimal value. If exactly one matching SMRG was found, the function returns the index of the matching SMRG. See also column **index** in the **SMMU.StreamMapTable** window.
-1	**Return Value Type**: Decimal value. If no matching SMRG was found for the given *<stream_id>*, -1 will be returned.
-2	**Return Value Type**: Decimal value. If more than one matching SMRG was found for the given *<stream_id>*, -2 will be returned. Additionally, a warning message will be printed to the AREA window. **NOTE**: If more than one SMRG matches a given *<stream_id>*, a stream matching fault occurs in the SMMU hardware.

Example: This PRACTICE script opens the **SMMU.StreamMapTable** window, searches for the *<stream_id>* 0x3464, and highlights the matching SMRG 0x0464 in yellow.

For the SMRG 0x0464, the function **SMMU.StreamID2SMRG()** returns the index 11 (decimal).

```
;open the window and highlight the matching SMRG in yellow
SMMU.StreamMapTable myGPU /StreamID 0x3464

;return the index of the SMRG as a decimal value
&index=SMMU.StreamID2SMRG("myGPU",0x3464)

;print the index as hex and decimal to the AREA window
PRINT "hex: 0x" CONVert.INTTOHEX(&index)   "   decimal: &index"
AREA.view
```

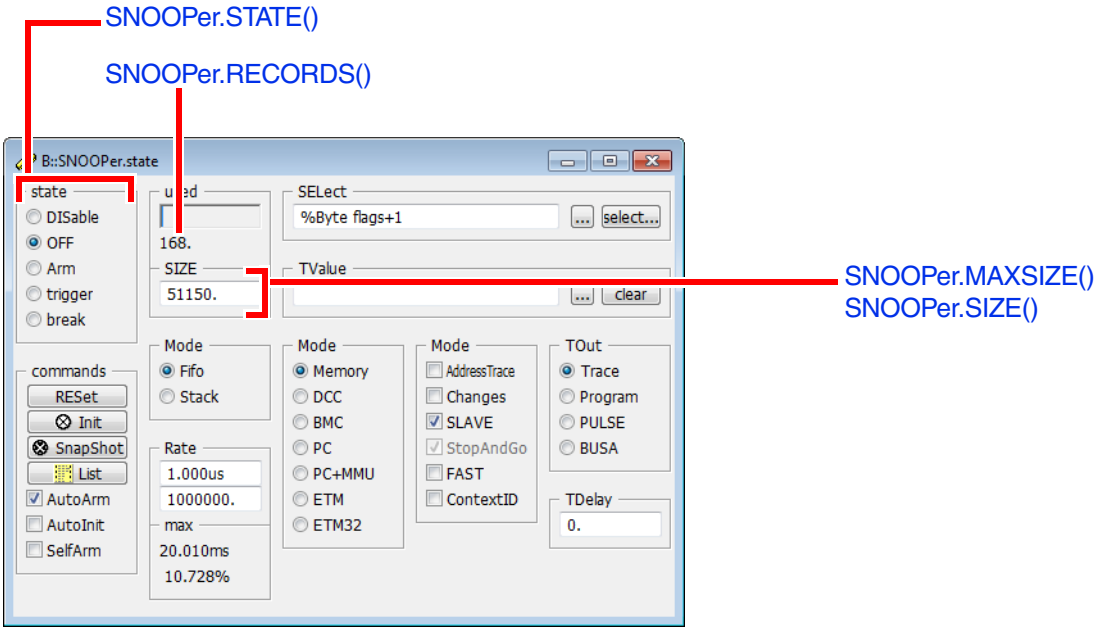


NOTE:	At first glance, the **StreamID** 0x**3**464 does not seem to match the SMRG 0x**0**464. However, if you take the ID mask 0x7000 (= 0y0111_0000_0000_0000) into account, the match is correct.

The row highlighted in yellow in the **SMMU.StreamMapTable** window is a correct match for the **StreamID** 0x3464 we searched for.

SNOOPer Functions

This figure provides an overview of the return values of some of the SNOOPer functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- SNOOPer.FIRST()
- SNOOPer.RECORD.ADDRESS()
- SNOOPer.RECORD.OFFSET()
- SNOOPer.RECORDS()
- SNOOPer.SIZE()
- SNOOPer.MAXSIZE()
- SNOOPer.RECORD.DATA()
- SNOOPer.RECORD.TIME()
- SNOOPer.REF()
- SNOOPer.STATE()

SNOOPer.FIRST() Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **SNOOPer.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

SNOOPer.MAXSIZE() Get max. size of trace buffer in records

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **SNOOPer.MAXSIZE()**

Returns the maximum possible number of records.

Return Value Type: Decimal value.

SNOOPer.RECORD.ADDRESS() Get address recorded in trace record

[build 38764]

Syntax: **SNOOPer.RECORD.ADDRESS(**<*record_number*>**)**

Returns the sampled address (access class and offset) from the specified record.

Parameter Type: Decimal value.

Return Value Type: Address.

SNOOPer.RECORD.DATA() Get data recorded in trace record

[build 38764]

Syntax: **SNOOPer.RECORD.DATA(**<*record_number*>**)**

Returns the sampled data of the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

SNOOPer.RECORD.OFFSET() Get address in trace record as number

[build 38764]

Syntax: **SNOOPer.RECORD.OFFSET(**<*record_number*>**)**

Returns the address-offset of the sampled address from the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

SNOOPer.RECORD.TIME() Get timestamp of trace record

[build 38764]

Syntax: **SNOOPer.RECORD.TIME(**<*record_number*>**)**

Returns the timestamp of the specified record. For an example, see **Analyzer.RECORD.TIME()**.

Parameter Type: Decimal value.

Return Value Type: Time value.

SNOOPer.RECORDS() Get number of used trace records

[Go to figure]

Syntax: **SNOOPer.RECORDS()**

The number of records currently recorded in the SNOOPer trace buffer.

Return Value Type: Decimal value.

SNOOPer.REF() Get record number of reference record

Syntax: **SNOOPer.REF()**

The number of the selected reference record in the SNOOPer trace.

Return Value Type: Decimal value.

SNOOPer.SIZE() Get current trace buffer size in records

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **SNOOPer.SIZE()**

Returns the currently defined size of the SNOOPer trace buffer in records.

Return Value Type: Decimal value.

[Go to figure]

Syntax: **SNOOPer.STATE()**

Returns the state of the SNOOPer trace.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state

STATE Functions (Target State)

In This Section

See also

- STATE.HALT()
- STATE.NOCPUACCESS()
- STATE.NOCTIACCESS()
- STATE.OSLK()
- STATE.POWER()
- STATE.PROCESSOR()
- STATE.RESET()
- STATE.RUN()
- STATE.TARGET()

STATE.HALT()

Syntax: **STATE.HALT()**

Returns the state of the “halt” display (i.e. no CPU cycles).

Return Value Type: Boolean.

STATE.OSLK()

32-bit and 64-bit ARM cores [build 67684 - DVD 02/2016]

Syntax: **STATE.OSLK()**

Returns the current state of the OS-lock bit of the CPU.

Return Value Type: Boolean.

Return Value and Description:

TRUE	The CPU is running and OS-lock bit is set.
FALSE	FALSE can mean either case 1 or case 2: • Case 1: The CPU is running and OS-lock bit is cleared. • Case 2: The debugger has connected to the CPU and detected that the CPU has stopped.

STATE.POWER()

[Example]

Syntax: **STATE.POWER()**

Returns the state of the target power line.

Return Value Type: Boolean.

Return Value and Description:

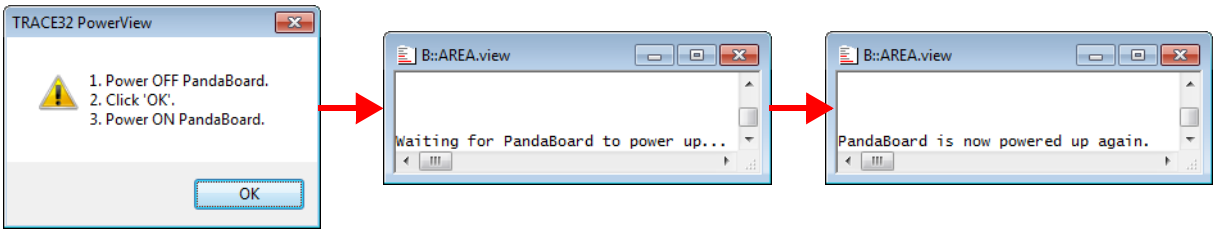
TRUE	Power is applied.
FALSE	Power is not applied.

NOTE:	Please check the power-off voltage if the power-off state cannot be detected. Especially for low-power applications, the debug/trace-signal current can be enough to power the application: Target power does not drop to GND level -> power-off cannot be detected.

Example

In the following example, the message box is displayed as long as power is applied to the target board. The message box disappears only if these conditions are met:

- You have unplugged all connectors from the target board.
- You have clicked **OK**.




```
LOCAL &msgA &msgB &msgC ; PRACTICE macros for the message texts A, B, C

&msgA="1. Power OFF PandaBoard."+CONV.CHAR(0x0d)+\
"2. Click 'OK'."+CONV.CHAR(0x0d)+"3. Power ON PandaBoard."

&msgB="Waiting for PandaBoard to power up..."

&msgC="PandaBoard is now powered up again."

AREA.view               ; Opens the AREA.view window.

WHILE STATE.POWER()     ; While power is on...
    DIALOG.OK "&msgA"   ; ...prompt the user to power off the board.

PRINT "&msgB"           ; Now prompt the user to power up the board again.

WAIT STATE.POWER()      ; Wait for the user to respond as requested.

AREA.Clear ; Clear the previous message from the AREA.view window.

PRINT "&msgC"           ; Display success message.
```

+CONV.CHAR(0x0d)+ creates a line break on the GUI. A backslash **** is used as a line continuation character in PRACTICE script files (*.cmm). No white space permitted after the backslash.

STATE.PROCESSOR()

Syntax: **STATE.PROCESSOR()**

Returns the name of the processor, which was selected with the command **SYStem.CPU**. This function is an alias for **SYStem.CPU()**.

Return Value Type: String.

STATE.RESET()

Syntax: **STATE.RESET()**

Returns the state of the target reset line.

Return Value Type: Boolean.

STATE.RUN()

Syntax: **STATE.RUN()**
RUN() (deprecated)

Returns the state of the run-flag (CPU running in target).

Return Value Type: Boolean.

STATE.TARGET() State of target displayed in TRACE32 state line

Syntax: **STATE.TARGET()**

Returns the message about the target state displayed in the state line. For example, `system down`, `system ready`, `running`, `stopped`.

Return Value Type: String.

SPE Function

SPE() Content from SPE register

Syntax: **SPE(***<register_name>***)**

Returns the content of the selected SPE register. See also **SPE** command group.

Parameter Type: String.

Return Value Type: Hex value.

SSE() — Segment from SSE register

Syntax: **SSE(***<register_name>.<column_number>***)**

Returns a 32-bit segment of the selected 128-bit SSE register. See also **SSE** command group.

Parameter Type: String.

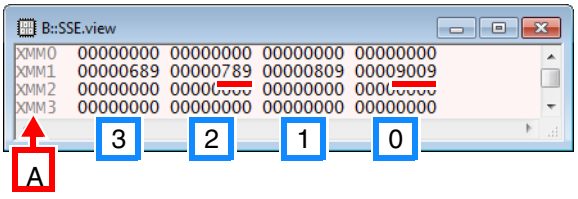
Parameter and Description:

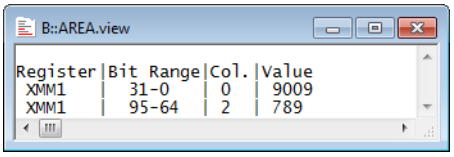
<register_name>	The register names are listed in the **SSE.view** window.
<column_number>	The column numbers start at 0 and are read from right to left in the **SSE.view** window. See example below. A column `k` corresponds to the bit range `(k · 32 + 31)-(k · 32)`, where `0<=k<=3`

Return Value Type: Hex value.

Example: This demo script returns 32-bit values of the register **XMM1** from the column **0** (bits 31 to 0) and the column **2** (bits 95 to 64) of the **SSE.view** window.

```
SSE.view
SSE.Set XMM1 689  789  809  9009
PRINT "Register|Bit Range|Col.|Value"
PRINT " XMM1    |  31-0    | 0   | " SSE(XMM1.0) ;32-bit value from col. 0
PRINT " XMM1    |  95-64   | 2   | " SSE(XMM1.2) ;32-bit value from col. 2
```





A Register names.

0 - 3 Column numbers - from right to left - in the **SSE.view** window.

hardware.STG() TRUE if Stimuli Generator hardware

Syntax: **hardware.STG()**
STG() (deprecated)

Returns TRUE if Stimuli Generator hardware is available.

Return Value Type: Boolean.

sYmbol Functions

In This Section

See also

- sYmbol
- sYmbol.AutoLOAD.CHECKCMD()
- sYmbol.BEGIN()
- sYmbol.END()
- sYmbol.EXIST()
- sYmbol.FUNCTION()
- sYmbol.ISFUNCTION()
- sYmbol.LANGUAGE()
- sYmbol.LIST.SOURCE()
- sYmbol.NAME()
- sYmbol.NEXT.BEGIN()
- sYmbol.SEARCHFILE()
- sYmbol.SECEND()
- sYmbol.SECNAME()
- sYmbol.SECRANGE()
- sYmbol.SOURCEFILE()
- sYmbol.SOURCEPATH()
- sYmbol.TRANSPOSE()
- sYmbol.VARNAME()
- sYmbol.AutoLOAD.CHECK()
- sYmbol.AutoLOAD.CONFIG()
- sYmbol.COUNT()
- sYmbol.EPILOG()
- sYmbol.EXIT()
- sYmbol.IMPORT()
- sYmbol.ISVARIABLE()
- sYmbol.LIST.PROGRAM()
- sYmbol.MATCHES()
- sYmbol.NAME.AT()
- sYmbol.RANGE()
- sYmbol.SECADDRESS()
- sYmbol.SECEXIST()
- sYmbol.SECPRANGE()
- sYmbol.SIZEOF()
- sYmbol.SOURCELINE()
- sYmbol.STATE()
- sYmbol.TYPE()

sYmbol.AutoLOAD.CHECK() Update option for the symbol autoloader

[build 71364 - DVD 09/2016]

Syntax: **sYmbol.AutoLOAD.CHECK()**

Returns the update option of the symbol autoloader that was given with **sYmbol.AutoLOAD.CHECK**.

Return Value Type: String. Returns an empty string if no automatic update is configured.

sYmbol.AutoLOAD.CHECKCMD() Load command for symbol autoloader

[build 71364 - DVD 09/2016]

Syntax: **sYmbol.AutoLOAD.CHECKCMD()**

Returns the command that was specified to be used to load a symbol file with the symbol autoloader (e.g. as parameter to **sYmbol.AutoLOAD.CHECKCoMmanD**).

Return Value Type: String. Returns an empty string if the symbol autoloader is not configured.

sYmbol.AutoLOAD.CONFIG() Used sub-command

[build 71364 - DVD 09/2016]

Syntax: **sYmbol.AutoLOAD.CONFIG()**

Returns the sub-command that was used to configure the symbol autoloader. E.g. “CHECKCoMmanD” if **sYmbol.AutoLOAD.CHECKCoMmanD** was used.

Return Value Type: String. Returns an empty string if the symbol autoloader is not configured.

sYmbol.BEGIN() First address of symbol

[build 18430 - DVD 12/2009]

Syntax: **sYmbol.BEGIN(***<symbol>***)**

Returns the first address occupied by the symbol. A symbol name used alone represent the beginning address too. Function only implemented to have a counterpart of function the **sYmbol.END()**.

Parameter Type: Symbol.

Return Value Type: Address.

Example:

```
Data.Print sYmbol.BEGIN(vbfield)
Data.Print vbfield                // displays the identical value
```

sYmbol.COUNT() Number of symbols

Syntax: **sYmbol.COUNT(**<*symbol*>**)**

Returns the number of symbols defined with the specified name. The wildcards '*****' and '**?**' are supported.

Parameter Type: Symbol.

Return Value Type: Decimal value.

Examples:

```
ECHO sYmbol.COUNT(func*)
ECHO sYmbol.COUNT(func2?)
```

sYmbol.END() Last address of symbol

Syntax: **sYmbol.END(**<*symbol*>**)**

Returns the last address occupied by the symbol.

Parameter Type: Symbol.

Return Value Type: Address.

Example:

```
Data.Print sYmbol.END(vbfield)
```

Syntax: **sYmbol.EPILOG(**<*symbol*>**)**

Returns the last address of the specified function where the local variables are still valid, i.e. the address of the return point. Prerequisite: The function has *exactly one* return point.

Parameter Type: Symbol.

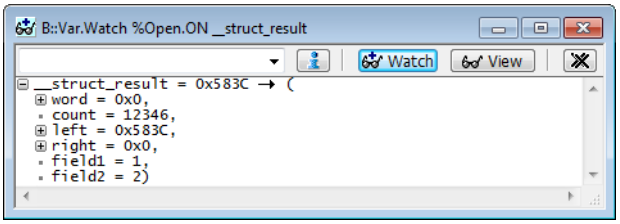
Return Value Type: Address.

Example 1: The function **sYmbol.EPILOG()** is used to display the address of the return point in the **Data.Print** window.

```
List.auto func1                     ;optional step: display the function
Go.Return                           ;optional step: go to the address of
                                    ;the return point
Data.Print sYmbol.EPILOG(func1)     ;get the address of the return point
                                    ;and display it in the Data.Print
                                    ;window
```

Example 2: The function **sYmbol.EPILOG()** is used to set a breakpoint to the return point in order to watch the local variable `__struct_result` in the **Var.Watch** window.

```
Break.Set sYmbol.EPILOG(func4)      ;set a breakpoint to the return point
                                    ;address of 'func4'
Go                                  ;start the program execution
WAIT !STATE.RUN()                   ;wait until program stops

Var.Watch %Open.ON __struct_result //watch local variable '__struct_re.'
```



sYmbol.EXIST() TRUE if symbol exists

Syntax: **sYmbol.EXIST(**<*symbol*>**)**

Returns TRUE if the symbol exists in the TRACE32 symbol database or FALSE otherwise.

Parameter Type: Symbol.

Return Value Type: Boolean.

Example 1:

```
PRINT sYmbol.EXIST(vbfield)
PRINT sYmbol.EXIST(qwertzuiop)
```

Example 2: The **sYmbol.EXIST()** function can be used together with the **Trace.Find** command in a PRACTICE script. This prevents the PRACTICE script from stopping if the searched symbol does not exist, and eliminates the need for an error handler.

```
IF sYmbol.EXIST(qwertzuiop)
(
   Trace.Find , Address qwertzuiop
   IF FOUND()
       PRINT "Record number of 'qwertzuiop' is: " Analyzer.TRACK.RECORD()
)
ELSE
(
   PRINT "The symbol 'qwertzuiop' does not exist in the symbol database."
)
```

sYmbol.EXIT() Exit address of function

Syntax: **sYmbol.EXIT(**<*symbol*>**)**

Returns the exit address of the specified function.

Parameter Type: Symbol.

Return Value Type: Address.

Example:

```
Data.Print sYmbol.EXIT(func40)
PRINT ADDRESS.OFFSET(sYmbol.EXIT(func40)))
```

sYmbol.FUNCTION() Function name

Syntax: **sYmbol.FUNCTION(**<*address*>**)**

Returns the path and name of the function that includes the specified address. The address has to be classified, e.g. **P**:0x200

Parameter Type: Address.

Return Value Type: String.

Example:

```
PRINT sYmbol.FUNCTION(P:0x40001000)
```

sYmbol.IMPORT() Import file names

Syntax: **sYmbol.IMPORT()**

Returns the name of the next not yet processed import file name. This function can be used to walk through the list of import names (Only relevant for *.exe target programs, on Windows CE for example).

Return Value Type: String.

sYmbol.ISFUNCTION() TRUE if symbol is function

[build 135097 - DVD 09/2021]

Syntax: **sYmbol.ISFUNCTION(**<*symbol*>**)**

Returns TRUE if the selected symbol is function.

Parameter Type: Symbol.

Return Value Type: Boolean.

Example:

```
Print sYmbol.ISFUNCTION(func1)                ; return TRUE
```

Syntax: **sYmbol.ISVARIABLE(***<symbol>***)**

Returns TRUE if the selected symbol is variable.

Parameter Type: Symbol.

Return Value Type: Boolean.

sYmbol.LANGUAGE() Selected high-level language

[build 122228 - DVD 09/2020]

Syntax: **sYmbol.LANGUAGE()**

Returns the currently selected language for high-level expressions.

Return Value Type: String.

sYmbol.List.MAP.<x>() Information about address ranges on the target

[build 86990 - DVD 09/2017]

Syntax: **sYmbol.List.MAP.**<*x*>**()**

<*x*>: **COUNT** | **BEGIN** | **END** | **RANGE**

(Full function name only required for HELP.Index):
sYmbol.List.MAP.COUNT()
sYmbol.List.MAP.BEGIN(<*index*>**)**
sYmbol.List.MAP.END(<*index*>**)**
sYmbol.List.MAP.RANGE(<*index*>**)**

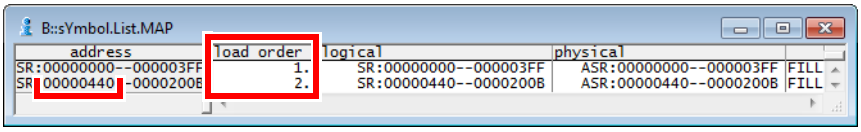
<x>	Description of sYmbol.List.MAP.<x>()
COUNT	Returns the number of address ranges loaded to the target with the **Data.LOAD.*** <*file*> commands. **Return Value Type**: Decimal value.
BEGIN(<*index*>**)**	Returns the start address of the range <*index*>. **Parameter Type**: Decimal value. **Return Value Type**: Address.
END(<*index*>**)**	Returns the end address of the range <*index*>. **Parameter Type**: Decimal value. **Return Value Type**: Address.
RANGE(<*index*>**)**	Returns the range <*index*>. **Parameter Type**: Decimal value. **Return Value Type**: Address range.

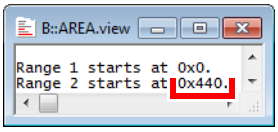
Parameter and Description:

<*index*>	The individual <*index*> numbers of the ranges are listed in the **load order** column of the **sYmbol.List.MAP** window. To return the highest <*index*> number, use **sYmbol.List.MAP.COUNT()**.

Example:

```
SYStem.Up  ;connect to target
Data.LOAD.Elf  sieve_flash_thumb_ii_v7m.elf  ;load application to target
&i=1.
RePeaT sYmbol.List.MAP.COUNT() ;loop through number of address ranges
( ;let's use PRINTF to format the output to the AREA.view window
  PRINTF "%s%i%s"  "Range " &i " starts at "
  PRINTF %CONTinue "0x%x%s" sYmbol.List.MAP.BEGIN(&i) "."
  &i=&i+1.
)
```





sYmbol.LIST.PROGRAM() Path and file name of binary files

[build 42354 - DVD 02/2013]

Syntax: **sYmbol.LIST.PROGRAM(1 | 0)**

Returns the path and file name of a loaded target binary, e.g. an ELF file.

Parameter and Description:

1	**Parameter Type**: Decimal or hex value. Starts with the first target binary listed in the **sYmbol.List.Program** window.
0	**Parameter Type**: Decimal or hex value. Continues with the next target binary in the list. To return all loaded target binaries, call the function repeatedly until an empty string is returned.

Return Value Type: String.

Syntax:	**sYmbol.List.PROGRAM.**<*x*>**()**
<*x*>:	**COUNT** | **COMMAND** | **FORMAT** | **FILE** | **NAME** | **RANGE**
(Full function name only required for HELP.Index):	**sYmbol.List.PROGRAM.COUNT()** **sYmbol.List.PROGRAM.COMMAND(**<*index*>**)** **sYmbol.List.PROGRAM.FORMAT(**<*index*>**)** **sYmbol.List.PROGRAM.FILE(**<*index*>**)** **sYmbol.List.PROGRAM.NAME(**<*index*>**)** **sYmbol.List.PROGRAM.RANGE(**<*index*>**)**

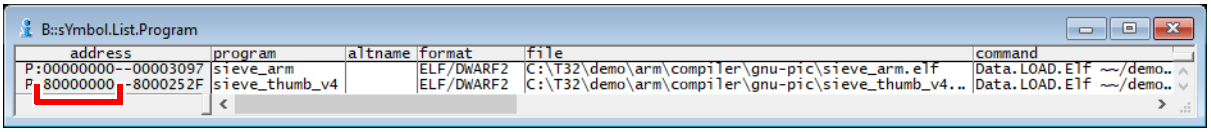
<x>	Description of sYmbol.List.PROGRAM.<x>()
COUNT	Returns the number of programs loaded to the symbol table with the **Data.LOAD.*** <*file*> commands. **Return Value Type**: Decimal value.
COMMAND(<*index*>**)**	Returns the commandline how <*index*> got added to the symbol database. **Parameter Type**: Decimal value. **Return Value Type**: String.
FORMAT(<*index*>**)**	Returns the format of program <*index*>. **Parameter Type**: Decimal value. **Return Value Type**: String.
FILE(<*index*>**)**	Returns the path of file <*index*>. **Parameter Type**: Decimal value. **Return Value Type**: String.
NAME(<*index*>**)**	Returns the symbol database name of <*index*>. **Parameter Type**: Decimal value. **Return Value Type**: String.
RANGE(<*index*>**)**	Returns the range <*index*>. **Parameter Type**: Decimal value. **Return Value Type**: Address range.

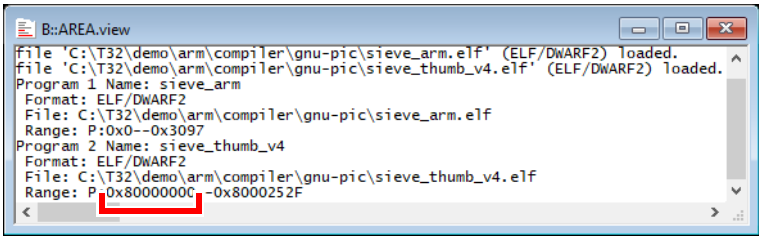
Parameter and Description:

<*index*>	The individual <*index*> numbers address the lines in the **sYmbol.List.Program** window. To return the highest <*index*> number, use **sYmbol.List.PROGRAM.COUNT()**.

Example:

```
; load file to symbol database
Data.LOAD.Elf ~~/demo/arm/compiler/gnu-pic/sieve_arm.elf /NoCODE
Data.LOAD.Elf ~~/demo/arm/compiler/gnu-pic/sieve_thumb_v4.elf \
                                                0x80000000 /NoCODE /NoClear
&i=1.
RePeaT sYmbol.List.PROGRAM.COUNT() ; loop through number of programs
( ; lets use PRINTF to format output to AREA.view window
  PRINTF "Program %i Name: %s" &i sYmbol.List.PROGRAM.NAME(&i)
  PRINTF " Format: %s" sYmbol.List.PROGRAM.FORMAT(&i)
  PRINTF " File: %s" sYmbol.List.PROGRAM.FILE(&i)
  PRINTF " Range: %#!R" sYmbol.List.Program.RANGE(&i)
  &i=&i+1.
)
```





sYmbol.List.SECtion.<x>() Information about section ranges

[build 141685 - DVD 02/2022]

Syntax:	**sYmbol.List.SECtion.**<*x*>**()**
<*x*>:	**COUNT** | **PATH** | **RANGE**
(Full function name only required for HELP.Index):	**sYmbol.List.SECtion.COUNT()** **sYmbol.List.SECtion.PATH(**<*index*>**)** **sYmbol.List.SECtion.RANGE(**<*index*>**)**

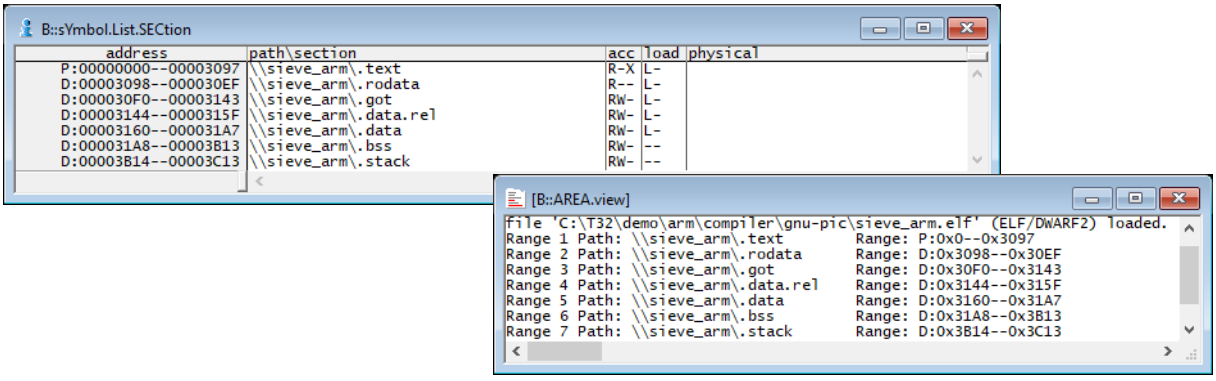
<x>	Description of sYmbol.List.PROGRAM.<x>()
COUNT	Returns the number of sections loaded to the symbol table with the **Data.LOAD.*** *<file>* commands. **Return Value Type**: Decimal value.
PATH(*<index>***)**	Returns the symbol path *<index>*. **Parameter Type**: Decimal value. **Return Value Type**: String.
RANGE(*<index>***)**	Returns the range *<index>*. **Parameter Type**: Decimal value. **Return Value Type**: Address range.

Parameter and Description:

<index>	The individual *<index>* numbers address the lines in the **sYmbol.List.SECtion** window. To return the highest *<index>* number, use **sYmbol.List.SECtion.COUNT()**.

Example:

```
; load file to symbol database
Data.LOAD.Elf ~~/demo/arm/compiler/gnu-pic/sieve_arm.elf /NoCODE
&i=1.
RePeaT sYmbol.List.SECtion.COUNT() ; loop through number of sections
( ; lets use PRINTF to format output to AREA.view window
  PRINTF "Range %i Path: %-24s Range: %#!R" &i \
              sYmbol.List.SECtion.PATH(&i) sYmbol.List.SECtion.RANGE(&i)
  &i=&i+1.
)
```



Syntax: **sYmbol.LIST.SOURCE(***<start_over>*,*<filter>*,*<refresh_source_list>***)**

Via the TRACE32 Remote API, 3rd-party tools can use the **sYmbol.LIST.SOURCE()** together with the **T32Cmd()** interface function to request the file location of all source files, e.g. *.c files, used to build the currently loaded target binary, e.g. an ELF file.

Because a target binary consists of more than one source file, it is necessary to call the **sYmbol.LIST.SOURCE()** function repeatedly until an empty string is returned.

Parameter and Description:

<start_over>	**Parameter Type**: Decimal or hex value. • **1**: Starts with the first source file listed in the **sYmbol.List.SOURCE** window, e.g. a *.c. file. • **0**: Continue with the next source file in the list.
<filter>	**Parameter Type**: Decimal or hex value. • **0**: Returns the file path of all files referenced in the symbol file, regardless if found on the local machine or not. • **1**: Returns the file location of all files that are currently loaded by TRACE32 PowerView. This implies that the files are found on the local machine. • **2**: Returns the file path of files that are not found by TRACE32 PowerView. • **4**: Returns the file path of files referenced in the symbol file, but do not contain any sourcecode with debug information. Such files will never show up in the **List** window.
<refresh_source_list>	**Parameter Type**: Decimal or hex value. • **1**: Tells TRACE32 PowerView to refresh the internal source file list before returning file locations. • **0**: Does not refresh the internal source file list, i.e. files not opened in a **Data.List** window will be unknown / missing.

Return Value Type: String.

sYmbol.MATCHES() Number of occurrences

Syntax: **sYmbol.MATCHES()**

Returns the number of hits of a preceding command like **sYmbol.Browse** or **sYmbol.ForEach**.

Return Value Type: Decimal value.

Example:

```
sYmbol.Browse func*0
PRINT sYmbol.MATCHES()
ENDDO
```

sYmbol.NAME() Symbol path and name based on address

Syntax: **sYmbol.NAME(***<address>***)**

Returns the symbol path and name of a specific address. The address has to be classified, e.g. **D**:0x200. If the address is within the body of a function, the result is the same as returned by **sYmbol.FUNCTION()**.

Parameter Type: Address.

Return Value Type: String.

Example:

```
PRINT sYmbol.NAME(D:0x40005EB8)
```

sYmbol.NAME.AT() Resolve ambiguous symbols based on address

[build 54072 - DVD 09/2014]

Syntax: **sYmbol.NAME.AT(***<address>***,***<context_address>***)**

Returns the symbol path and name of an *<address>*.

The function tries to find the symbol in the symbol database which fits best to the specified context address. Address, space ID, and access class of the specific context address are used as filter criteria to find the best match.

The function is useful if the same symbol name exists more than once in the symbol database. The function allows to find the symbol which belongs to a specific program address range or a specific task (space ID). In addition in ARM architectures, the function finds the symbol which belongs to a specific ARM Zone (non-secure, secure, hypervisor, see **SYStem.Option.ZoneSPACES**).

Parameter and Description:

<address>	**Parameter Type**: String.
<context_address>	**Parameter Type**: Address.

Return Value Type: String. Returns an empty string if no matching symbol is found.

Examples:

```
PRINT sYmbol.NAME.AT(sieve,D:0x00209860)
PRINT sYmbol.NAME.AT(main,D:0x02A3:0x0)
PRINT sYmbol.NAME.AT(__per_cpu_offset,H:0x0)
PRINT sYmbol.NAME.AT(__per_cpu_offset,N:0x0)
```

sYmbol.NEXT.BEGIN() Start address of next symbol

[build 33011 - DVD 02/2012]

Syntax: **sYmbol.NEXT.BEGIN(**<*symbol*>**)**

Returns the start address of the next symbol.

Parameter Type: Symbol.

Return Value Type: Address.

sYmbol.RANGE() Address range of symbol

Syntax: **sYmbol.RANGE(**<*symbol*>**)**

Returns the address range occupied by the symbol (e.g. function or variable).

Parameter Type: Symbol.

Return Value Type: Address range.

Example:

```
Data.Print sYmbol.RANGE(flags)
```

sYmbol.SEARCHFILE() Absolute path of source file

Syntax: **sYmbol.SEARCHFILE(**<*file*>**)**

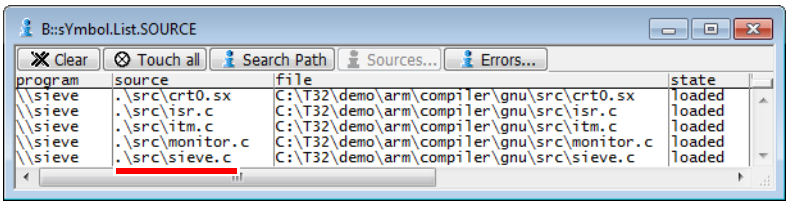
Walks through the search path for source files and returns the absolute path of the first found file.

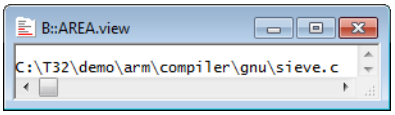
Parameter Type: String.

Return Value Type: String. If no file is found the function returns an empty string.

Example:

```
PRINT sYmbol.SEARCHFILE(demo.c)
PRINT sYmbol.SEARCHFILE(.src\sieve.c)
```





sYmbol.SECADDRESS() Start address of section

Syntax: **sYmbol.SECADDRESS(**<*section*>**)**

Returns the logical start address of the named section.

Parameter Type: String.

Return Value Type: Address.

sYmbol.SECEND() End address of section

Syntax: **sYmbol.SECEND(**<*section*>**)**

Returns the logical end address of the named section.

Parameter Type: String.

Return Value Type: Address.

sYmbol.SECEXIST() Check for existence of a section

[build 115086 - DVD 02/2020]

Syntax: **sYmbol.SECEXIST(**<*section*>**)**

Returns TRUE if the named section exists.

Parameter Type: String

Return Value Type: Boolean.

sYmbol.SECNAME() Section name

Syntax: **sYmbol.SECNAME(**<*address*>**)**

Returns the name of the section at the specified address.

Parameter Type: Address.

Return Value Type: String.

See also: **sYmbol.List.SECtion** window.

sYmbol.SECPRANGE() Physical address range of section

Syntax: **sYmbol.SECPRANGE(**<*section*>**)**

Returns the physical address range occupied by the named section.

Parameter Type: String.

Return Value Type: Address range.

sYmbol.SECRANGE() Logical address range of section

Syntax: **sYmbol.SECRANGE(**<*section*>**)**

Returns the logical address range occupied by the named section.

Parameter Type: String.

Return Value Type: Address range.

sYmbol.SIZEOF() Size of debug symbol

Syntax: **sYmbol.SIZEOF(***<symbol>***)**

Returns the size occupied by the specified debug symbol in memory (e.g. function, variable, module).

Parameter Type: Symbol.

Return Value Type: Hex value.

Example:

```
Data.Print flags++sYmbol.SIZEOF(flags)

Data.Find  flags++sYmbol.SIZEOF(flags) 0x00
IF FOUND()
(
     Data.Set   TRACK.ADDRESS() 0xEE
     Data.Print TRACK.ADDRESS()
)
ENDDO
```

sYmbol.SOURCEFILE() Name of source file

Syntax: **sYmbol.SOURCEFILE(***<address>* | *<symbol>[,<want_load_path>]***)**

Returns the name of the source file for the specified program address or symbol. The address has to be classified, e.g. **P**:0x200. In this example, **P:** stands for the memory class *Program Memory*.

This function can only be used with program addresses/symbols. It returns an empty string for data addresses / variables.

Parameter and Description:

<address>	**Parameter Type**: Address.
<symbol>	**Parameter Type**: Symbol.
<want_load_path>	**Parameter Type**: Boolean. Returns the file loading path, if parameter is true or omitted. Otherwise the compilation file path, generated at build time, is returned.

Return Value Type: String.

Example:

```
PRINT sYmbol.SOURCEFILE(main)
PRINT sYmbol.SOURCEFILE(main,FALSE())
```

sYmbol.SOURCELINE() HLL-line number of address

Syntax: **sYmbol.SOURCELINE(***<address>***)**

Returns the HLL-line number of the specified address. The address has to be classified, e.g. **P**:0x200. In this example, **P** stands for the memory class *Program Memory*.

Parameter Type: Address.

Return Value Type: Decimal value.

Example 1:

```
PRINT sYmbol.SOURCELINE(P:0x40005EB8)
```

Example 2: A user-defined command with the name **EDT** is created. When you execute it, TRACE32 opens the source file in your external editor (here: TextPad) and positions the curser in the line where the program counter (PC) is located in TRACE32.

```
; command extension to call external editor
; adapt path and parameter syntax to your own editor, e.g.:
; - uedit32.exe filename/line
; - textpad.exe filename(line)

ON CMD EDT GOSUB
(
   LOCAL &file &line &cmdline
   ENTRY &file
   IF "&file"==""
   (
      &file=sYmbol.SOURCEFILE(P:Register(pc))
      &line=sYmbol.SOURCELINE(P:Register(pc))
      &line=STRing.CUT("&line",-1.)
   )
   &cmdline="OS.Command start textpad.exe &file(&line)"
   &cmdline
   RETURN
)
STOP
ENDDO
```

sYmbol.SOURCEPATH() TRUE if path is search path

Syntax: **sYmbol.SOURCEPATH(**<*directory_path*>**)**

Returns TRUE if the given directory name is already defined as directory search path (by **sYmbol.SourcePATH** or loader option **/PATH**).

Parameter Type: String.

Return Value Type: Boolean.

Example:

```
&my_path=sYmbol.SOURCEFILE(main)
&my_path=OS.FILE.PATH(&my_path)
PRINT "&my_path"
sYmbol.SourcePATH.Set &my_path
PRINT  sYmbol.SOURCEPATH(&my_path)
sYmbol.SourcePATH.List
ENDDO
```

sYmbol.STATE() Value from sYmbol.state window

[build 53267 - DVD 08/2014]

Syntax: **sYmbol.STATE(**<*name*>**)**

Returns a value from the window **sYmbol.STATE** specified by its name.

Parameter Type: String.

Return Value Type: String.

Example:

```
&functions=sYmbol.STATE(functions)
PRINT %Decimal &functions " known functions in your target program(s)"
```

sYmbol.TRANSPOSE() Transpose program and module names

[build 122118 - DVD 09/2020]

Syntax: **sYmbol.TRANSPOSE(**<*name*>**)**

Allows to transpose program and module names following this rule:

1. A '_' is added as first characcter, if the name begins with 0--9 or '$'.
2. Any character expect 0--9, A--Z, a--z is replaced by a '_'.

This transposing can be switched off by adding the option /**NoTranspose** to any **Data.LOAD** comman.

Parameter Type: String.

Return Value Type: String.

Example:

```
PRINT sYmbol.TRANSPOSE("1_test-12@test-bay")
; prints "_1_test_12_test_bay" to AREA
```

sYmbol.TYPE() Type of symbol

Syntax: **sYmbol.TYPE(***<symbol>***)**

Returns the basic type of a symbol.

Parameter Type: Symbol.

Return Value Type: Hex value.

Return Value and Description:

0	Symbol does not exist.
1	Plain label without type information.
2	HLL function.
3	HLL variable.
(-)	Other values may be defined in the future.

Examples:

```
PRINT sYmbol.TYPE(qwertzuiop)
PRINT sYmbol.TYPE(_main)
PRINT sYmbol.TYPE(sieve)
PRINT sYmbol.TYPE(flags)
```

Syntax: **sYmbol.VARNAME(**<*address*>**)**

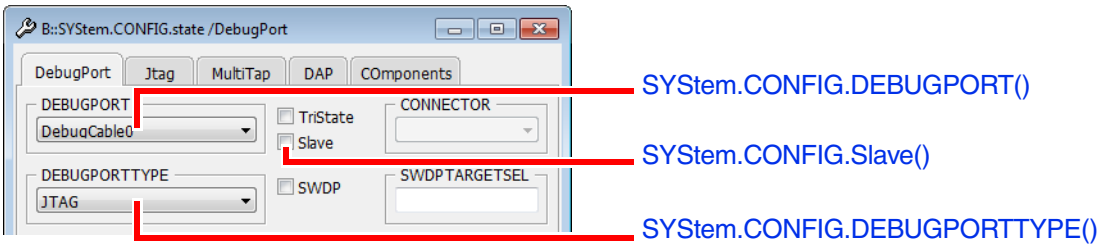
Returns the name of the variable or structure element at the specified address.

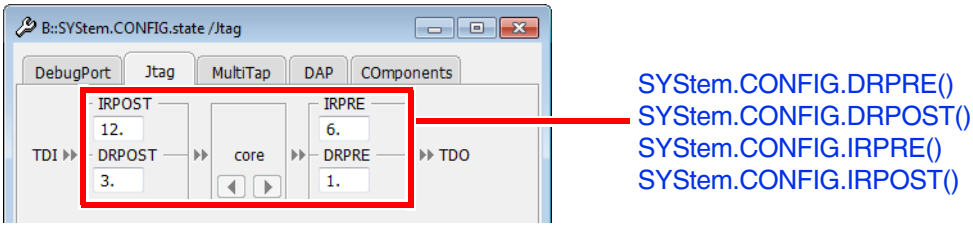
Parameter Type: Address.

Return Value Type: String.

SYStem Functions

This figure provides an overview of the return values of some of the SYStem functions. For descriptions of the illustrated SYStem functions and other SYStem functions, see below.





In This Section

See also

- SYStem
- SYStem.AMBA()
- SYStem.CADIconfig.RemoteServer()
- SYStem.CONFIG.DEBUGPORT()
- SYStem.CONFIG.DRPOST()
- SYStem.CONFIG.IRPOST()
- SYStem.CONFIG.JTAGTAP()
- SYStem.CONFIG.ListSIM()
- SYStem.CONFIG.TAPState()
- SYStem.DCI.Bridge()
- SYStem.DCI.TIMEOUT()
- SYStem.GTL.CONNECTED()
- SYStem.GTL.LIBname()
- SYStem.GTL.VENDORID()
- SYStem.HOOK()
- SYStem.IMASKHLL()
- SYStem.INSTANCECOUNT()
- SYStem.JtagClock()
- SYStem.MCDCommand.ResultString()
- SYStem.Mode()
- SYStem.Option.DUALPORT()
- SYStem.Option.HRCWOVerRide()
- SYStem.Option.MMUSPACES()
- SYStem.Option.SPILLLOCation()
- SYStem.RESetBehavior()
- SYStem.USECORE()
- SYStem.ACCESS.DENIED()
- SYStem.BigEndian()
- SYStem.CADIconfig.Traceconfig()
- SYStem.CONFIG.DEBUGPORTTYPE()
- SYStem.CONFIG.DRPRE()
- SYStem.CONFIG.IRPRE()
- SYStem.CONFIG.ListCORE()
- SYStem.CONFIG.Slave()
- SYStem.CPU()
- SYStem.DCI.BssbClock()
- SYStem.GTL.CALLCOUNTER()
- SYStem.GTL.CYCLECOUNTER()
- SYStem.GTL.PLUGINVERSION()
- SYStem.GTL.VERSION()
- SYStem.IMASKASM()
- SYStem.INSTANCE()
- SYStem.IRISconfig.RemoteServer()
- SYStem.LittleEndian()
- SYStem.MCDconfig.LIBrary()
- SYStem.NOTRAP()
- SYStem.Option.EnReset()
- SYStem.Option.MACHINESPACES()
- SYStem.Option.ResBreak()
- SYStem.Option.ZoneSPACES()
- SYStem.Up()
- SYStem.USEMASK()

SYStem.ACCESS.DENIED() TRUE if memory access is denied

Syntax: **SYStem.ACCESS.DENIED()**

Returns whether memory accesses are allowed during real-time emulation.

Return Value Type: Boolean.

Example:

```
SYStem.MemAccess Denied
PRINT SYStem.ACCESS.DENIED()   ;returns TRUE
```

SYStem.AMBA() TRUE if AMBA bus mode is selected

Syntax: **SYStem.AMBA()**

Returns TRUE if **SYStem.Option.AMBA** is active.

Refer to **"Arm Debugger"** (debugger_arm.pdf) for more information.

Return Value Type: Boolean.

SYStem.BigEndian() TRUE if target core runs in big endian mode

[build 05147 - DVD 12/2009]

Syntax: **SYStem.BigEndian()**

Returns TRUE if target core runs in big endian mode.

Return Value Type: Boolean.

Syntax: **SYStem.CADIconfig.RemoteServer(***<key>***)**

<key>: **1** | **2** | **3**

Returns the information about the connection to the CADI server. The connection is configured with the command **SYStem.CADIconfig.RemoteServer**.

Parameter Type: Decimal or hex or binary value.

Parameter and Description:

1	Returns the IP address or host name of the remote computer where the virtual target including the CADI server is running.
2	Returns the TCP/IP port of the CADI server.
3	Returns the IP address or host name and the port as follows: `<ip>:<port>`

Return Value Type: String.

Return Value and Description:

- IP address or host name or port depend on the parameter passed to the function.
- For local settings, `localhost` without any port is returned.

Examples:

```
SYStem.CADIconfig.RemoteServer
PRINT SYStem.CADIconfig.RemoteServer(3)     ;returns localhost

SYStem.CADIconfig.RemoteServer 192.168.178.2 7002.
PRINT SYStem.CADIconfig.RemoteServer(1)     ;returns 192.168.178.2
PRINT SYStem.CADIconfig.RemoteServer(3)     ;returns 192.168.178.2:7002

SYStem.CADIconfig.RemoteServer RmtPC 7000.
PRINT SYStem.CADIconfig.RemoteServer(2)     ;returns RmtPC
```

Syntax: **SYStem.CADIconfig.Traceconfig(1 | 2 | 3)**

Returns information about the connection to the CADI trace plug-in. The connection is configured with the command **SYStem.CADIconfig.Traceconfig**.

Parameter Type: Decimal or hex or binary value.

Parameter and Description:

1	Returns the IP address of the host machine where the virtual target is running.
2	Returns the TCP/IP port of the trace plug-in.
3	Returns the IP address and the port as follows: `<ip>:<port>`

Return Value Type: String.

SYStem.CONFIG.<tap_position>()

[build 45294 - DVD 08/2013] [Go to figure]

Syntax:	**SYStem.CONFIG.***<tap_position>***(***<core_index>***)**
<tap_position>:	**DRPOST** | **DRPRE** | **IRPOST** | **IRPRE**
Full function name only required for HELP.Index:	**SYStem.CONFIG.DRPOST(***<core_index>***)** **SYStem.CONFIG.DRPRE(***<core_index>***)** **SYStem.CONFIG.IRPOST(***<core_index>***)** **SYStem.CONFIG.IRPRE(***<core_index>***)**

Returns the effective JTAG PRE and POST settings of the core to be debugged. See also command **SYStem.CONFIG.DRPRE**.

Parameter and Description:

<core_index>	**Parameter Type**: Decimal or hex or binary value. • The index number of the core is **0** if you are debugging only one core. • If you are debugging multiple cores in an SMP debug session and if the cores use different TAP controller in the JTAG scan chain, then *<core_index>* refers to the core of interest.

Return Value Type: Decimal value.

SYStem.CONFIG.DEBUGPORT()

[build 53511 - DVD 08/2014] [Go to figure]

Syntax:	**SYStem.CONFIG.DEBUGPORT()**

Returns the selected debug port.

Details about the meaning of the return value can be found in the description of the **SYStem.CONFIG.DEBUGPORT** command.

Return Value Type: String.

SYStem.CONFIG.DEBUGPORTTYPE()

[build 53511 - DVD 08/2014] [Go to figure]

Syntax:	**SYStem.CONFIG.DEBUGPORTTYPE()** **DEBUGPORT.TYPE()** (deprecated)

Returns the selected debug port type, e.g. JTAG, CJTAG.

Return Value Type: String.

SYStem.CONFIG.JTAGTAP() Return the JTAG PRE and POST settings

[build 103442 - DVD 02/2019] [Examples]

Syntax:	**SYStem.CONFIG.JTAGTAP(***<item>***,***<config_index>***)**
<item>:	[*<component>*]**IRPRE**[**.***<subitem>*] [*<component>*]**DRPRE**[**.***<subitem>*] [*<component>*]**IRPOST**[**.***<subitem>*] [*<component>*]**DRPOST**[**.***<subitem>*]
<component>:	**DAP** | **DAP2** | **ETB** | … (depending on the CPU architecture)
<subitem>:	**ABSOLUTE** | **FIXED**

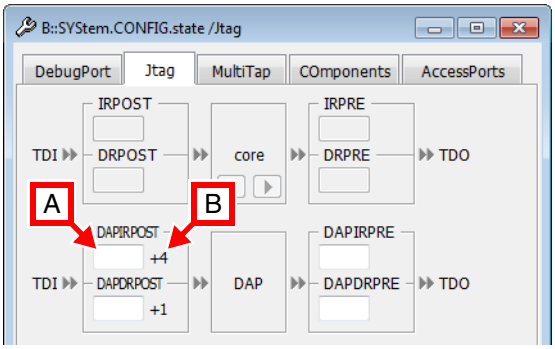
Returns the JTAG PRE and POST settings of the core TAP, DAP TAP, ETB TAP and others used for debugging. See also commands **SYStem.CONFIG.IRPRE**, **SYStem.CONFIG DAPIRPRE**, etc. in the **Processor Architecture Manuals**.

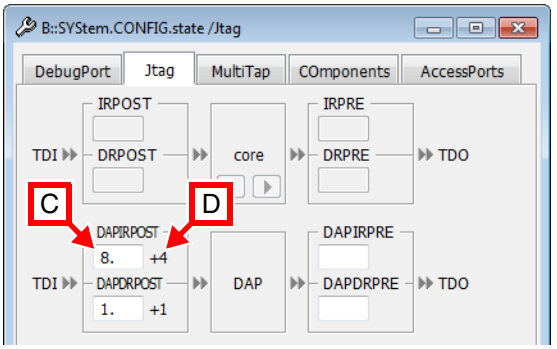
Parameter and Description:

<item>	**Parameter Type**: String. Name of the JTAG TAP coordinate which reflects a setting in the **SYStem.CONFIG.state /Jtag** window, e.g. the setting for **DAPIRPOST**. Per default, the value entered by the user is returned.
<subitem>	**Parameter Type**: String. Per default, the value entered by the user is returned. • **FIXED** returns the preconfigured value in TRACE32 relative to the SoC. • **ABSOLUTE** returns the preconfigured value + the value entered by the user.
<config_index>	**Parameter Type**: Decimal value. Index of the physical core TAP; in case of DAP or ETB the value must be 0. • **1**<= x <= **CONFIGNUMBER()**. • **0** is an alias for the first physical core or items that are shared among multiple cores.

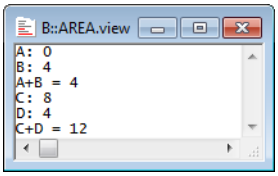
Return Value Type: Decimal value.

Example 1: For the ARM architecture

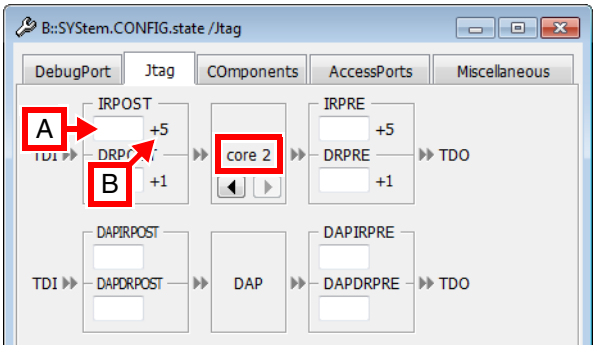


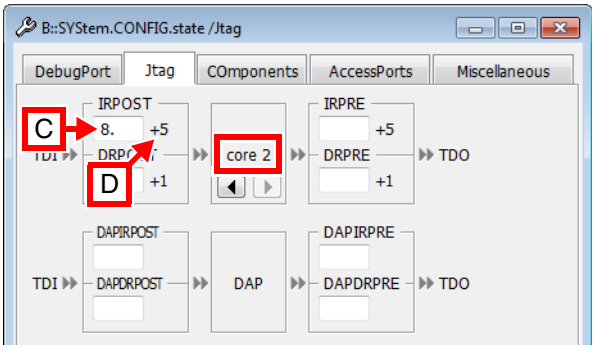



```
;[A] returns the value entered by the user for DAPIRPOST
PRINT "A: " SYStem.CONFIG.JTAGTAP(DAPIRPOST,0)                ;returns 0

;[B] returns the preconfigured value for DAPIRPOST relative
;to the SoC
PRINT "B: " SYStem.CONFIG.JTAGTAP(DAPIRPOST.FIXED,0)          ;returns 4

;[A]+[B] returns the value entered by the user + the
;preconfigured value. In this case the result is again 4
PRINT "A+B = " SYStem.CONFIG.JTAGTAP(DAPIRPOST.ABSOLUTE,0)    ;returns 4


;now the SoC is daisy-chained with another SoC which has
;the IR-length 8 and the DR-length 1
SYStem.CONFIG DAPIRPOST 8.
SYStem.CONFIG DAPDRPOST 1.

;[C] returns the value entered by the user for DAPIRPOST
PRINT "C: " SYStem.CONFIG.JTAGTAP(DAPIRPOST,0)                ;returns 8

;[D] returns the preconfigured value for DAPIRPOST relative
;to the SoC
PRINT "D: " SYStem.CONFIG.JTAGTAP(DAPIRPOST.FIXED,0)          ;returns 4

;[C]+[D] returns the value entered by the user + the
;preconfigured value. Now the result is 12
PRINT "C+D = " SYStem.CONFIG.JTAGTAP(DAPIRPOST.ABSOLUTE,0)    ;returns 12
```

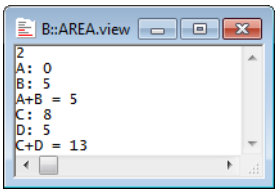


Example 2: For the MIPS architecture






```
PRINT CONFIGNUMBER() ;returns the upper bound of <config_index>, here 2

;[A] returns the value entered by the user for IRPOST
PRINT "A: " SYStem.CONFIG.JTAGTAP(IRPOST,2)                    ;returns 0

;[B] returns the preconfigured value for IRPOST relative
;to the SoC
PRINT "B: " SYStem.CONFIG.JTAGTAP(IRPOST.FIXED,2)              ;returns 5

;[A]+[B] returns the value entered by the user + the
;preconfigured value. in this case the result is again 5
PRINT "A+B = " SYStem.CONFIG.JTAGTAP(IRPOST.ABSOLUTE,2)        ;returns 5


;now the SoC is daisy-chained with another SoC which has the
;IR-length 8 and the DR-length 1
SYStem.CONFIG IRPOST 8.
SYStem.CONFIG DRPOST 1.

;[C] returns the value entered by the user for IRPOST
PRINT "C: " SYStem.CONFIG.JTAGTAP(IRPOST,2)                    ;returns 8

;[D] returns the preconfigured value for DAPIRPOST relative
;to the SoC
PRINT "D: " SYStem.CONFIG.JTAGTAP(IRPOST.FIXED,2)              ;returns 5

;[C]+[D] returns the value entered by the user + the
;preconfigured value. now the result is 13
PRINT "C+D = " SYStem.CONFIG.JTAGTAP(IRPOST.ABSOLUTE,2)        ;returns 13
```



SYStem.CONFIG.ListCORE()

[Examples]

Syntax: **SYStem.CONFIG.ListCORE(***<line_number>***,"***<column_string>***")**

Returns the core list from the virtual target platform only once with first call of the function **SYStem.CONFIG.ListCORE()** or the command **SYStem.CONFIG.ListCORE**. This will require a **SYStem.Mode Down** state. All later function calls return the core list from a table stored locally.

Parameter and Description:

<line_number>	**Parameter Type**: Decimal or hex value. A numerical input indicating the line number for the search; *<line_number>* equals the core # number. **Range**: • **0x0** or **0.** returns the number of available lines, i.e. cores. • **-1** re-reads the core information for the virtual target platform.
<column_string>	**Parameter Type**: String. A case sensitive search string that must match the searched column name. An empty string "" is allowed.

Return Value Type: String.

Return Value and Description:

String	If the search string *<column_string>* is found, the complete element is returned as a string.
Empty string	If the search string *<column_string>* does not match a column name and/or the *<line_number>* was out of its currently valid range, an empty string is returned.

Examples for SYStem.CONFIG.ListCORE():

```
;returns the number of cores
PRINT SYStem.CONFIG.ListCORE(0,"")

;returns the value at the intersection of the 2nd row and the column
;labeled "simulation".
PRINT SYStem.CONFIG.ListCORE(2,"simulation")

;returns an empty string because the column labeled "SimulatioN" does not
;exist. Remember that the "<column_string>" is case-sensitive.
PRINT SYStem.CONFIG.ListCORE(2,"SimulatioN")

PRINT SYStem.CONFIG.ListCORE(200.,"device")

PRINT SYStem.CONFIG.ListCORE(2,"core")
```

```
;re-reads the core information for the virtual target platform
PRINT SYStem.CONFIG.ListCORE(-1,"")

;returns the updated value at the intersection of the 2nd row and the
column labeled "core"
PRINT SYStem.CONFIG.ListCORE(2,"core")
```

SYStem.CONFIG.ListSIM()

Syntax: **SYStem.CONFIG.ListSIM(***<line_number>***,"***<column_string>***")**

The function returns a string, and both parameters are mandatory. The content of the simulation list is read from the virtual target platform only once with first call of the function **SYStem.CONFIG.ListSIM()** or the command **SYStem.CONFIG.ListSIMulation**. This will require a **SYStem.Mode Down** state. All later function calls return strings from a table stored locally.

Parameter and Description:

<line_number>	**Parameter Type**: Decimal or hex value. A numerical input, indicating the line number for the search; *<line_number>* equals the core # number. **Range**: • **0x0** or **0.** returns the number of available lines, i.e. cores. • **-1** re-reads the core information for the virtual target platform.
<column_string>	**Parameter Type**: String. A case-sensitive search string that must match the searched column name. An empty string "" is allowed.

Return Value Type: String.

Return Value and Description:

String	If the search string *<column_string>* was found, the complete element is returned as a string.
Empty string	If the search string *<column_string>* does not match a column name and/or the *<line_number>* was out of its currently valid range, an empty string is returned.

SYStem.CONFIG.Slave()

[build 45263 - DVD 08/2013] [Go to figure]

Syntax: **SYStem.CONFIG.Slave()**

Returns TRUE if setting **SYStem.CONFIG Slave** is set to **ON**.

Return Value Type: Boolean.

SYStem.CONFIG.TAPState()

[build 71651 - DVD 02/2016]

Syntax: **SYStem.CONFIG.TAPState()**

Returns the default JTAG TAP state configured with the command **SYStem.CONFIG TAPState**.

Return Value Type: Decimal value.

Return Value and Description:

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

SYStem.CPU() Name of processor

[build 51040 - DVD 02/2014]

Syntax: **SYStem.CPU()**
CPU() (deprecated)

Returns the name of the processor, which was selected with the command **SYStem.CPU**. This function is an alias for **STATE.PROCESSOR()**.

Return Value Type: String.

SYStem.GTL.CALLCOUNTER() Amount of calls to GTL library

[build 59192 - DVD 02/2015]

Syntax: **SYStem.GTL.CALLCOUNTER()**

Retrieves the number of calls to the GTL library to measure performance.

Return Value Type: Decimal value.

SYStem.GTL.CONNECTED() Connection status

[build 101750 - DVD 09/2018]

Syntax: **SYStem.GTL.CONNECTED()**

Returns TRUE if all configured transactors are connected to the simulation or emulation.

Return Value Type: Boolean.

SYStem.GTL.CYCLECOUNTER() load GTL interface for bit banging protocol

[build 147511 - DVD 09/2022]

Syntax: **SYStem.GTL.CYCLECOUNTER()**

Retrieves the number of clock cycles of the executed protocol since the last call. In case of JTAG, **SYStem.GTL.CYCLECOUNTER()** counts the JTAG clock cycles.

Return Value Type: Decimal value.

SYStem.GTL.LIBname() Name of GTL library

Syntax: **SYStem.GTL.LIBname()**

Returns the name of the GTL library that was passed by the command **SYStem.GTL.LIBname**.

Return Value Type: String.

SYStem.GTL.ModelINFO() Info string from GTL API

Syntax: **SYStem.GTL.ModelINFO(**<*n*>**)**

Returns the info string of the n[th] model enumerated from GTL API. The info string can be used to pass information about the model.

If *n* equals -1, the function returns the info string of the current model. The index for iteration start by 0. The returned values are valid for all values n where **SYStem.GTL.ModelNAME(n)** returns a valid name.

Parameter Type: Decimal value.

Return Value Type: String.

Example:

```
LOCAL &n
&n=0.
while SYStem.GTL.ModelNAME(&n)!=""
(
  PRINT "Model: " SYStem.GTL.ModelNAME(&n) " Info: "
SYStem.GTL.ModelINFO(&n)
  &n=&n+1.
)
ENDDO
```

SYStem.GTL.ModelNAME() Model Name

Syntax: **SYStem.GTL.ModelNAME(**<*index*>**)**

Returns the name of a certain model.

Parameter Type: Decimal value.

Return Value Type: String.

SYStem.GTL.PLUGINVERSION() Version number

Syntax: **SYStem.GTL.PLUGINVERSION()**

Returns the version number of the GTL API retrieved from the GTL library/plug-in.

Return Value Type: Decimal value.

SYStem.GTL.TransactorNAME() Transactor name

Syntax: **SYStem.GTL.TransactorNAME(**<*index*>**)**

Returns the name of a transactor.

Parameter Type: Decimal value.

Return Value Type: String.

SYStem.GTL.TransactorTYPE() Transactor type

Syntax: **SYStem.GTL.TransactorTYPE(**<*index*>**)**

Returns the type of a transactor.

Parameter Type: Decimal value.

Return Value Type: String.

Return Value and Description: Types can be JTAGPROBE, GPIO, TRACE, BUSMASTER,...

SYStem.GTL.VENDORID() Vendor ID

Syntax: **SYStem.GTL.VENDORID()**

Returns the vendor ID retrieved from the GTL library.

Return Value Type: String.

Syntax: **SYStem.GTL.VERSION()**

Returns the version number of the GTL API of TRACE32.

Return Value Type: Decimal value.

SYStem.HOOK()

Syntax: **SYStem.HOOK()**

Returns address of the hook function defined with **SYStem.Option.HOOK**.

Return Value Type: Hex value.

SYStem.IMASKASM()

Syntax: **SYStem.IMASKASM()**

Returns the TRUE if interrupts are disabled during assembler stepping due to the setting **SYStem.Option.IMASKASM ON**.

Return Value Type: Boolean.

SYStem.IMASKHLL()

Syntax: **SYStem.IMASKHLL()**

Returns the TRUE if interrupts are disabled during HLL stepping due to the setting **SYStem.Option.IMASKHLL ON**.

Return Value Type: Boolean.

Syntax: **SYStem.INSTANCE()**

If a TRACE32 PowerView instance belongs to an AMP debug session, then this function returns the index of the TRACE32 PowerView instance.

Return Value Type: Hex value.

Return Value and Description:

0x1…	If the PowerView instances have been started via the T32Start utility, then the return values correspond to the values that T32Start has assigned to the PowerView instances. Without the use of T32Start, the return value corresponds to the *<index>* value *you* have specified for `INSTANCE=<index>` in the config file.
0x0	The index of a PowerView instance is `0x0` (a) if the configuration file contains the setting `CORE=0` or (b) if TRACE32 was started as TRACE32 Instruction Set Simulator (`PBI=SIM` in the config file).

For information about `CORE=` and `INSTANCE=`, refer to:

- **"Section PBI"** (installation.pdf)

Example:

```
;the instance indexes of all PowerView GUIs participating in an
;AMP debug session are printed to their respective message lines
InterCom.execute ALL ECHO SYStem.INSTANCE()
```

SYStem.INSTANCECOUNT() Count of GUIs connected to a PowerDebug

[build 111259 - DVD 09/2019]

Syntax: **SYStem.INSTANCECOUNT()**

This function returns the number of TRACE32 PowerView GUIs connected to the same PowerDebug box.

Return Value Type: Decimal value.

SYStem.IRISconfig.RemoteServer()

[build 109127 - DVD 09/2019]

Syntax: **SYStem.IRISconfig.RemoteServer(***<key>***)**

<key>: **1** | **2** | **3**

Returns the details about the connection to the IRIS server. The connection is configured with the command **SYStem.IRISconfig.RemoteServer**.

Parameter Type: Decimal or hex or binary value.

Parameter and Description:

1	Returns the IP address or host name of the remote computer where the virtual target including the IRIS server is running.
2	Returns the TCP/IP port of the IRIS server.
3	Returns the IP address and the port as follows: `<ip>:<port>.`

Return Value Type: String.

SYStem.JtagClock()

Syntax: **SYStem.JtagClock()**

Returns the JTAG clock set with the command **SYStem.JtagClock**.

Return Value Type: Decimal value.

SYStem.LittleEndian()

[build 05147 - DVD 12/2009]

Syntax: **SYStem.LittleEndian()**

Returns TRUE if target core runs in little endian mode.

Return Value Type: Boolean.

SYStem.MCDCommand.ResultString()

[build 136589 - DVD 09/2021]

Syntax: **SYStem.MCDCommand.ResultString()**

Returns the answer-string of the MCD server given by the last call of **SYStem.MCDCommand**. The maximum length of this string is 100 characters. It returns an empty string if no answer is available.

Return Value Type: String.

SYStem.MCDconfig.LIBrary()

[build 76420 - DVD 09/2016]

Syntax: **SYStem.MCDconfig.LIBrary(***<key>***)**

<key>: **1** | **2** | **3**

Returns the path and/or file name of the used MCD library.

Parameter and Description:

1	**Parameter Type**: Decimal or hex or binary value. Returns the file name of the library, e.g. mcddrv.dll
2	**Parameter Type**: Decimal or hex or binary value. Returns the path of the library, e.g. c:\virtual_target\mcd\
3	**Parameter Type**: Decimal or hex or binary value. Returns the path and the file name of the library, e.g. c:\virtual_target\mcd\mcddrv.dll

Return Value Type: String.

SYStem.Mode()

Syntax: **SYStem.Mode()**

Returns the actual debugger mode.

Return Value Type: Hex value.

Return Value and Description:

0	Down
1	StandBy
2	NoDebug
4	Prepare
11	Up
12	Up (StandBy)
13	Prepare (StandBy)

SYStem.NOTRAP() 1 if the option NOTRAP is active

Syntax: **SYStem.NOTRAP()**

Returns 1 if the command **SYStem.Option.NOTRAP** is active.

Return Value Type: Hex value.

SYStem.Option.DUALPORT() State of like-named command

Only available for architectures having the like-named command [build 108886 - DVD 09/2019]

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

Returns TRUE if the command **SYStem.Option.DUALPORT** is set to **ON**, else the function returns FALSE.

Return Value Type: Boolean.

SYStem.Option.MACHINESPACES() State of like-named command

ARM, PowerPC, Intel® x86 [build 104967 - DVD 02/2019]

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

Returns TRUE if the command **SYStem.Option.MACHINESPACES** is set to **ON**, else the function returns FALSE.

Return Value Type: Boolean.

Example:

```
IF SYStem.Option.MACHINESPACES()==FALSE()
   SYStem.Option.MACHINESPACES ON
```

SYStem.Option.MMUSPACES() State of like-named command

ARM, PowerPC, Intel® x86 [build 104967 - DVD 02/2019]

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

Returns TRUE if the command **SYStem.Option.MMUSPACES** is set to **ON**, else the function returns FALSE.

Return Value Type: Boolean.

Example

```
IF SYStem.Option.MMUSPACES()==FALSE()
   SYStem.Option.MMUSPACES ON
```

SYStem.Option.EnReset() State of like-named command

ARM [build 135725 - DVD 09/2021]

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

Returns the current state of **SYStem.Option.EnReset** as TRUE or FALSE.

Return Value Type: Boolean.

SYStem.Option.ResBreak() State of like-named command

ARM [build 134096 - DVD 09/2021]

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

Returns the current state of **SYStem.Option.ResBreak** as TRUE or FALSE.

Return Value Type: Boolean.

SYStem.Option.SPILLLOCation() State of like-named command

XTENSA [build 141496 - DVD 02/2022]

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

Returns the current state of **SYStem.Option.SPILLLOCation** as TRUE or FALSE.

Return Value Type: Hex value.

SYStem.Option.ZoneSPACES() State of like-named command

ARM, PowerPC, Intel® x86 [build 104967 - DVD 02/2019]

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

Returns TRUE if the command **SYStem.Option.ZoneSPACES** is set to **ON**, else the function returns FALSE.

Return Value Type: Boolean.

Example

```
IF SYStem.Option.ZoneSPACES()==FALSE()
   SYStem.Option.ZoneSPACES ON
```

Syntax: **SYStem.RESetBehavior()**

Returns the current setting of **SYStem.Option.RESetBehavior**.

Return Value Type: String.

HALT **RestoreGo** **RunRestore**	For a description of the return values, see **SYStem.Option.RESetBehavior** in **"TriCore Debugger and Trace"** (debugger_tricore.pdf).

Example:

```
SYStem.Option.RESetBehavior Halt
PRINT SYstem.RESetBehvior()      ;returns the string HALT
```

SYStem.Up() TRUE if debugger has access to debug resources

Syntax: **SYStem.Up()**

Returns whether the actual mode of the probe is up (see also **SYStem.Mode()** function).

Return Value Type: Boolean.

Example 1:

```
IF !SYStem.Up()      ;alternatively: IF SYStem.Up()==FALSE()
(
  ;your code
)
ELSE
(
  ;your code
)
```

Example 2: This script shows how to access a peripheral register on the AHB bus prior to **SYStem.Mode Up** or **Attach**, e.g. to disable a watchdog.

```
SYStem.Mode Prepare

IF SYStem.Up()==TRUE()       ; if Prepare was successful
(
  ; configure peripheral register before SYStem.Mode Up or Attach
  Data.Set EAHB:<peripheral_address> %Long 0x1
)
```

SYStem.USECORE()

Syntax: **SYStem.USECORE()**

Returns the value of CORE= from the PBI= section in the config file.

If CORE= or INSTANCE= or both are missing in the config file, then **SYStem.USECORE()** returns 1 by default. That is, in this case the return value is equivalent to the explicit setting CORE=1 in the config file.

For information about CORE= and INSTANCE=, refer to:

- **"Section PBI"** (installation.pdf)

Return Value Type: Hex value.

Example: In this script, a second instance is started, and then **SYStem.USECORE()** is used to return the value of CORE=.

```
;returns 1
PRINT SYStem.USECORE()

;starts a 2nd TRACE32 PowerView instance with the user-defined name
;'secondInst', clones and extends the current config file for the 2nd
;instance
TargetSystem.NewInstance secondInst /ChipIndex 10. /ONCE

;the CONVert.HEXTOIN() function is used to convert the hex return value
;of SYStem.USECORE() to the decimal value 10
InterCom.execute secondInst PRINT CONVert.HEXTOINT(SYStem.USECORE())
```

SYStem.USEMASK()

Syntax: **SYStem.USEMASK()**

Returns the USEMASK of this TRACE32 PowerView GUI.

Return Value Type: Hex value.

The USEMASK selects the Lauterbach device, when several devices are connected to each other via PODBUS IN/OUT. The bitmask defines which devices (PowerDebug, PowerTrace, PowerProbe, ...) are controlled by the host application program. A '1' means control the device, a '0' means skip the device. The left most bit controls the first device on the bus i.e. the device with the host connection (USB, ETH). Normally a use mask will contain a single set bit e.g. “1000”, indicating that only one of the devices in the PODBUS is used by the respective program. When using PowerView to control two devices the use mask will contain two ’1’ bits. An example for this is a PowerDebug device and a PowerProbe device which are used to record signals corresponding to a debug session of PowerDebug. Using this mechanism it is possible to combine a debugger and a PowerProbe device, independently of their position in the PODBUS chain.

The USEMASK is set in the TRACE32 configuration file (config.t32) with option `USE=<bits>` in the section `PBI=`

However the function **SYStem.USEMASK()** returns the value set in `USE=` *in an inverse bit-order*. The same value is also shown in the **VERSION.HARDWARE** window.

The function returns 0xFFFFFFFFFFFFFFFF for a single debugger (where `USE=` was not set).

```
device    USE=    PRINT %BINary SYStem.USEMASK()   VERSION.HARDWARE
 1st      100             00000001                      0x0001
 2nd      010             00000010                      0x0002
 3rd      001             00000100                      0x0003
```

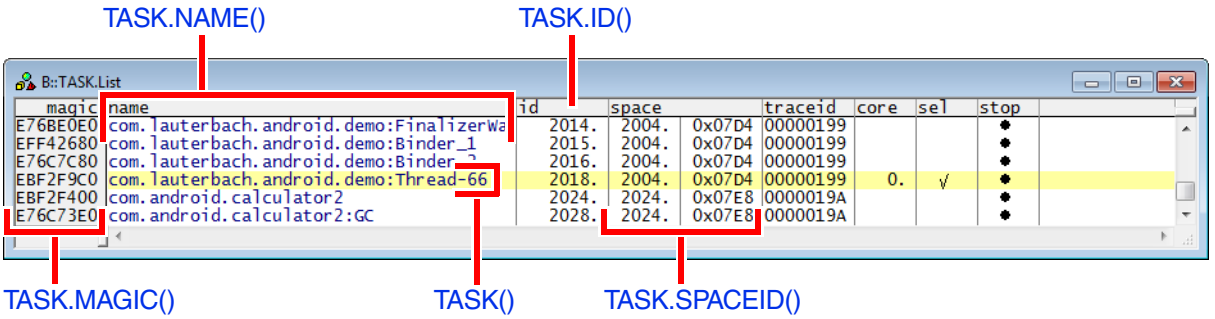
TASK Functions

The TASK functions can be used after OS-aware debugging has been configured, as described in the **Configuration** chapters of the **"OS Awareness Manuals"** (rtos_*<os>*.pdf).

In the following, we list the TASK functions that work with all OS-aware configurations.

For the TASK functions that work only with specific OS-aware configurations, refer to the **"OS Awareness Manuals"** (rtos_*<os>*.pdf).

This figure provides an overview of the return values of some of the TASK functions. For descriptions of the illustrated TASK functions and other TASK functions, see below.



In This Section

See also

- TASK
- TASK.ACCESS()
- TASK.BACK()
- TASK.CONFIGFILE()
- TASK.CURRENT.MACHINEID()
- TASK.CURRENT.TASK()
- TASK.FIRST()
- TASK.ID()
- TASK.MACHINE.ID()
- TASK.MACHINE.VTTB()
- TASK.MAGIC()
- TASK.MAGICRANGE()
- TASK.NAME()
- TASK.ORTIFILE()
- TASK.SPACEID()
- TASK()
- TASK.ACCESS.ZONE()
- TASK.CONFIG()
- TASK.COUNT()
- TASK.CURRENT.SPACEID()
- TASK.CURRENT.TASKNAME()
- TASK.FORE()
- TASK.MACHINE.ACCESS()
- TASK.MACHINE.NAME()
- TASK.MACHINEID()
- TASK.MAGICADDRESS()
- TASK.MAGICSIZE()
- TASK.NEXT()
- TASK.SPACE.COUNT()

TASK() Name of current task

[Go to figure]

Syntax: **TASK()**

Returns the name of the current task.
Short for TASK.CURRENT.TASKNAME()

Return Value Type: String.

TASK.ACCESS() Access class

[build 69123]

Syntax: **TASK.ACCESS()**

Returns the access class set by the command **TASK.ACCESS**.

Return Value Type: String.

TASK.ACCESS.ZONE() Access class zone

[build 68412]

Syntax: **TASK.ACCESS.ZONE()**

Returns the access class zone set by the command **TASK.ACCESS**.

Return Value Type: String.

TASK.BACK() Background task number

Syntax: **TASK.BACK()**

Returns the background task number.

Return Value Type: Hex value.

Syntax: **TASK.CONFIG(magic | magicsize)**

Parameter and Description:

magic	**Parameter Type**: String (***without*** quotation marks). Returns the magic address, which is the location that contains the currently running task (i.e. its task magic number). The task is identified by a unique value called *task magic number*. The magic numbers of the tasks are displayed in the **magic** column of the **TASK.List.tasks** window.
magicsize	**Parameter Type**: String (***without*** quotation marks). Returns the size of the task magic number (1, 2 or 4).

Return Value Type: Hex value.

Example 1:

```
PRINT TASK.CONFIG(magic)
```

Example 2:

```
PRINT TASK.CONFIG(magicsize)
```

TASK.CONFIGFILE() Path of loaded OS Awareness

[build 78972]

Syntax: **TASK.CONFIGFILE()**

Returns the file (with path) used to load an OS Awareness with **TASK.CONFIG**.

Return Value Type: String.

TASK.COUNT() Number of tasks

[build 123903]

Syntax: **TASK.COUNT()**

Returns the number of tasks in the task list.

Return Value Type: Hex value.

TASK.CURRENT.MACHINEID() ID of current machine

[build 137549]

Syntax: **TASK.CURRENT.MACHINEID()**

Returns the ID of the current machine. Only valid if SYStem.Option MACHINESPACES is set.

Return Value Type: Hex value.

TASK.CURRENT.SPACEID() ID of current MMU space

[build 137549]

Syntax: **TASK.CURRENT.SPACEID()**

Returns the ID of the current MMU space. Only valid if SYStem.Option MMUSPACES is set.

Return Value Type: Hex value.

TASK.CURRENT.TASK() Magic value of current task

[build 137549]

Syntax: **TASK.CURRENT.TASK()**

Returns the “magic value” of the current task.

Return Value Type: Hex value.

TASK.CURRENT.TASKNAME() Name of current task

[build 137549]

Syntax: **TASK.CURRENT.TASKNAME()**

Returns the name of the current task.

Return Value Type: String.

TASK.FIRST() First task in list

[build 123903]

Syntax: **TASK.FIRST()**

Returns the task magic number of the first task in the task list.

Return Value Type: Hex value.

TASK.FORE() Foreground task number

Syntax: **TASK.FORE()**

Returns the foreground task number.

Return Value Type: Hex value.

TASK.ID() ID of task

[Go to figure]

Syntax: **TASK.ID("***<task_name>***")**

Returns the ID of the specified task name.

Parameter Type: String.

Return Value Type: Hex value.

TASK.MACHINE.ACCESS() Default access class

[build 92265 - DVD 01/2018]

Syntax: **TASK.MACHINE.ACCESS(***<machine_id>***)**

Returns the default access class of the machine specified by *<machine_id>*.

This function is useful on systems with a hypervisor after **SYStem.Option.MACHINESPACES** has been set to **ON**. For some hypervisors it is difficult to determine whether a given guest machine runs in the guest mode (hardware virtualized machines) or in the host/hypervisor mode (pseudo virtualized machines). The **TASK.MACHINE.ACCESS()** function can be used to find out with which access class a logical addresses of a given machine must be prefixed.

Examples:

- On Intel systems the default access class may be based on **H:** (machines running in VMX host mode) or **G:** (machines running in VMX guest mode).
- On ARM systems, the default access class may be based on **H:** (machines running in hypervisor mode) or **N:** (machines running in non-secure/guest mode).

Parameter Type: Decimal value.

Return Value Type: String.

TASK.MACHINE.ID() ID of machine

[build 123903]

Syntax: **TASK.MACHINE.ID("***<machine_name>***")**
TASK.MACHINEID("*<machine_name>***") (deprecated)**

Returns the ID of the given machine name if the machine exists.
Returns "1F" if the machine does not exist.

The machine name is set by a Hypervisor Awareness or by an extension name (see also the command **EXTension.LOAD**).

Parameter Type: String.

Return Value Type: Hex value.

TASK.MACHINE.NAME() Name of machine

[build 92273 - DVD 02/2018]

Syntax: **TASK.MACHINE.NAME(***<machine_id>* | *<machine_magic>***)**

Returns the name of the given machine ID if the machine exists.
Returns the machine ID if the machine exists but does not have a name.
Returns "(unknown)" if the machine does not exist.

The machine name is set by a Hypervisor Awareness or by an extension name (see also the command **EXTension.LOAD**).

Parameter and Description:

<machine_id>	**Parameter Type**: Decimal or hex or binary value. For information about the parameter, see machine ID.
<machine_magic>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: String.

Example 1: This script lists all functions of the machine with ID 1 in the **sYmbol.Browse.Function** window, provided the symbol program name is identical to the machine name.

```
LOCAL &machinename
&machinename=TASK.MACHINE.NAME(1)
sYmbol.Browse.Function \\&machinename\*\*
```

Example 2: Based on the specified address, the **ADDRESS.MACHINEID()** function extracts the machine ID. The machine ID is then passed to the **TASK.MACHINE.NAME()** function, which returns the machine name belonging to the machine ID.

```
PRINT TASK.MACHINE.NAME(ADDRESS.MACHINEID(P:0x1:::0x3000))
```

TASK.MACHINE.VTTB() VTTB of machine

[

Syntax: **TASK.MACHINE.VTTB(***<machine_id>* | *<machine_magic>***)**

Returns the virtualization translation table base address of the given machine if the machine exists. Returns -1 if the machine does not exist.

Parameter and Description:

<machine_id>	**Parameter Type**: Decimal or hex or binary value. For information about the parameter, see machine ID.
<machine_magic>	**Parameter Type**: Decimal or hex or binary value.

Return Value Type: Hex value.

TASK.MAGIC() Task magic number

[Go to figure]

Syntax: **TASK.MAGIC("***<task_name>***")**

Returns the task magic number of the specified task name.

Parameter Type: String.

Return Value Type: Hex value.

TASK.MAGICADDRESS() "magic address"

[build 123903]

Syntax: **TASK.MAGICADDRESS(**[*<core>*[**,***<machine_id>*]]**)**

Returns the address of the memory location holding the running task of the specified core and machine (aka "magic_address").

Parameter and Description:

<core>	**Parameter Type**: Hex value.
<machine_id>	**Parameter Type**: Hex value.
NONE	If no parameter is given the function returns the address holding the running task of the core and machine currently in use.

Return Value Type: Address.

TASK.MAGICRANGE() Range of "magic address"

[build 123903]

Syntax: **TASK.MAGICRANGE(**[*<core>*[**,***<machine_id>*]]**)**

Returns the range of the memory location holding the running task of the specified core and machine (aka "magic_address").

Parameter and Description:

<core>	**Parameter Type**: Hex value.
<machine_id>	**Parameter Type**: Hex value.
NONE	If no parameter is given the function returns the range of the memory location holding the running task of the core and machine currently in use.

Return Value Type: Address.

TASK.MAGICSIZE() Size of "magic address"

[build 123903]

Syntax: **TASK.MAGICSIZE(**[*<core>*[**,***<machine_id>*]]**)**

Returns the size of the memory location holding the running task of the specified core and machine (aka "magic_address").

Parameter and Description:

<core>	**Parameter Type**: Hex value.
<machine_id>	**Parameter Type**: Hex value.
NONE	If no parameter is given the function returns the size of the memory location holding the running task of the core and machine currently in use.

Return Value Type: Address.

TASK.NAME() Name of task

[Go to figure]

Syntax: **TASK.NAME(***<task_magic>***)**

Returns a task name based on the specified task magic number.

Parameter Type: Hex value.

Return Value Type: String.

Example:

```
PRINT TASK.NAME(0xEBF2F9C0)
```

TASK.NEXT() Next task in list

[build 123903]

Syntax: **TASK.NEXT(***<task_magic>***)**

Returns the task magic number of the task following the specified task in the task list.

Parameter Type: Hex value.

Return Value Type: Hex value.

Syntax: **TASK.ORTIFILE(**[*<machine_id>*]**)**

Returns the file (with path) used to load an ORTI file with **TASK.ORTI**/**EXT.ORTI.LOAD**. The optional parameter *<machine_id>* can be used in hypervisor and iAMP setups to specify the machine ID the ORTI file was loaded to.

Parameter and Description:

<machine_id>	Optional: Machine ID the ORTI file was loaded to. Useful for hypervisor/iAMP setups. **Parameter Type**: Decimal or hex or binary value.

Return Value Type: String.

Example:

```
; Single OS case
CD C:\Projects\os1
TASK.ORTI deploy/os.orti
PRINT TASK.ORTIFILE()
; Result: "C:\Projects\os1\deploy\os.orti"

; Hypervisor/iAMP scenario
CD C:\Projects
EXT.ORTI.LOAD os1/deploy/os.orti /MACHINE 1.
EXT.ORTI.LOAD os2/deploy/os.orti /MACHINE 2.
PRINT TASK.ORTIFILE(2.)
; Result: "C:\Projects\os2\deploy\os.orti"
PRINT TASK.ORTIFILE(1.)
; Result: "C:\Projects\os1\deploy\os.orti"
PRINT TASK.ORTIFILE(3.)
; Result: ""
```

TASK.SPACE.COUNT() Number of spaces

[build 123903]

Syntax: **TASK.SPACE.COUNT()**

Returns the number of address spaces in the list of spaces.

Return Value Type: Hex value.

[Go to figure]

Syntax: **TASK.SPACEID("***<task_name>***")**

Returns the space ID of the specified task name.

Parameter Type: String.

Return Value Type: Hex value.

TERM Functions (Terminal Window)

In This Section

See also

■ TERM ❑ TERM.LINE() ❑ TERM.RETURNCODE() ❑ TERM.TRIGGERED()

TERM.LINE() Get line from terminal window

Syntax: **TERM.LINE(***<channel>***,***<line_number>***)**
TERM.LINE(*<address>***,***<line_number>***)** (deprecated)

<channel>: **#***<number>*

Returns the content of the specified line from the active terminal window.

Parameter and Description:

<channel>	**Parameter Type**: String. Handle to refer to a terminal. A new handle can be created with **TERM.METHOD**.
<address>	**Parameter Type**: Address. *<address>* is the communication port of the virtual terminal.
<line_number>	**Parameter Type**: Decimal value.

Return Value Type: String.

TERM.READBUSY() TRUE as long as TERM.READ is in progress

[build 143560 - DVD 02/2022]

Syntax: **TERM.READBUSY()**

Returns TRUE as long as a **TERM.READ** command is in progress.

Return Value Type: Boolean.

Example:

```
TERM.READ #1 <file_name>      ; start upload
SCREEN.ALWAYS                 ; to make upload as fast as possible

;PRACTICE will loop through this while loop until the read is done
WHILE TERM.READBUSY(#1)
(
   //Some PRACTICE commands
)

SCREEN.ON
```

TERM.RETURNCODE() Get returncode from terminal routine

[build 58677 - DVD 02/2015]

Syntax: **TERM.RETURNCODE(***<channel>***)**
TERM.RETURNCODE(*<address>***)** (deprecated)

<channel>: **#***<number>*

Returns the return value of the semihosting function from the active terminal window.

Parameter and Description:

<channel>	**Parameter Type**: String. Handle to refer to a terminal. A new handle can be created with **TERM.METHOD**.
<address>	**Parameter Type**: Address. *<address>* is the communication port of the virtual terminal.

Return Value Type: Decimal value.

TERM.TRIGGERED() Get trigger state of terminal window

[build 95013 - DVD 09/2018]

Syntax: **TERM.TRIGGERED(***<channel>***)**
TERM.TRIGGERED(*<address_out>***)** (deprecated)

<channel>: **#***<number>*

Returns TRUE once the message string to be searched for in the active terminal window has been printed to the active terminal window. Use the **TERM.TRIGGER** command to specify the message string you want.

Parameter and Description:

<channel>	**Parameter Type**: String. Handle to refer to a terminal. A new handle can be created with **TERM.METHOD**.
<address_out>	**Parameter Type**: Address. The address parameter is only required for memory-based data exchange (SingleE, BufferE, SingleS, BufferS). Pass an arbitrary address for all non-memory-based methods e.g. Serial Console/COM.

Return Value Type: Boolean.

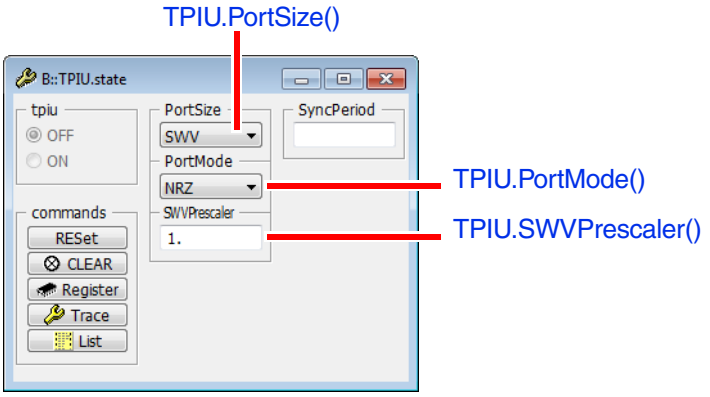
Example 1:

```
TERM.TRIGGER #3 "Hit any key"
SCREEN.WAIT TERM.TRIGGERED(#3)
```

Example 2: Refer to the **TERM.TRIGGER** command.

TPIU Functions

This figure provides an overview of the return values of the functions. For descriptions of the illustrated functions, see below.



In This Section

See also

■ TPIU ❑ TPIU.PortMode() ❑ TPIU.PortSize() ❑ TPIU.SWVPrescaler()

TPIU.PortMode() Port mode setting

[build 65602 - DVD 09/2015] [Go to figure]

Syntax: **TPIU.PortMode()**

Returns current **PortMode** setting of the TPIU made with the command **TPIU.PortMode**. For example, **Bypass**, **Wrapper**, **Continuous**, **NRZ**.

Return Value Type: String.

TPIU.PortSize() Port size setting

[build 65602 - DVD 09/2015] [Go to figure]

Syntax: **TPIU.PortSize()**

Returns the current **PortSize** setting of the TPIU made with the command **TPIU.PortSize**.

Return Value Type: String.

Return Value and Description:

1, **2**, **3**, **4**, ...	Port size setting for parallel trace.
1Lane, **2Lane**, **3Lane**, **4Lane**	Port size setting for serial trace.
SWV	Port size setting for SerialWireViewer.

TPIU.SWVPrescaler() SWVPrescaler value

[build 65602 - DVD 09/2015] [Go to figure]

Syntax: **TPIU.SWVPrescaler()**

Returns the current **SWVPrescaler** value set with the command **TPIU.SWVPrescaler**.

Return Value Type: Decimal value.

TPUBASE Function

TPUBASE.ADDRESS() Address of TPU

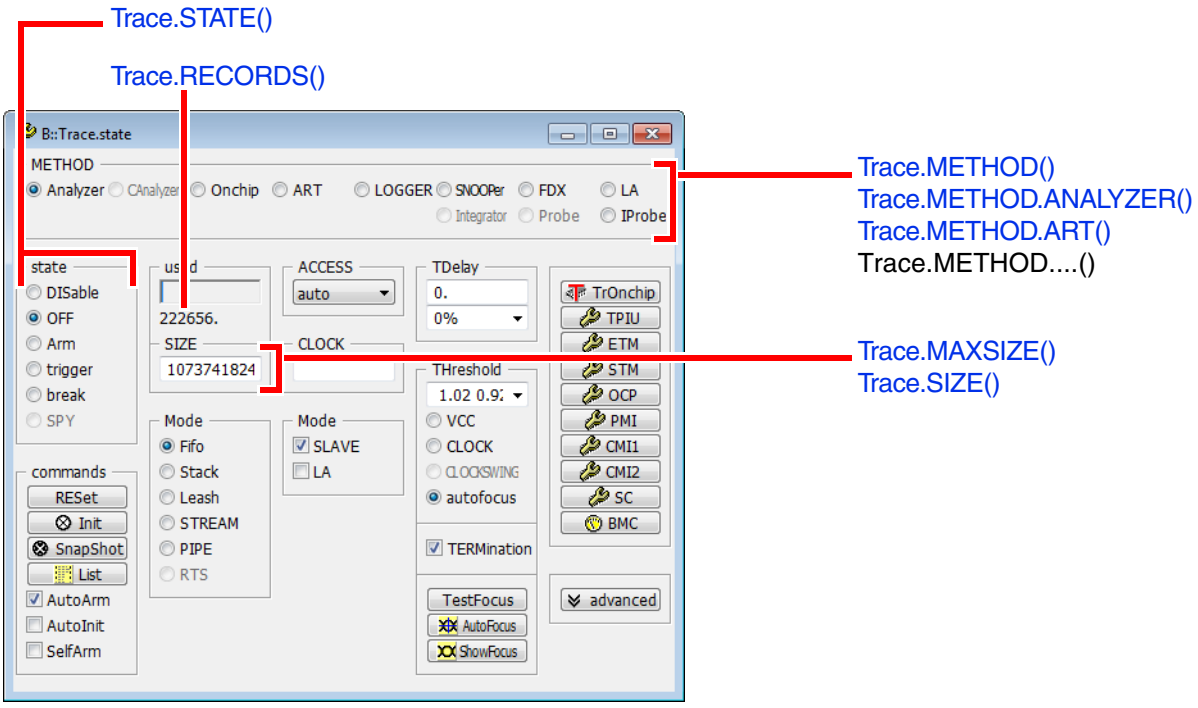
Syntax: **TPUBASE.ADDRESS()**

Returns the address where the TPU is located.

Return Value Type: Address.

Trace Functions

This figure provides an overview of the return values of some of the Trace functions. For descriptions of the illustrated functions and the functions not shown here, see below.



In This Section

See also

- Trace
- Trace.FIFOFULL()
- Trace.FIRST()
- Trace.FLOW()
- Trace.FLOW.ERRORS()
- Trace.MAXSIZE()
- Trace.METHOD()
- Trace.METHOD.Analyzer()
- Trace.METHOD.ART()
- Trace.METHOD.CAnalyzer()
- Trace.METHOD.FDX()
- Trace.METHOD.HAnalyzer()
- Trace.METHOD.Integrator()
- Trace.METHOD.IProbe()
- Trace.METHOD.LA()
- Trace.METHOD.LOGGER()
- Trace.METHOD.ONCHIP()
- Trace.METHOD.Probe()
- Trace.METHOD.SNOOPer()
- Trace.RECORD.ADDRESS()
- Trace.RECORD.DATA()
- Trace.RECORD.OFFSET()
- Trace.RECORD.TIME()
- Trace.RECORDS()
- Trace.SIZE()
- Trace.STATE()
- Trace.STATistic.COUNT()
- Trace.STATistic.EXIST()
- Trace.STATistic.IMAX()
- Trace.STATistic.IMIN()
- Trace.STATistic.Internal()
- Trace.STATistic.MAX()
- Trace.STATistic.MIN()
- Trace.STATistic.Total()
- Trace.TraceCONNECT()

Trace.FIRST() Get record number of first trace record

[build 71062 - DVD 09/2016]

Syntax: **Trace.FIRST()**

Returns the record number of the first record. The first record is the record with the lowest record number.

Return Value Type: Decimal value.

Trace.FLOW() TRUE if trace method is flow trace

[build 70748 - DVD 09/2016]

Syntax: **Trace.FLOW()**

Returns TRUE if the selected trace method is a flow trace.

Return Value Type: Boolean.

Trace.FLOW.ERRORS() Get number of flow errors / hard errors

Syntax: **Trace.FLOW.ERRORS()**

Returns the number of flow errors and hard errors found while processing the trace recording.

Return Value Type: Decimal value.

Please be aware that the return value of this function is the accumulated count of events that were encountered while processing the trace recording. All opened windows showing trace data contribute to this value. The value is reset when a new trace recording is made, or when the **Trace.FLOWSTART** or **Trace.FLOWPROCESS** command is executed.

The use of this function is only recommended if you want to find out if a specified part of a trace recording is error free. The part to be analyzed can be defined using **Trace.STATistic.FIRST** and **Trace.STATistic.LAST**. If the defined part is error free (and thus this function returns zero), the analysis results are reliable as well.

Example 1: This script shows how to return only the number of flow errors and hard errors in the trace that is currently visible within the **Trace.List** window. If you now start scrolling up or down in the **race.List** window or increase the window size, more trace data will be decoded, and thus the number of errors returned by the function may increase.

```
Trace.List

PRINT Trace.FLOW.ERRORS()

; scroll up or down in the window

PRINT Trace.FLOW.ERRORS()
```

Example 2: This script shows how to obtain the exact number of flow errors in the whole trace recording.

```
Trace.Find FLOWERROR /ALL
PRINT FOUND.COUNT()
```

Trace.FLOW.FIFOFULL() Get number of FIFO overflows

Syntax: **Trace.FLOW.FIFOFULL()**

Returns the number of **FIFO overflows** found while processing the trace recording.

Return Value Type: Decimal value.

Please be aware that the return value of this function is the accumulated count of events that were encountered while processing the trace recording. All opened windows showing trace data contribute to this value. The value is reset when a new trace recording is made, or when the **Trace.FLOWSTART** or **Trace.FLOWPROCESS** command is executed.

The use of this function is only recommended if you want to find out if a specified part of a trace recording is error free. The part to be analyzed can be defined using **Trace.STATistic.FIRST** and **Trace.STATistic.LAST**. If the defined part is error free (and thus this function returns zero), the analysis results are reliable as well.

The example below shows how to obtain the exact number of flow errors in the whole trace recording.

```
Trace.Find FIFOFULL /ALL
PRINT FOUND.COUNT()
```

Trace.MAXSIZE() Get max. size of trace buffer in records

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **Trace.MAXSIZE()**

Returns the maximum possible number of records.

Return Value Type: Decimal value.

Trace.METHOD() Currently configured trace method

[build 60615 - DVD 02/2015] [Go to figure]

Syntax: **Trace.METHOD()**

Returns the long form of the currently configured trace method.

Return Value Type: String.

Trace.METHOD.Analyzer() TRUE if the trace method is Analyzer

[Go to figure]

Syntax: **Trace.METHOD.Analyzer()**

Returns TRUE if the trace method is Analyzer.

The trace method is set with the command **Trace.METHOD.Analyzer**.

Return Value Type: Boolean.

Trace.METHOD.ART() TRUE if the trace method is ART

[Go to figure]

Syntax: **Trace.METHOD.ART()**

Returns TRUE if the trace method is ART.

The trace method is set with the command **Trace.METHOD.ART**.

Return Value Type: Boolean.

Trace.METHOD.CAnalyzer() TRUE if the trace method is CAnalyzer

[Go to figure]

Syntax: **Trace.METHOD.CAnalyzer()**

Returns TRUE if the trace method is CAnalyzer (CombiProbe).

The trace method is set with the command **Trace.METHOD.CAnalyzer**.

Return Value Type: Boolean.

Trace.METHOD.FDX() TRUE if the trace method is FDX

[Go to figure]

Syntax: **Trace.METHOD.FDX()**

Returns TRUE if the trace method is FDX.

The trace method is set with the command **Trace.METHOD.FDX**.

Return Value Type: Boolean.

Trace.METHOD.HAnalyzer() TRUE if the trace method is HAnalyzer

Syntax: **Trace.METHOD.HAnalyzer()**

Returns TRUE if the trace method is HAnalyzer (Trace-over-USB).

The trace method is set with the command **Trace.METHOD.HAnalyzer**.

Return Value Type: Boolean.

Trace.METHOD.Integrator() TRUE if the trace method uses the Integrator

[Go to figure]

Syntax: **Trace.METHOD.Integrator()**

Returns TRUE if the trace method uses the Integrator hardware analyzer.

The trace method is set with the command **Trace.METHOD.Integrator**.

Return Value Type: Boolean.

Trace.METHOD.IProbe() TRUE if the trace method uses the IProbe

[build 60615 - DVD 02/2015] [Go to figure]

Syntax: **Trace.METHOD.IProbe()**

Returns TRUE if the trace method uses the IProbe of PowerTrace II / PowerTrace III hardware analyzer.

The trace method is set with the command **Trace.METHOD.IProbe**.

Return Value Type: Boolean.

Trace.METHOD.LA() TRUE if the trace method is LA

[Go to figure]

Syntax: **Trace.METHOD.LA()**

Returns TRUE if the trace method is LA.

The trace method is set with the command **Trace.METHOD.LA**.

Return Value Type: Boolean.

Trace.METHOD.LOGGER() TRUE if the trace method is LOGGER

[Go to figure]

Syntax: **Trace.METHOD.LOGGER()**

Returns TRUE if the trace method is LOGGER.

The trace method is set with the command **Trace.METHOD.LOGGER**.

Return Value Type: Boolean.

Trace.METHOD.ONCHIP() TRUE if the trace method is ONCHIP

[Go to figure]

Syntax: **Trace.METHOD.ONCHIP()**

Returns TRUE if the trace method is ONCHIP.

The trace method is set with the command **Trace.METHOD.Onchip**.

Return Value Type: Boolean.

Trace.METHOD.Probe() TRUE if trace method uses the PowerProbe

[Go to figure]

Syntax: **Trace.METHOD.Probe()**

Returns TRUE if the trace method uses the PowerProbe hardware analyzer.

Return Value Type: Boolean.

Trace.METHOD.SNOOPer() TRUE if the trace method is SNOOPer

[Go to figure]

Syntax: **Trace.METHOD.SNOOPer()**

Returns TRUE if the trace method is SNOOPer.

The trace method is set with the command **Trace.METHOD.SNOOPer**.

Return Value Type: Boolean.

Trace.RECORD.ADDRESS() Get address recorded in trace record

[build 38764]

Syntax: **Trace.RECORD.ADDRESS(***<record_number>***)**

Returns the sampled address (access class and offset) from the specified record.

Parameter Type: Decimal value.

Return Value Type: Address.

Trace.RECORD.DATA() Get data recorded in trace record

[build 38764]

Syntax: **Trace.RECORD.DATA(***<record_number>***)**

Returns the sampled data of the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

Trace.RECORD.OFFSET() Get address in trace record as number

[build 38764]

Syntax: **Trace.RECORD.OFFSET(***<record_number>***)**

Returns the address-offset of the sampled address from the specified record.

Parameter Type: Decimal value.

Return Value Type: Hex value.

Example:

```
PRINT Trace.RECORD.OFFSET(-000009.)
```

Trace.RECORD.TIME() Get timestamp of trace record

[build 38764]

Syntax: **Trace.RECORD.TIME(***<record_number>***)**

Returns the timestamp of the specified record. For an example, see **Analyzer.RECORD.TIME()**.

Parameter Type: Decimal value.

Return Value Type: Time value.

Trace.RECORDS() Get number of used trace records

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **Trace.RECORDS()**

Returns the number of records currently recorded in the trace buffer.

If the state is **OFF** and the current mode is **STREAM**, this function will block until all buffered data has been received.

Return Value Type: Decimal value.

Trace.SIZE() Get current trace buffer size in records

[build 38323 - DVD 08/2012] [Go to figure]

Syntax: **Trace.SIZE()**

Returns the currently defined size of the trace buffer to be used for recording in records.

Return Value Type: Decimal value.

Trace.STATE() Get state of PowerTrace hardware

[build 54721 - DVD 09/2014] [Go to figure]

Syntax: **Trace.STATE()**

Returns the state of the PowerTrace hardware.

If the state is **OFF** and the current mode is **STREAM**, this function will block until all buffered data has been received.

Return Value Type: Hex value.

Return Value and Description:

0	**OFF** state
1	**Arm** state
2	**break** state
3	**trigger** state
4	**DISable** state
8	**SPY** state
9	**OFF** state, but data is still being processed (**PIPE** and **RTS** modes only)

Trace.STATistic.COUNT() Number of occurences of selected function

[build 121928 - DVD 09/2020]

Syntax: **Trace.STATistic.COUNT(**<*address*>**)**

Returns the number of occurences of the selected function in the trace statistic results.

Return Value Type: Decimal value.

Trace.STATistic.EXIST() TRUE if function exists in trace statistics

[build 121928 - DVD 09/2020]

Syntax: **Trace.STATistic.EXIST(**<*address*>**)**

Returns TRUE if the selected function exists in the trace statistic results.

Return Value Type: Boolean.

Trace.STATistic.IMAX() Longest time between function entry and exit

[build 121928 - DVD 09/2020]

Syntax: **Trace.STATistic.IMAX(**<*address*>**)**

Returns the longest time between function entry and exit without called sub-functions.

Return Value Type: Time value.

Trace.STATistic.IMIN() Shortest time between function entry and exit

[build 121928 - DVD 09/2020]

Syntax: **Trace.STATistic.IMIN(**<*address*>**)**

Returns the shortest time between function entry and exit without called sub-functions.

Return Value Type: Time value.

Trace.STATistic.Internal() Time spent within the selected function

[build 121928 - DVD 09/2020]

Syntax: **Trace.STATistic.Internal(**<*address*>**)**

Returns the time spent within the selected function in the trace statistic results.

Return Value Type: Time value.

Trace.STATistic.MAX() Maximum time of selected function

[build 121928 - DVD 09/2020]

Syntax: **Trace.STATistic.MAX(**<*address*>**)**

Returns the longest measured time it took to execute the function.

Return Value Type: Time value.

Trace.STATistic.MIN() Minimum time of selected function

[build 121928 - DVD 09/2020]

Syntax: **Trace.STATistic.MIN(**<*address*>**)**

Returns the shortest measured time it took to execute the function.

Return Value Type: Time value.

Trace.STATistic.Total() Total time of selected function

[build 121928 - DVD 09/2020]

Syntax: **Trace.STATistic.Total(**<*address*>**)**

Returns the total time within the selected function in the trace statistic results.

Return Value Type: Time value.

Syntax: **Trace.TraceCONNECT()**

Returns the name of the currently selected trace sink of the SoC. In case no trace-sink is selected/available, the function returns **NONE**. The trace sink is selected with the **<trace>.TraceCONNECT** command.

Return Value Type: String.

Example: See **Onchip.TraceCONNECT()**.

In This Section

See also

■ TRACEPORT ❑ TRACEPORT.LaneCount()

TRACEPORT.LaneCount() Number of serial lanes

[build 104159 - DVD 02/2019]

Syntax: **TRACEPORT.LaneCount(***<index>***)**

<index>: **1.** … *<n>*

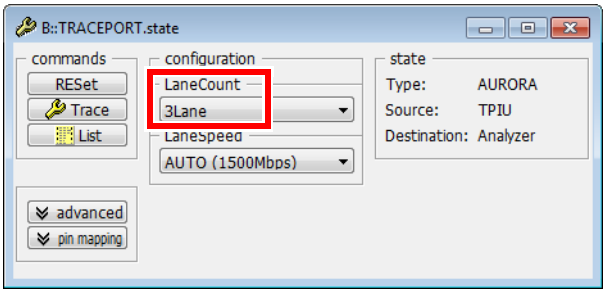
Returns the number of serial lanes for the specified traceport *<index>*.

Parameter Type: Decimal value.

Return Value Type: Decimal value.

Example 1:

```
TRACEPORT.state                    ;optional step: open window for
                                   ;traceport 1. (= <index>)
ECHO TRACEPORT.LaneCount(1.)       ;returns 3., see screenshot below
```



Example 2:

```
TRACEPORT2.state                   ;optional step: open window for
                                   ;traceport 2. (= <index>)
ECHO TRACEPORT.LaneCount(2.)
```

TRACK Functions

In This Section

See also

- TRACK.ADDRESS()
- TRACK.COLUMN()
- TRACK.LINE()
- TRACK.RECORD()
- TRACK.STRing()
- TRACK.TIME()

TRACK.ADDRESS() Get tracking address

Syntax: **TRACK.ADDRESS()**

Returns the tracking address e.g. after search or memory test commands, such as **Data.Find** or **Data.Test**.

Return Value Type: Address.

Example:

```
Data.Find flags++0xFF 0x01
IF FOUND()==TRUE()
(
     Data.Set   TRACK.ADDRESS() 0xAA
     Data.Print TRACK.ADDRESS()
)
```

TRACK.COLUMN() Number of column where the found item starts

Syntax: **TRACK.COLUMN()**

After a successful search operation, this function returns the column number.

Return Value Type: Decimal value.

TRACK.LINE() Number of line containing the found item

Syntax: **TRACK.LINE()**

After a successful search operation, this function returns the line number.

Return Value Type: Decimal value.

Example: The function **TRACK.LINE()** is used to automatically scroll to the required line number in the **TYPE** window, where the search string was found.

```
LOCAL &file &line
&file="~~/demo/arm/compiler/gnu/src/sieve.c"

FIND &file , "main("     ;search for the string "main(" in whole file
&line=TRACK.LINE()       ;return the line number of the first occurrence

IF FOUND()==TRUE()       ;if found, open file in TYPE window and
(                        ;scroll to the &line where the string was found
   TYPE &file &line /LineNumbers
)
```

See also: **FIND** and **ComPare**.

TRACK.RECORD() Number of record containing the found item

Syntax: **TRACK.RECORD()**

After a successful search operation, this function returns the record number. For an example, see **Analyzer.TRACK.RECORD()**.

Return Value Type: Decimal value.

TRACK.STRing() Current selection in a TRACE32 window

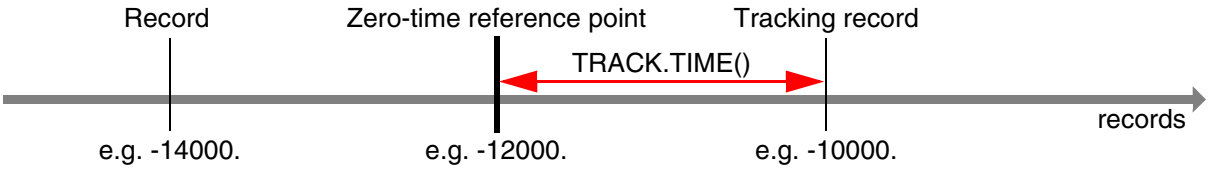
Syntax: **TRACK.STRing()**
SELECTION.STRing() (deprecated)

Returns a singe-line or multi-line selection made by a user in a TRACE32 window. For information about how to return values and selections from user-defined dialogs, see **DIALOG Functions**.

Return Value Type: String.

Syntax: **TRACK.TIME()**

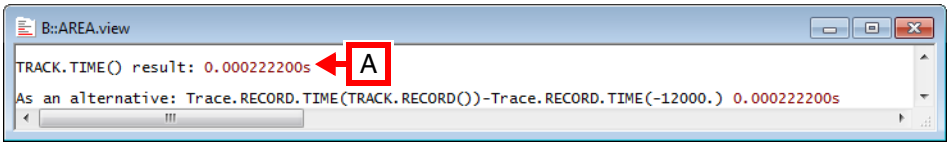
Returns the timestamp of the current tracking record in relation to the zero-time reference point.

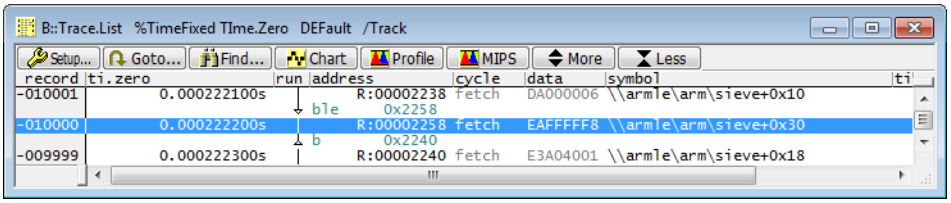


Return Value Type: Time value.

Example:

```
;set the zero-time reference point to record no. -12000.
ZERO.offset Trace.RECORD.TIME(-12000.)

;set the tracking record to the record -10000.
Trace.TRACK -10000.

PRINT "TRACK.TIME() result: " %COLOR.MAROON TRACK.TIME()  ;see [A] below

;display the result in the trace listing with the ti.zero column as the
;first column followed by all DEFault columns
Trace.List   %TimeFixed TIme.Zero   DEFault   /Track
```





TRANS Functions (Debugger Address Translation)

In This Section

See also

- TRANSlation
- TRANS.INTERMEDIATE()
- TRANS.INTERMEDIATEEX()
- TRANS.LINEAR()
- TRANS.LINEAREX()
- TRANS.LIST.LOGRANGE()
- TRANS.LIST.PHYSADDR()
- TRANS.LOGICAL()
- TRANS.PHYSICAL()
- TRANS.PHYSICALEX()
- TRANS.TABLEWALK()
- TRANS.ENABLE()
- TRANS.INTERMEDIATE.VALID()
- TRANS.INTERMEDIATEEX.VALID()
- TRANS.LINEAR.VALID()
- TRANS.LINEAREX.VALID()
- TRANS.LIST.NUMBER()
- TRANS.LIST.TYPE()
- TRANS.LOGICAL.VALID()
- TRANS.PHYSICAL.VALID()
- TRANS.PHYSICALEX.VALID()

TRANS.LIST.NUMBER() Number of TRANS.List entries

Syntax 1:	**TRANS.LIST.NUMBER()**
Syntax 2:	**TRANS.LIST.NUMBER.ZONE(***<address>***)**
<address>:	*<access_class>***:**[*<machine_id>***:::]0x0**

Both functions return the number of entries in the **TRANSlation.List** window.

Syntax 1: The queried entry is located in the currently active zone of the core.
Syntax 2: The queried entry is located in the zone that is selected with *<address>*.

Return Value Type: Decimal value.

The type of entries in the **TRANSlation.List** window which can be queried are

- static translations created with command **TRANSlation.Create** or **MMU.Scan**
- protected address ranges created with command **TRANSlation.Protect**
- transparent entries created with command **TRANSlation.TRANSparent**
- common ranges created with command **TRANSlation.COMMON** or **TRANSlation.COMMON.ADD**

Note that default translations created with command **MMU.FORMAT** and displayed in the first lines of the **TRANSlation.List** window can not be queried with this set of functions. Use function **MMU.DEFAULTTRANS.<range>** to query default translations.

Parameter and Description:

<access_class>	Mandatory if **SYStem.Option.ZoneSPACES** is **ON**. See access class in the glossary.pdf.
<machine_id>	Mandatory if **SYStem.Option.MACHINESPACES** is **ON**. See machine ID in the glossary.pdf.
0x0	Fixed *<address>* suffix.

TRANS.LIST.LOGRANGE() Query TRANS.List entry

Syntax 1: **TRANS.LIST.LOGRANGE(***<entry_index>***)**

Syntax 2: **TRANS.LIST.LOGRANGE.ZONE(***<entry_index>, <address>***)**

<address>: *<access_class>***:**[*<machine_id>***:::]0x0**

Both functions return the logical address range of entry *<entry_index>* in the in the **TRANSlation.List** window.

Syntax 1: The queried entry is located in the currently active zone of the core.
Syntax 2: The queried entry is located in the zone that is selected with *<address>*.

Return Value Type: Address.range

Note: an address offset of 0xFFFFFFFF (for 32-bit architectures) or 0xFFFFFFFFFFFFFFFF (for 64-bit architectures) is returned if *<entry_index>* does not select any existing entry.

Parameter and Description:

<entry_index>	the index of the queried entry. Positive numbers query the entries in the TRANSlation.List window, starting with 0 for the first entry in the window. Negative numbers query entries in a reverse sequence, starting with the last entry.
<access_class>	Mandatory if **SYStem.Option.ZoneSPACES** is **ON**. See access class in the glossary.pdf.
<machine_id>	Mandatory if **SYStem.Option.MACHINESPACES** is **ON**. See machine ID in the glossary.pdf.
0x0	Fixed *<address>* suffix.

Example - Syntax 2:

```
;optional step: list all translation entries
TRANSlation.list

; display the logical address range of the first entry in the
TRANSlation.List window
PRINT TRANS.LIST.LOGRANGE(0)

; display the logical address range of the second entry in the
TRANSlation.List window for machine '2:::' in the nonsecure zone 'N'
PRINT TRANS.LIST.LOGRANGE.ZONE(1, N:2:::0)

; display the logical address range of the last entry in the
TRANSlation.List window
PRINT TRANS.LIST.LOGRANGE(-1)

; display the logical address range of the second last entry in the
TRANSlation.List window
PRINT TRANS.LIST.LOGRANGE(-2)
```

TRANS.LIST.PHYSADDR() Query TRANS.List entry

Syntax 1:	**TRANS.LIST.PHYSADDR(***<entry_index>***)**
Syntax 2:	**TRANS.LIST.PHYSADDR.ZONE(***<entry_index>***,** *<address>***)**
<address>:	*<access_class>***:**[*<machine_id>***:::]0x0**

Both functions return the physical base address of entry *<entry_index>* in the in the **TRANSlation.List** window.

Syntax 1: The queried entry is located in the currently active zone of the core.
Syntax 2: The queried entry is located in the zone that is selected with *<address>*.

Return Value Type: Address.

Note: an address offset of 0xFFFFFFFF (for 32-bit architectures) or 0xFFFFFFFFFFFFFFFF (for 64-bit architectures) is returned if *<entry_index>* does not select any existing entry.

Parameter and Description:

<entry_index>	the index of the queried entry. Positive numbers query the entries in the TRANSlation.List window, starting with 0 for the first entry in the window. Negative numbers query entries in a reverse sequence, starting with the last entry.
<access_class>	Mandatory if **SYStem.Option.ZoneSPACES** is **ON**. See access class in the glossary.pdf.
<machine_id>	Mandatory if **SYStem.Option.MACHINESPACES** is **ON**. See machine ID in the glossary.pdf.
0x0	Fixed *<address>* suffix.

TRANS.LIST.TYPE() Query TRANS.List entry

Syntax 1:	**TRANS.LIST.TYPE(***<entry_index>***)**
Syntax 2:	**TRANS.LIST.TYPE.ZONE(***<entry_index>, <address>***)**
<address>:	*<access_class>***:**[*<machine_id>***:::]0x0**

Both functions return one of the follow strings, indicating the type of entry *<entry_index>* in the in the **TRANSlation.List** window:

TRANSLATION	the queried entry is a translation entry created with command **TRANSlation.Create** or **MMU.Scan**
TRANSPARENT	the queried entry is a transparent entry created with command **TRANSlation.TRANSparent**
PROTECTED	the queried entry is a protected entry created with command **TRANSlation.Protect**
COMMON	the queried entry is a common address range created with command **TRANSlation.COMMON** or **TRANSlation.COMMON.ADD**
empty string	*<entry_index>* does not select a valid entry

Syntax 1: The queried entry is located in the currently active zone of the core.
Syntax 2: The queried entry is located in the zone that is selected with *<address>*.

Return Value Type: String.

Parameter and Description:

<entry_index>	the index of the queried entry. Positive numbers query the entries in the TRANSlation.List window, starting with 0 for the first entry in the window. Negative numbers query entries in a reverse sequence, starting with the last entry.
<access_class>	Mandatory if **SYStem.Option.ZoneSPACES** is **ON**. See access class in the glossary.pdf.
<machine_id>	Mandatory if **SYStem.Option.MACHINESPACES** is **ON**. See machine ID in the glossary.pdf.
0x0	Fixed *<address>* suffix.

TRANS.ENABLE() TRUE if address translation is enabled

Syntax: **TRANS.ENABLE()**

Returns TRUE if the address translation of the debugger is enabled with the command **TRANSlation.ON**.

Return Value Type: Boolean.

TRANS.INTERMEDIATE() Convert a guest logical address

ARM and Intel®

Syntax 1: **TRANS.INTERMEDIATE(***<address>***)**
MMU.INTERMEDIATE(*<address>***)** (alias)

Syntax 2: **TRANS.INTERMEDIATEEX(***<address>***)**
MMU.INTERMEDIATEEX(*<address>***)** (alias)

Converts a guest logical address to an intermediate address if the stage 2 address translation (for ARM targets) or Extended Physical Translation (for Intel® targets) is enabled.

TRANS.INTERMEDIATE() behaves like **TRANS.PHYSICAL()** and converts *<address>* to a physical address if:

- *<address>* is not a *guest* logical address
- or the stage 2 translation / Extended Physical Translation in the target MMU is not available or disabled.

Syntax 1: No access class expansion of *<address>* is done prior to the translation. Tries to translate *<address>* as it is. Use this syntax to do a translation only. The result is independent of the current CPU mode.

Syntax 2: The access class of *<address>* is expanded prior to the translation, missing information is added from the current PC's access class. Use this syntax to get the same translation result as a standard access to the target. The translation result may depend on the current CPU mode**.**

Parameter Type: Address.

Return Value Type: Address.

TRANS.INTERMEDIATE.VALID() TRUE if address translation is valid

ARM and Intel®

Syntax 1:	**TRANS.INTERMEDIATE.VALID(***<address>***)** **MMU.INTERMEDIATE.VALID(***<address>***)** (alias)
Syntax 2:	**TRANS.INTERMEDIATEEX.VALID(***<address>***)** **MMU.INTERMEDIATEEX.VALID(***<address>***)** (alias)

Checks whether a valid guest logical-to-intermediate address translation exists for the specified address.

Syntax 1: No access class expansion of *<address>* is done prior to the translation.

Syntax 2: The access class of *<address>* is expanded prior to the translation, missing information is added from the current PC's access class.

Parameter Type: Address.

Return Value Type: Boolean.

TRANS.LINEAR() Convert logical to linear address

Intel®

Syntax 1:	**TRANS.LINEAR(***<address>***)** **MMU.LINEAR(***<address>***)** (alias)
Syntax 2:	**TRANS.LINEAREX(***<address>***)** **MMU.LINEAREX(***<address>***)** (alias)

Converts a logical address to a linear address.

Syntax 1: No access class expansion of *<address>* is done prior to the translation. Tries to translate *<address>* as it is. Use this syntax to do a translation only. The result is independent of the current CPU mode.

Syntax 2: The access class of *<address>* is expanded prior to the translation, missing information is added from the current PC's access class. Use this syntax to get the same translation result as a standard access to the target. The translation result may depend on the current CPU mode**.**

Parameter Type: Address.

Return Value Type: Address.

TRANS.LINEAR.VALID() TRUE if address translation is valid

Intel®

Syntax 1:	**TRANS.LINEAR.VALID(***<address>***)** **MMU.LINEAR.VALID(***<address>***)** (alias)
Syntax 2:	**TRANS.LINEAREX.VALID(***<address>***)** **MMU.LINEAREX.VALID(***<address>***)** (alias)

Checks whether a valid logical-to-linear address translation exists for the specified address.

Syntax 1: No access class expansion of *<address>* is done prior to the translation.

Syntax 2: The access class of *<address>* is expanded prior to the translation, missing information is added from the current PC's access class.

Parameter Type: Address.

Return Value Type: Boolean.

TRANS.LOGICAL() Convert physical to logical address

Syntax:	**TRANS.LOGICAL(***<physical_address>***)** **MMU.LOGICAL(***<physical_address>***)** (alias)

Converts a physical address to a logical address using the TRACE32 static translation table. No access class expansion is done prior to the translation. This function tries to translate *<address>* as it is. The result is independent of the current CPU mode.

Use the command **TRANSlation.List** to view the static translation table.
If the static translation table contains more than one valid entry matching the specified *<physical_address>*, then the first entry will be used for the address conversion.

NOTE:

- **TRANS.LOGICAL(***<physical_address>***)** does not search in page tables.
- If you want to search through the entries of a page table, you must first use the command **MMU.SCAN** to read the contents of the page table into the static translation table.
- If you want to search through the entries of a page table, you can also use the command **MMU.INFO** *<physical_address>* to search the system page tables for entries that yield *<physical_address>*.

Parameter Type: Address.

Return Value Type: Address.

TRANS.LOGICAL.VALID() TRUE if address translation is valid

Syntax: **TRANS.LOGICAL.VALID(***<physical_address>***)**
MMU.LOGICAL.VALID(*<physical_address>***)** (alias)

Checks whether a valid physical-to-logical address translation exists for the specified physical address. Only the static translation table will be searched, no page tables. No access class expansion of *<address>* is done prior to the translation.

Parameter Type: Address.

Return Value Type: Boolean.

TRANS.PHYSICAL() Convert logical to physical address

Syntax 1: **TRANS.PHYSICAL(***<address>***)**
MMU.PHYSICAL(*<address>***)** (alias)

Syntax 2: **TRANS.PHYSICALEX(***<address>***)**
MMU.PHYSICALEX(*<address>***)** (alias)

Converts a logical address to a physical address.

Syntax 1: No access class expansion of *<address>* is done prior to the translation. Tries to translate *<address>* as it is. Use this syntax to do a translation only. The result is independent of the current CPU mode.

Syntax 2: The access class of *<address>* is expanded prior to the translation, missing information is added from the current PC's access class. Use this syntax to get the same translation result as a standard access to the target. The translation result may depend on the current CPU mode**.**

Parameter Type: Address.

Return Value Type: Address.

Use **ADDRESS.PHYSICAL()** to check whether the translation was successful and the result is really a physical address.

If the translation failed, **ADDRESS.PHYSICAL(TRANS.PHYSICAL(***<address>***))** will return FALSE.

Example:

```
; This is an example for the ARM simulator
; It demonstrates the difference between functions
; TRANS.PHYSICAL() and TRANS.PHYSICALEX()

; Select a CPU with trust zone (nonsecure/secure mode)
SYStem.CPU CORTEXA5
AREA.View

; Let's make the static address translation zone specific so that it is
; easier to demonstrate the difference between TRANS.PHYSICAL()
; and TRANS.PHYSICALEX()
SYStem.Option.ZoneSPACES ON

SYStem.UP

Register.Set M 0x13     ; set CPU to supervisor mode
Register.Set NS 1       ; set CPU to nonsecure mode
; Now the PC's access class is NSD:

; Define a static translation for a nonsecure address range
TRANSlation.Create N:0xA0000000--0xAFFFFFFFF A:0x30000000

TRANSlation.List        ; show the static translation list
TRANSlation.ON          ; enable the debugger address translation

; 1. Try to translate an address with unspecific access class
PRINT TRANS.PHYSICAL(D:0xA0005000)
;    the result is D:0xA0005000 because there is no translation
;    for D:0xA0005000 with SYStem.Option.ZoneSPACES ON
;    There is a translation for nonsecure addresses only.

; 2. Try to translate an address with unspecific access class,
;    now with access class expansion using the PC's access class
PRINT TRANS.PHYSICALEX(D:0xA0005000)
;    the result is ANSD:0x0:0x30005000 because D:0xA0005000 becomes
;    NSD:0xA0005000 after the access class is expanded

; 3. Now do the same like TRANS.PHYSICALEX(), but in two distinct steps
PRINT TRANS.PHYSICAL(ADDRESS.EXPANDACCESS(D:0xA0005000))
;    the result is A:0x30005000 again, the same as in 2.
```

TRANS.PHYSICAL.VALID() TRUE if address translation is valid

Syntax 1:	**TRANS.PHYSICAL.VALID(***<address>***)** **MMU.PHYSICAL.VALID()** (alias)
Syntax 2:	**TRANS.PHYSICALEX.VALID(***<address>***)** **MMU.PHYSICALEX.VALID()** (alias)

Checks whether a valid logical-to-physical address translation exists for the specified address.

Syntax 1: No access class expansion of *<address>* is done prior to the translation.

Syntax 2: The access class of *<address>* is expanded prior to the translation, missing information is added from the current PC's access class.

Parameter Type: Address.

Return Value Type: Boolean.

TRANS.TABLEWALK() TRUE if address translation table walk is ON

Syntax:	**TRANS.TABLEWALK()**

Returns TRUE if the table walk for address translation is enabled in the debugger with the command **TRANSlation.TableWalk ON**.

Return Value Type: Boolean.

TSS()

TSS base address

Syntax: **TSS()**

Returns the TSS base address of the last loaded object file (only 80386).

Return Value Type: Hex value.

Var Functions

In This Section

See also

- Var
- Var.ADDRESS()
- Var.BITPOS()
- Var.BITSIZE()
- Var.END()
- Var.EXIST()
- Var.FVALUE()
- Var.ISBIT()
- Var.RANGE()
- Var.SIZEOF()
- Var.STRing()
- Var.TYPEOF()
- Var.VALUE()

Var.ADDRESS() Address of HLL expression

Syntax: **Var.ADDRESS(**<*hll_expression*>**)**

Returns the address of the HLL expression.

Parameter Type: String.

Return Value Type: Address.

Example:

```
Data.Print Var.ADDRESS(flags)
Data.Print Var.ADDRESS(flags[3])
```

Var.BITPOS() Bit position inside a C bit field

Syntax: **Var.BITPOS(**<*hll_expression*>**)**

Returns the start bit position of an element inside a bit field.

Parameter Type: String.

Return Value Type: Hex value.

Example:

```
PRINT Var.BITPOS(vbfield.d)
```

Var.BITSIZE() Size of bit field element

Syntax: **Var.BITSIZE(***<hll_expression>***)**

Returns the size in bits of a bit field element.

Parameter Type: String.

Return Value Type: Hex value.

Example:

```
PRINT %Decimal Var.BITSIZE(vbfield.f)
```

Var.END() Last address of HLL expression

Syntax: **Var.END(***<hll_expression>***)**

Returns the last address occupied by the HLL expression.

Parameter Type: String.

Return Value Type: Address.

Example:

```
Data.Print Var.END(vbfield)
```

Syntax: **Var.EXIST(**<*hll_expression*>**)**

Returns TRUE if an HLL expression is syntactically valid or if an HLL variable exists.

Parameter Type: String.

Return Value Type: Boolean.

Example 1:

```
;your code
Data.LOAD.Elf "armle.axf" /StripPATH /LowerPATH
;your code

PRINT Var.EXIST(sieve)        ;returns TRUE
PRINT Var.EXIST(flags)        ;returns TRUE
```

Example 2: Remember that inline comments for **Var.*** commands must start with **//**.

```
PRINT Var.EXIST(\val1)        ;returns FALSE

Var.NEWLOCAL int \val1       //let's now create the TRACE32-internal
                              ;variable \val1
PRINT Var.EXIST(\val1)        ;returns TRUE because now it exists

Var.set \val1=0x42           //initialize the TRACE32-internal variable

PRINT Var.EXIST(\val1==0x5)   ;returns TRUE because syntactically
                              ;\val1==0x5 is a valid HLL expression
```

Var.FVALUE() Contents of HLL expression

Syntax: **Var.FVALUE(**<*hll_expression*>**)**

Returns the contents of the HLL expression.

Parameter Type: String.

Return Value Type: Float.

Example:

```
PRINT Var.FVALUE(ast.float)
PRINT Var.FVALUE(i)
PRINT Var.FVALUE(ast.left->x)
```

Var.ISBIT() TRUE if HLL expression is a bit field element

Syntax: **Var.ISBIT(**<*hll_expression*>**)**

Returns whether the HLL expression is a bit field element or not.

Parameter Type: String.

Return Value Type: Boolean.

Example:

```
PRINT Var.ISBIT(vbfield.f)
PRINT Var.ISBIT(vbfield)
```

Var.RANGE() Address range of HLL expression

Syntax: **Var.RANGE(**<*hll_expression*>**)**

Returns the address range occupied by the HLL expression in memory.

Parameter Type: String.

Return Value Type: Address range.

Example:

```
PRINT Var.RANGE(flags)      ;returns the address range D:0x6EA4--0x6EB6


Data.Print Var.RANGE(flags)

Data.Find  Var.RANGE(flags) 0x00
IF FOUND()
(
     Data.Set   TRACK.ADDRESS() 0xBB
     Data.Print TRACK.ADDRESS()
)
ENDDO
```

Var.SIZEOF() Size of HLL expression

Syntax: **Var.SIZEOF(**<*hll_expression*>**)**

Returns the size occupied by the HLL expression in memory.

Parameter Type: String.

Return Value Type: Hex value.

Example:

```
Data.Print flags++Var.SIZEOF(flags)

Data.Find  flags++Var.SIZEOF(flags) 0x00
IF FOUND()
(
     Data.Set   TRACK.ADDRESS() 0xDD
     Data.Print TRACK.ADDRESS()
)
ENDDO
```

Syntax: **Var.STRing(***<hll_expression>***)**

Returns a zero-terminated string, if *<hll_expression>* is a pointer to character or an array of characters. Returns a string that represents the variable contents otherwise.

Parameter Type: String.

Return Value Type: String.

Example:

```
&enumvalue=Var.STRing(ptr->member)
PRINT "&enumvalue"
```

Var.TYPEOF() Type of HLL expression

[build 28656 - DVD 06/2011] [Example]

Syntax: **Var.TYPEOF(***<hll_expression>***)**

Returns the type of the HLL expression. The expressions can also be TRACE32-internal variables, which are created with the commands **Var.NEWLOCAL** and **Var.NEWGLOBAL**.

Parameter Type: String.

Return Value Type: String.

Example:

```
;Create some TRACE32-internal variables, integer \val1 and
;character array \myStr, on the local PRACTICE stack frame
;Use the backslash for TRACE32-internal variables: '\val1', '\myStr'
Var.NEWLOCAL  int   \val1
Var.NEWLOCAL  char[6][20] \myStr

//see Var.NEWLOCAL command for information about the local PRACTICE
//stack frame and on how to initialize TRACE32-internal variables

;Returns the type of the TRACE32-internal variables as a string
PRINT Var.TYPEOF(\val1)  ;Returns in this case: "int"
PRINT Var.TYPEOF(\myStr) ;Returns in this case: "char [6][20]"

;Returns the type of the HLL expression as a string
;Omit the backslash for HLL expressions, here for 'flags'
PRINT Var.TYPEOF(flags)  ;Returns in this case: "unsigned char [19]"
```

Syntax: **Var.VALUE(**<*hll_expression*>**)**

Returns the contents of the HLL expression.

Parameter Type: String.

Return Value Type: Hex value.

Example:

```
PRINT Var.VALUE(ast.field1)
PRINT Var.VALUE(i)
PRINT Var.VALUE(ast.left->count)
PRINT %Decimal Var.VALUE(func5(7,8,9))
```

VCO() Frequency of VCO generator

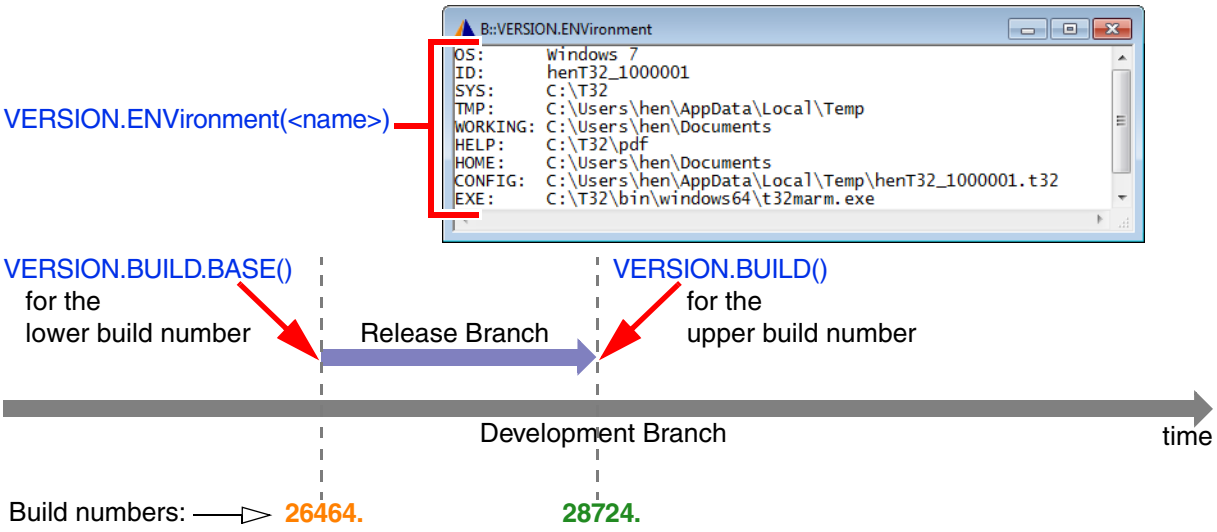
Syntax: **VCO()**

Returns the set frequency of the VCO generator.

Return Value Type: Decimal value.

VERSION Functions

This figure provides an overview of the return values of some of the **VERSION** functions. For descriptions of the illustrated functions and the functions not shown here, see below.



For more information, see also:

- **"Appendix - About the TRACE32 Software Version Numbers"** (ide_user.pdf)
- **"LICENSE Functions"** (ide_func.pdf)

In This Section

See also

- ■ VERSION
- ❑ VERSION.BUILD.BASE()
- ❑ VERSION.DATE()
- ❑ VERSION.FirmWare.DEBUG()
- ❑ VERSION.SERIAL()
- ❑ VERSION.SERIAL.DEBUG()
- ❑ VERSION.SERIAL.NEXUSadapter()
- ❑ VERSION.SERIAL.POWERTRACEAUXPORT()
- ❑ VERSION.SERIAL.SERialPort1()
- ❑ VERSION.SERIAL.WHISKER()
- ❑ VERSION.SOFTWARE.TYPE()
- ❑ VERSION.BUILD()
- ❑ VERSION.CABLE()
- ❑ VERSION.ENVironment()
- ❑ VERSION.LICENSE()
- ❑ VERSION.SERIAL.CABLE()
- ❑ VERSION.SERIAL.Integrator()
- ❑ VERSION.SERIAL.POWERPROBE()
- ❑ VERSION.SERIAL.PREPROcessor()
- ❑ VERSION.SERIAL.TRACE()
- ❑ VERSION.SOFTWARE()

VERSION.BUILD() Upper build number

[Go to figure]

Syntax: **VERSION.BUILD()**

Returns the upper build number of TRACE32, e.g. **28724.** Alias for **SOFTWARE.BUILD()**.
The same version information is displayed in the **VERSION.SOFTWARE** window.

The **VERSION.BUILD()** number is greater than the **VERSION.BUILD.BASE()** number if the TRACE32 executable is a build from a branch with some changes (e.g. in the case of a release branch after a feature freeze with included bug-fixes).

The **VERSION.BUILD()** number equals the **VERSION.BUILD.BASE()** number if the TRACE32 executable is a snapshot build *from the development branch*. A snapshot build is also referred to as an interim build.

Return Value Type: Decimal value.

VERSION.BUILD.BASE() Lower build number

[build 15283 - DVD 10/2008] [Go to figure]

Syntax: **VERSION.BUILD.BASE()**

Returns the lower build number of TRACE32, e.g. **26464.** Alias for **SOFTWARE.BUILD.BASE()**.

Return Value Type: Decimal value.

VERSION.CABLE() Hardware version of debug cable

Syntax: **VERSION.CABLE()**

Returns the hardware version of the debug cable.

Return Value Type: Decimal value.

VERSION.DATE() Version date YYYY/MM

[build 32168 - DVD 02/2012]

Syntax: **VERSION.DATE()**

Returns the date of the main software in the form YYYY/MM. e.g. "2016/07"

Return Value Type: String.

VERSION.ENVironment() TRACE32 environment setting

[build 38327 - DVD 08/2012] [Go to figure]

Syntax: **VERSION.ENVironment(***<name>***)**

Returns a single TRACE32 environment setting as shown in the **VERSION.ENVironment** window.

Parameter and Description:

<name>	**Parameter Type**: String. Specify the environment setting by name.

Return Value Type: String.

Example:

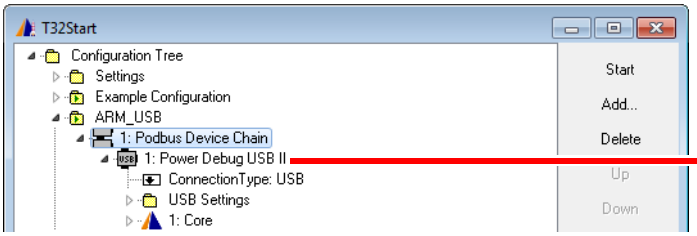
```
;returns the path of the used TRACE32 config file
PRINT VERSION.ENVironment(CONFIG)

;returns the version number of the used QT-Library
PRINT VERSION.ENVironment(QT)
```

VERSION.FirmWare.DEBUG() Version number of firmware

[build 34337 - DVD 02/2012]

Syntax: **VERSION.FirmWare.DEBUG()**

Returns the version number of the firmware of a PowerDebug or PowerTrace module - if the module is the first Podbus device. Otherwise it returns zero.



Return Value Type: Float.

VERSION.LICENSE() Deprecated

Syntax: **VERSION.LICENSE()** (deprecated)
Use LICENSE.SERIAL() instead.

VERSION.SERIAL() Serial number

Syntax: **VERSION.SERIAL()**

Returns the serial number of the system.

Return Value Type: String.

VERSION.SERIAL.CABLE() First serial number of debug cable

[build 21341 - DVD 04/2010]

Syntax: **VERSION.SERIAL.CABLE()**

Returns the first serial number of the plugged debug cable. It is the same value that is also shown in the **VERSION.HARDWARE** window. To return all serial numbers, use **LICENSE.SERIAL()**.

Return Value Type: String.

VERSION.SERIAL.DEBUG() Serial number of debug module

[build 11815 - DVD 10/2008]

Syntax: **VERSION.SERIAL.DEBUG()**

Returns the serial number of the debug module. It is the same value that is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

VERSION.SERIAL.Integrator() Serial number of PowerIntegrator

[build 59902 - DVD 02/2015]

Syntax: **VERSION.SERIAL.Integrator()**

Returns the serial number of the PowerIntegrator or PowerIntegrator-II. It is the same value that is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

VERSION.SERIAL.NEXUSadapter() Serial number of nexus adapter

[build 59902 - DVD 02/2015]

Syntax: **VERSION.SERIAL.NEXUSadapter()**

Returns the serial number of the nexus adapter or preprocessor. It is the same value that is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

VERSION.SERIAL.PREPROcessor() Serial number of preprocessor

[build 11815 - DVD 10/2008]

Syntax: **VERSION.SERIAL.PREPROcessor()**

Returns the serial number of the preprocessor. It is the same value that is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

VERSION.SERIAL.POWERPROBE() Serial number of PowerProbe

[build 59902 - DVD 02/2015]

Syntax: **VERSION.SERIAL.POWERPROBE()**

Returns the serial number of the PowerProbe. It is the same value that is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

VERSION.SERIAL.POWERTRACEAUXPORT() S/N of device at PT aux port

[build 133662 - DVD 02/2021]

Syntax: **VERSION.SERIAL.POWERTRACEAUXPORT()**

Returns the serial number of the Lauterbach device plugged to the AUX PORT of a PowerTrace. It is the same value that is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

VERSION.SERIAL.SERialPort1() S/N of device at Serial Port 1 of PT Serial

[build 156822 - DVD 02/2023]

Syntax: **VERSION.SERIAL.SERialPort1()**

Returns the serial number of the Lauterbach device (adapter or preprocessor) plugged to the Serial Port 1 of a PowerTrace Serial.

Return Value Type: String.

VERSION.SERIAL.WHISKER() S/N of whiskers at CombiProbe or µTrace

[build 133662 - DVD 02/2021]

Syntax: **VERSION.SERIAL.WHISKER(**<*int*>**)**

Returns the serial number of the whisker cable connected to the connector specified by <*int*> plugged to a CombiProbe, µTrace (MicroTrace), or QuadProbe or.
It is the same value that is also shown in the **VERSION.HARDWARE** window.
For the meaning of the <*int*> Parameter see command **ID.WHISKER()**.
Note, that only some whisker actually contain a serial number. For all the others, the function returns an empty string.

Return Value Type: String.

Syntax: **VERSION.SERIAL.TRACE()**

Returns the serial number of the trace module. It is the same value that is also shown in the **VERSION.HARDWARE** window.

Return Value Type: String.

Syntax: **VERSION.SOFTWARE()**

Returns the version of the main software. Alias for **SOFTWARE.VERSION()**.

Return Value Type: String.

Return Value Format: *<type>.<year>.<month>.<build_number>* where the building block *<type>* can be one of the following:

- **R**: release build
- **P**: pre-release build
- **N**: nightly build
- **S**: interim build (“snapshot”)
- **F**: feature build

For more information about the *<type>* building blocks, see **“Appendix - About the TRACE32 Software Version Numbers”** in PowerView User’s Guide, page 123 (ide_user.pdf).

Examples:

```
PRINT VERSION.SOFTWARE()                    ;print the software version
```

```
;check whether the TRACE32 software being used matches a certain release
IF VERSION.SOFTWARE()!="R.2010.07.000024615"
   PRINT "wrong TRACE32 SW started"
```

Syntax: **VERSION.SOFTWARE.TYPE()**

Returns the version type of the main software.

Return Value Type: String.

Return Value Format: *<type>* :

- **R**: **r**elease build
- **P**: **p**re-release build
- **N**: **n**ightly build
- **S**: interim build ("**s**napshot")
- **F**: **f**eature build

For more information about the *<type>* building blocks, see **"Appendix - About the TRACE32 Software Version Numbers"** in PowerView User's Guide, page 123 (ide_user.pdf).

Examples:

```
PRINT VERSION.SOFTWARE.TYPE()          ;print the software version type
```

```
;check whether the TRACE32 software being used is not a release version
IF VERSION.SOFTWARE.TYPE()!="R"
   PRINT "wrong TRACE32 SW started"
```

VPU Functions

In This Section

See also

■ VPU ❑ VPU() ❑ VPUCR()

VPU() Value of VPU register

Syntax: **VPU(***<register_name>***.W0 .. .W3)**

Returns the content of the selected VPU register.

Parameter Type: String.

Return Value Type: Hex value.

Example:

```
PRINT VPU(VR2.W3)      ;returns word 3 of register VR2
```

VPUCR() Value of VRSAVE or VSCR register

Syntax: **VPUCR(***<register>***)**

<register>: **VRSAVE** | **VSCR**

Returns the content of the registers **VRSAVE** or **VSCR**.

Parameter Type: String.

Return Value Type: Hex value.

Example:

```
PRINT VPUCR(VRSAVE)    ;returns the content of register VRSAVE
```