SAS/SATA Protocol Suite Verification Script Engine Reference Manual

For SAS/SATA Protocol Suite software version 6.15

Generated: 8/27/2020 1:52 PM

Document Disclaimer

The information contained in this document has been carefully checked and is believed to be reliable. However, no responsibility can be assumed for inaccuracies that may not have been detected. Teledyne LeCroy reserves the right to revise the information presented in this document without notice or penalty.

Trademarks and Servicemarks

Teledyne LeCroy, CATC, SAS*Suite*, SATA*Suite*, SAS*Tracer*, SA*Tracer*, SAS*Trainer*, SA*Trainer* and SAS*Tracker* are trademarks of Teledyne LeCroy. Microsoft, Windows, Windows 2000, and Windows XP are registered trademarks of Microsoft Inc. All other trademarks are property of their respective companies.

Copyright

Table of Contents

Chapter 1: Introduction

This document contains a description of the Teledyne LeCroy Verification Script Engine (VSE), a new utility in the SATA*Suite*/SAS*Suite* software that allows users to perform custom analyses of SAS/SATA traffic, recorded using the new generation of SAS/SATA protocol analyzers.

VSE allows users to ask the SAS*Suite*/SATA*Suite* application to send some desired "events" (currently defined as Primitive/Frame/Transaction/Command) from a trace to a verification script written using the Teledyne LeCroy script language. This script then evaluates the sequence of events (timing, data, or both) in accordance with user-defined conditions and performs post-processing tasks; such as exporting key information to external text-based files or sending special Automation/COM notifications to user client applications.

VSE was designed to allow users to easily retrieve information about any field in a SAS/SATA Frame/Transaction/Command and to make complex timing calculations between different events in a pre-recorded trace. It also allows data filtering-in or filtering-out with dynamically changing filtering conditions, porting of information to a special output window, saving data to text files, and sending data to COM clients connected to a SAS*Suite*/SATA*Suite* application.

Chapter 2: Verification Script Structure

Writing a verification script is easy, as long as you follow a few rules and have some understanding of how the SAS*Suite*/SATA*Suite* application interacts with running scripts.

The main script file, which contains the text of the verification script, should have extension **.sasvs** and be in the subfolder **SASVFScripts** of the main SAS*Suite*/SATA*Suite* folder. Some other files might be included in the main script file using directive **%include**.
(See the **CATC Scripting Language Reference Manual** for details.)

The following schema presents a common structure of a verification script. This script is similar to the content of the script template **VSTemplate.sasv_** included with VSE:

```
#
#
# VS1.sasvs
#
# Verification script
#
# Brief Description:
# Performs specific verification
#
################################################################################
# Module info
################################################################################
# Filling of this block is necessary for proper verification script operation.
################################################################################
set ModuleType  = "Verification Script"; # Should be set for all verification scripts.
set OutputType  = "VS";                  # Should be set for all verification scripts
                                         # that output only Report string and Result.
set InputType   = "VS";

set DecoderDesc = "<Your Verification Script description>"; # Optional

################################################################################
# include main Verification Script Engine definitions
#
%include  "VSTools.inc"                 # Should be set for all verification scripts.

################################################################################
#                             Global Variables and Constants
################################################################################
# Define your verification script-specific global variables and constant
# in this section.
# (Optional)
#
const MY_GLOBAL_CONSTANT = 10;
set g_MyGlobalVariable   = 0;

################################################################################

################################################################################
# OnStartScript()
################################################################################
#
#        It is a main intialization routine for setting up all necessary
#        script parameters before running the script.
#
################################################################################
```

```
OnStartScript()
{
  ##############################################################################
  # Specify in the body of this function the initial values for global variables
  # and what kinds of trace events should be passed to the script.
  # By default, all packet level events from all channels
  # will be passed to the script.
  #
  # For details about how to specify what kind of events should be passed to the script
  # please see the topic 'sending functions'.
  #
  # OPTIONAL
  ##############################################################################

    g_MyGlobalVariable = 0;

    # Uncomment the line below if you want to disable output from
    # ReportText()-functions.
    #
    # DisableOutput();
}

##############################################################################
#   ProcessEvent()
##############################################################################
#
##############################################################################
# It is a main script function called by the application when the next waited event
# occured in the evaluated trace.
#
# !!! REQUIRED !!! - MUST BE IMPLEMENTED IN VERIFICATION SCRIPT
##############################################################################

ProcessEvent()
{
    # Write the body of this function depending upon your needs.
    # It might require branching on event type:
    # select {
    #   in. TraceEvent == … : …
    #   in. TraceEvent == … : …
    #    …
    # }
    return Complete();
}

##############################################################################
# OnFinishScript()
##############################################################################
#
##############################################################################
# It is a main script function called by the application when the script completes
# running. Specify in this function some resetting procedures for a successive run
# of this script.
#
# OPTIONAL
##############################################################################

OnFinishScript()
{
    return 0;
}

##############################################################################
# Additional script functions.
##############################################################################
# Write your own script-specific functions here.
##############################################################################

MyFunction( arg )
{
       if( arg == "My Arg" ) return 1;
       return 0;
}
```

Chapter 3: Interaction Between SAS*Suite*/SATA*Suite* and a Verification Script

When you run a script against a pre-recorded trace, the following sequence occurs:

Prior to sending information to the script's main processing function **ProcessEvent(),** VSE looks for the function **OnStartScript()** and calls it if it is found. In this function, setup actions are defined, such as specifying the kind of trace events that should be passed to the script and setting up initial values for script-specific global variables.

Next, VSE parses the recorded trace to verify that the current packet or other event meets specific criteria. If it does, VSE calls the script's main processing function **ProcessEvent(),** placing information about the current event in the script's input context variables.
(Please refer to the topic "Input context variables" later in this document for a full description of verification script input context variables.)

ProcessEvent() is the main verification routine for processing incoming trace events. This function must be present in all verification scripts. When the verification program consists of a few stages, the **ProcessEvent()** function processes the event sent to the script, verifies that information contained in the event is appropriate for the current stage, and decides if VSE should continue running the script or, if the whole result is clear on the current stage, tells VSE to complete execution of the script.

The completion of the test before the entire trace has been evaluated is usually done by setting the output context variable:

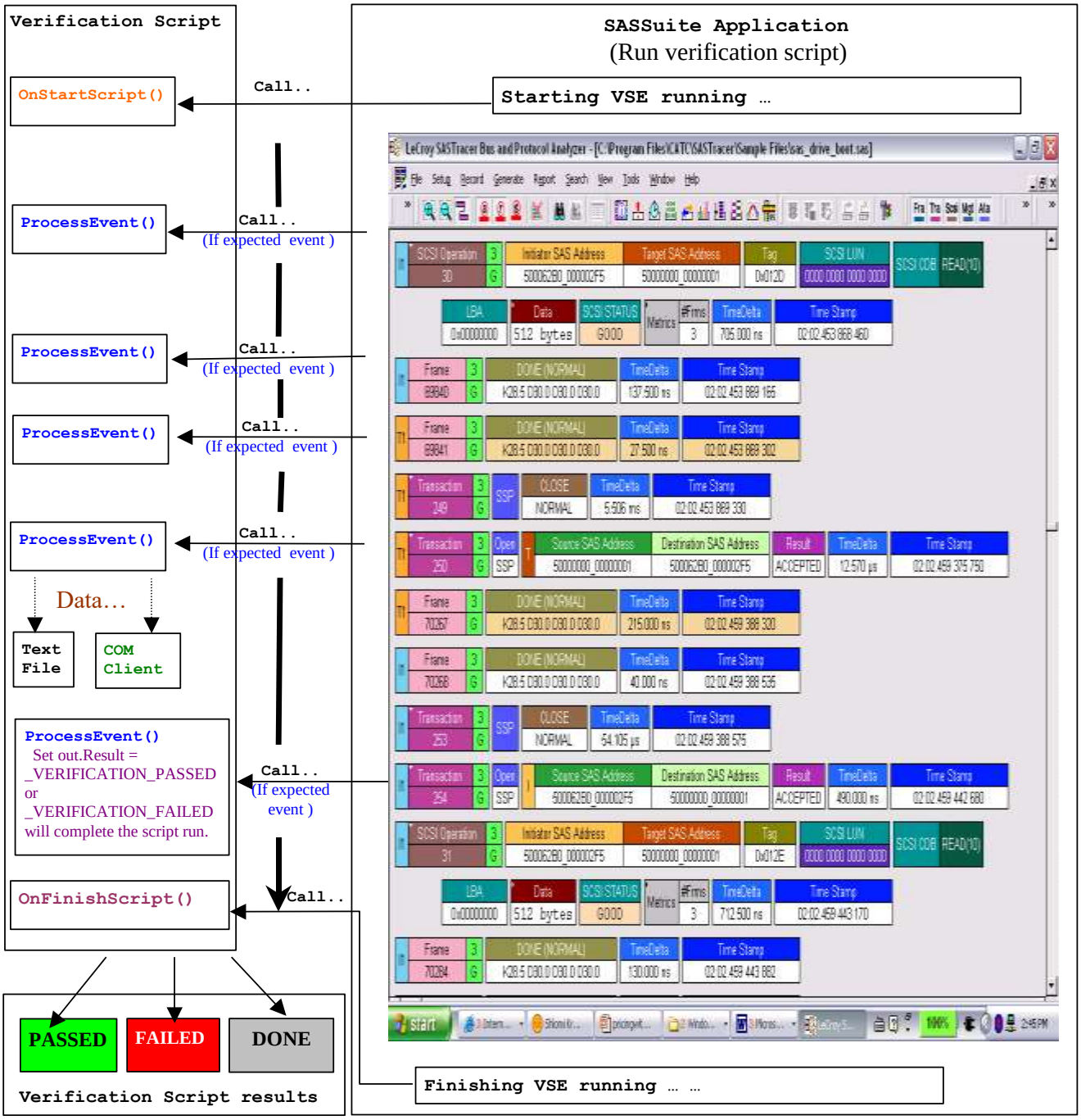
```
out.Result = _VERIFICATION_PASSED or _VERIFICATION_FAILED
```

(Please refer to the topic "Output context variables" later in this document for a full description of verification script output context variables.)

NOTE: Not only does a verification script evaluate recorded traces against some criteria, but it can also extract information of interest and post-process it later by some third-party applications. A set of script functions allows you to save extracted data in text files, or send it to other applications, via COM/Automation interfaces.

When the script has completed running, VSE looks for the function **OnFinishScript()** and calls it if found. In this function, some resetting procedures can be done.

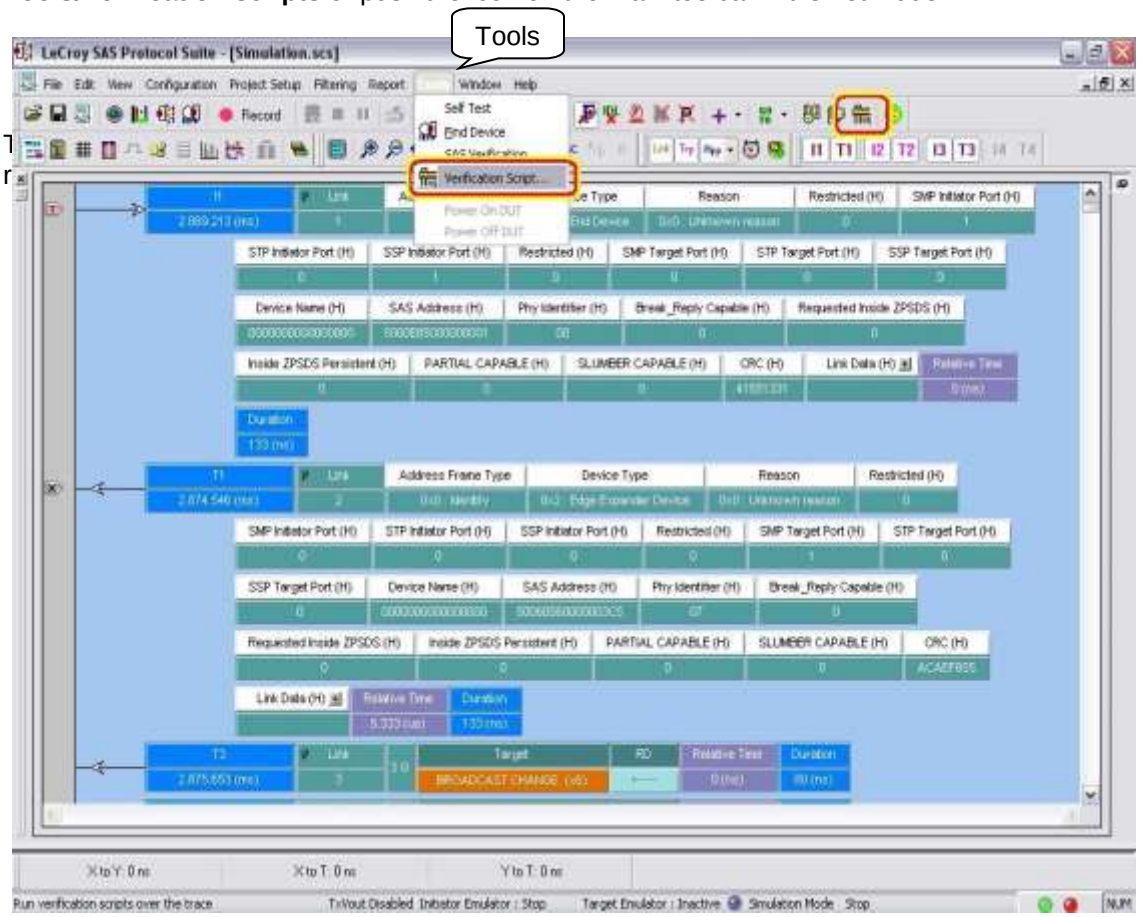
The following figure illustrates the interaction between the SAS*Suite*/SATA*Suite* application and a running verification script:



The Verification script result **DONE** occurs when the script has been configured to extract and display some information about the trace, but not to display PASSED/FAILED results. To configure a script so that it only displays information, place a call somewhere in your script to the function **ScriptForDisplayOnly(),** for example in **OnStartScript()**.

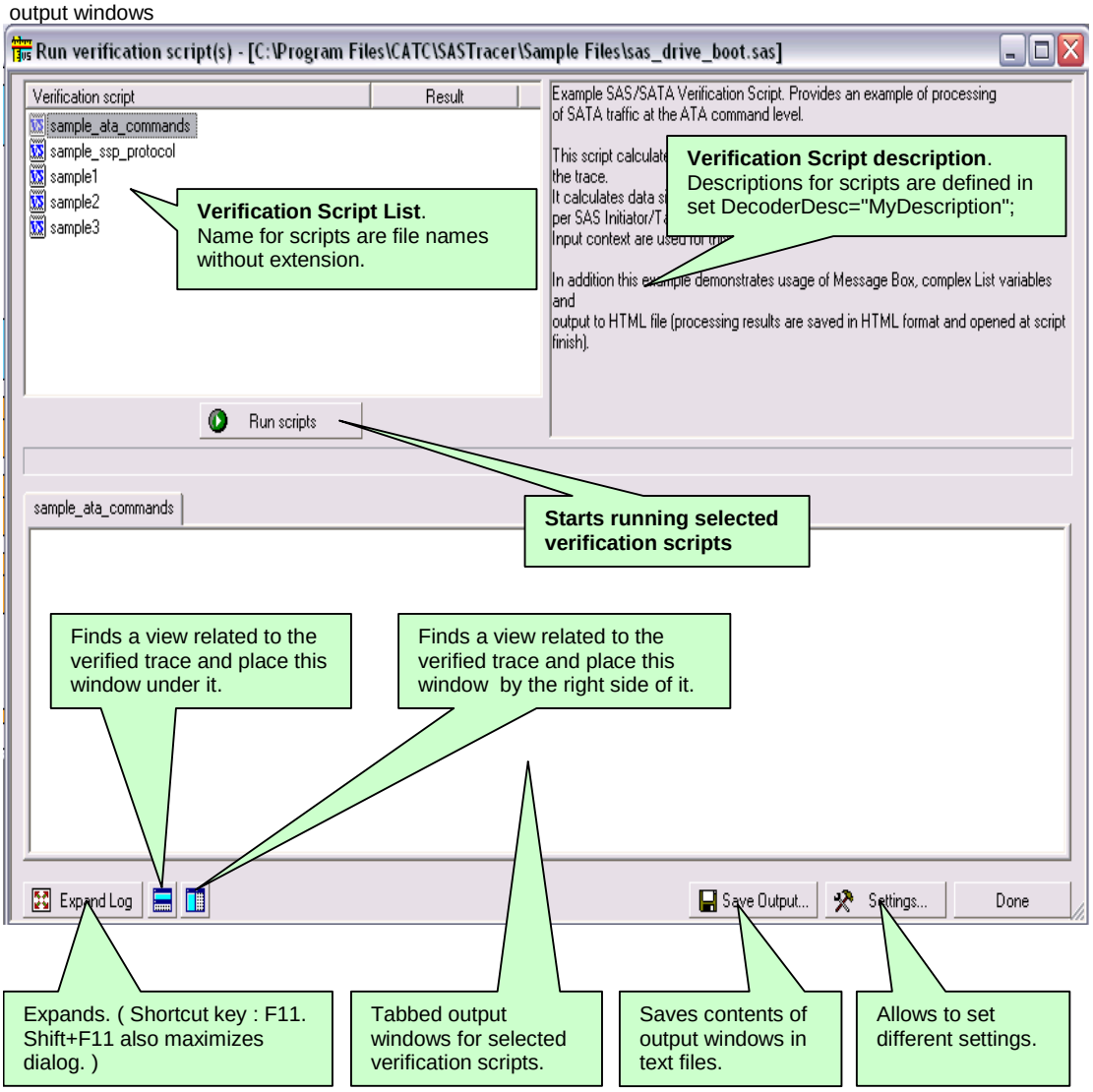
Chapter 4: Running Verification Scripts from SAS*Suite*/SATA*Suite*

To run a verification script over a trace, open the SAS*Suite*/SATA*Suite* main menu item **Tools/Verification scripts** or push the icon on the main toolbar if it is not hidden.



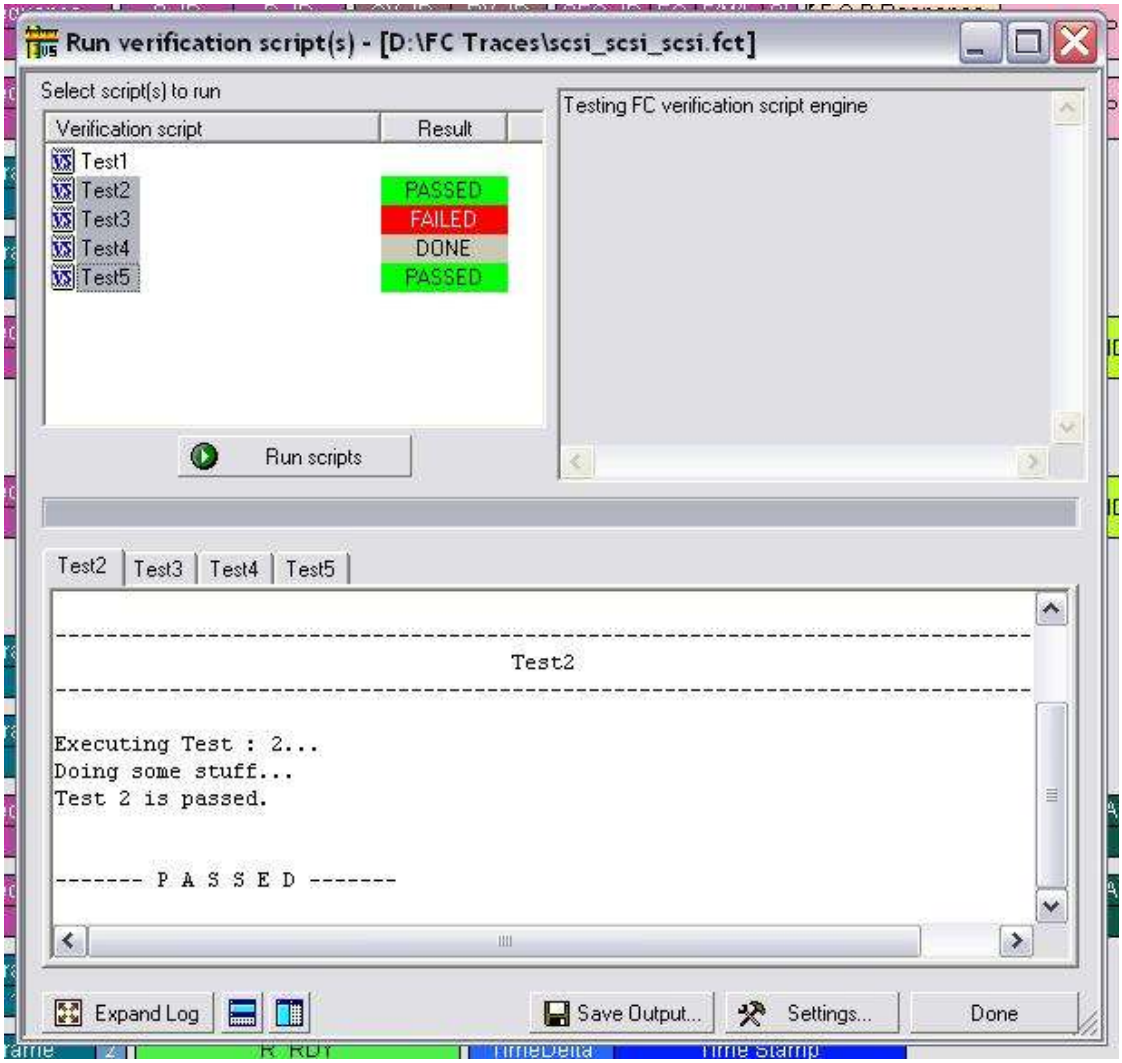
The special dialog will open and display a list of verifications scripts. You can select one script to run, or several scripts to run in parallel:

output windows

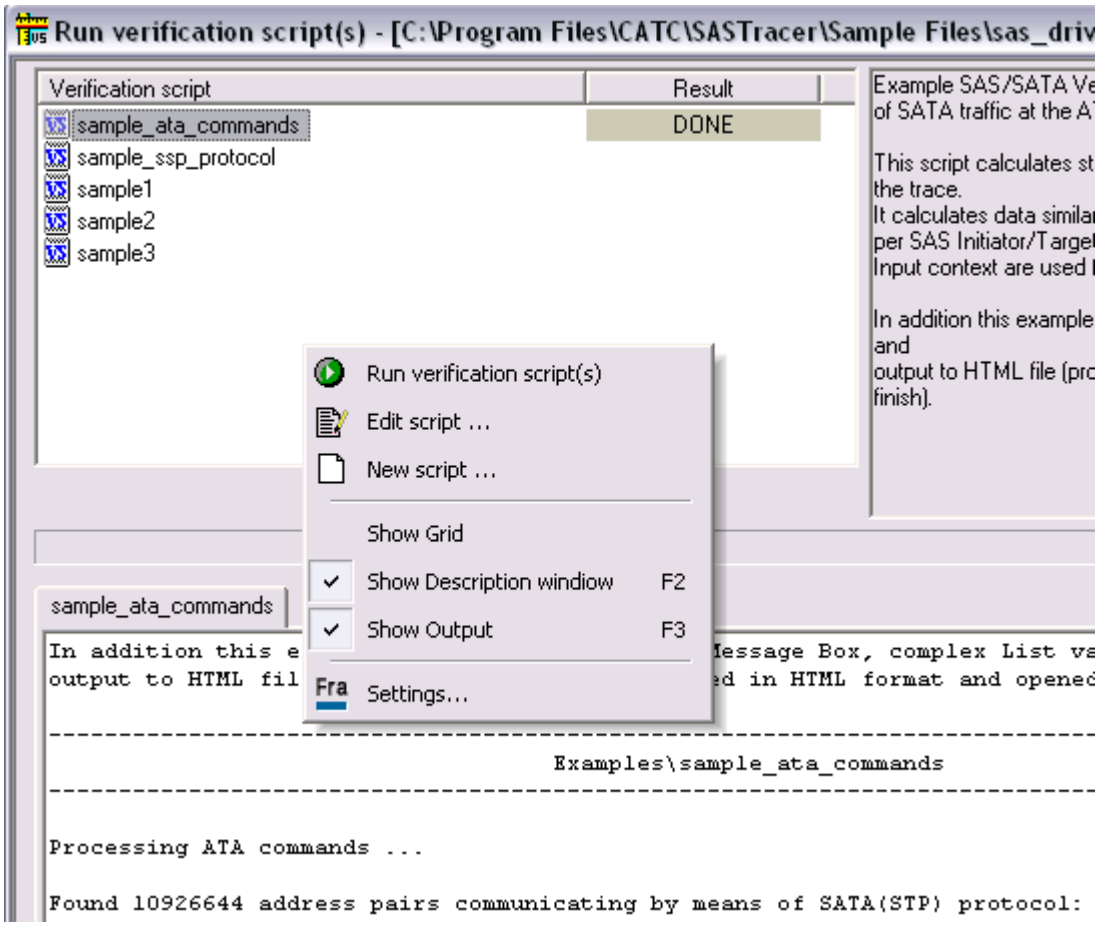


4.1 Running Verification Scripts

After you select the script(s) to run, push the button **Run scripts**. VSE starts running the selected verification script(s), shows script report information in the output windows, and presents results of verifications in the script list:



Right-click in the script list to open a pop-up menu with options for performing additional operations on the selected scripts:



Run verification script(s) Starts running selected script(s).

Edit script Allows editing of the selected script(s) using the editor specified in the **Editor settings.**

New script Creates a new script file using the template specified in the **Editor settings.**

Show Grid Shows/hides a grid in the verification script list.

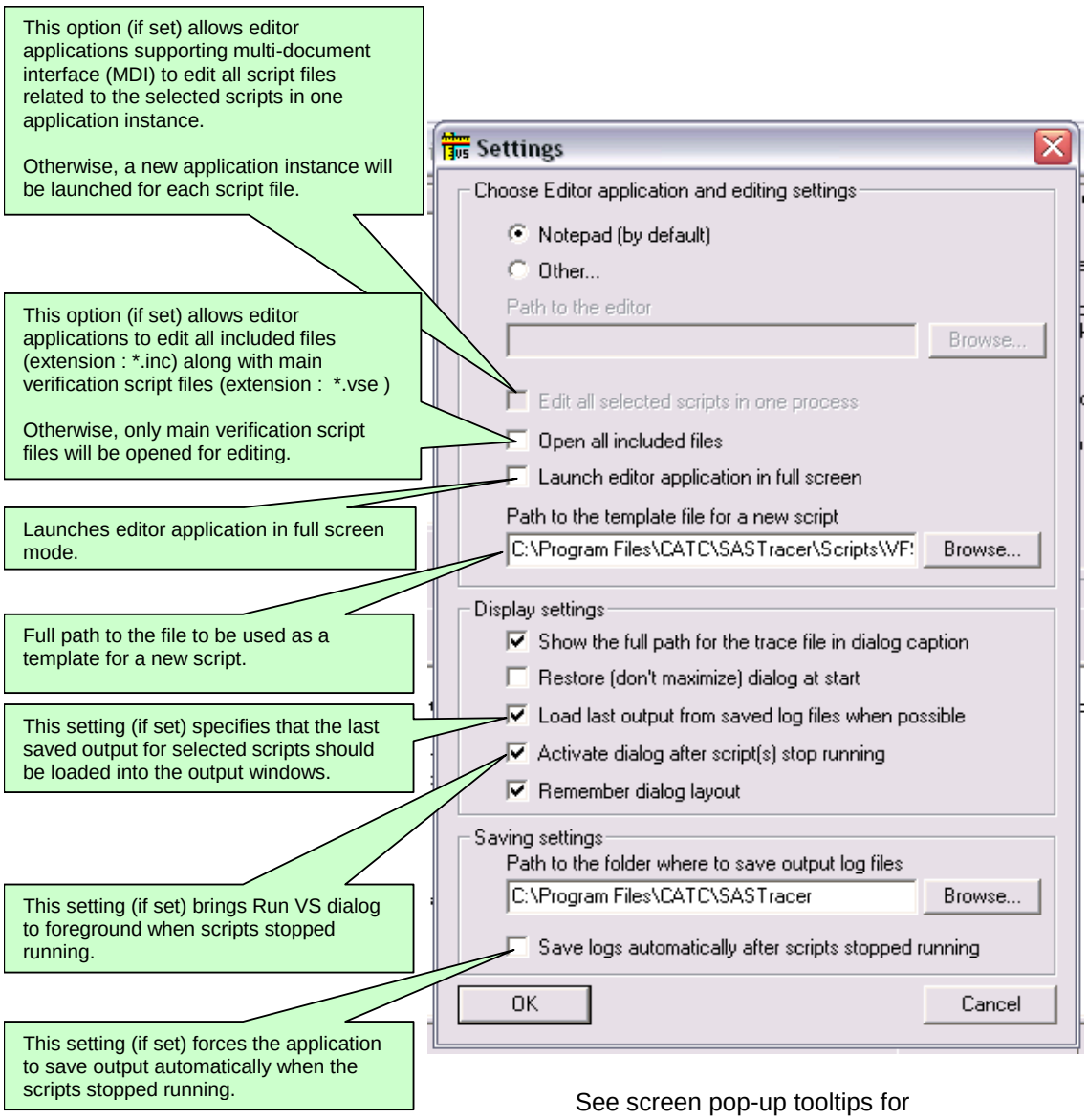
Show Description window Shows/hides the script description window**.** **(Shortcut key : F2)**

Show Output Shows/hides the script output windows. **(Shortcut key : F3)**

Settings Opens a special **Setting** dialog for you to specify different settings for VSE.

4.2 VSE GUI Settings

After choosing **Settings**, the following dialog appears:



See screen pop-up tooltips for explanation of other settings…

Chapter 5: Verification Script Engine Input Context Members

All verification scripts have input contexts –special structures whose members are filled by the application and can be used inside of the scripts. (For more details about input contexts, refer to the *CATC Script Language(CSL) Manual.*. The verification script input contexts have two sets of members:

- Trace event-independent set of members
- Trace event-dependent set of members

5.1 Trace Event-independent Set of Members

This set of members is defined and can be used for any event passed to script:

in.Level — Logical level of the trace event: 0 = Link, 1 = Transport, 2 = Application.

in.Index — Index of the event in the trace file: frame number for frames, sequence number for sequences.

in.Time — Time of the event: type = list, having format `2 sec 125 ns -> [2 , 125]`. (See VSE time object for details.)

in.Duration — Duration: time that packet contains.

in.Channel — Channel where the event occured: may be **_CHANNEL_1 (1)** or **_CHANNEL_2 (2),** indicating which direction of the SAS/SATA link the event occurred)

in.TraceEvent — Type of trace event: application predefined constants are used. See the list of possible events below.

in.Notification — Type of notification: application predefined constants are used. Currently, no notifications are defined.

in.LinkSpeed — Speed of the trace event: may be **_LINK_SPEED_1_5 (1.5 Gbps)** or **_LINK_SPEED_3_0 (3Gbps)** or **_LINK_SPEED_6_0 (6Gbps)**
*Gbps – Giga Bits per second

in.ProtocolErrorMask — A bitmask pattern contains detected errors on the specified event: Each bit on this mask demonstrates a possible error. To know about error bit position look at See SendProtocolError for details.

In.DWORDS — All DWORDs of current packet: type = list of list, having format
[DWORD, Count, HasError]
HasError: 0 means it has no error, 1 means it has error
Example
[[“SOF”, 1, 0], [“50E80050”, 1, 1], …]
[[“D10.2”, 258, 0]]

In.COMWAKE_PHYCAP_BIT — *Phy-Capability-bit position of the COMWAKE*

Example:

```
if(event_type == _FRM_OOB_SIGNAL)
{
if(in.OOBType == SAS_OOB_COMWAKE)
    {
        PhyCapBitPos = in.COMWAKE_PHYCAP_BIT;
    }
}
```

In.I_PHYCAP — *32-bit SNW3 Phy-Capabilities Value of Initiator*

In.T_PHYCAP — *32-bit SNW3 Phy-Capabilities Value of Target*

Example:

```
if(in.Level == _OOB_SEQ)
{
    SNW3_Initiator_Phy_Cap  = in.I_PHY_CAPABILITIES;
    SNW3_Target_Phy_Cap     = in.T_PHY_CAPABILITIES;
}
```

In.I_PHYCAP_REQUESTEDLOGICALLINKRATE — *Value of the field REQUESTED LOGICAL LINK RATE of Initiator Phy-Capabilities*

In.I_PHYCAP_TXSSCTYPE — *Value of the field TXSSCTYPE of Initiator Phy-Capabilities*

In.I_PHYCAP_START — *Value of the field START of Initiator Phy-Capabilities*

In.I_PHYCAP_G4WITHSSC — *Value of the field G4 WITH SSC of Initiator Phy-Capabilities*

In.I_PHYCAP_G4WITHOUTSSC — *Value of the field G4 WITHOUTSSC of Initiator Phy-Capabilities*

In.I_PHYCAP_G3WITHSSC — *Value of the field G3 WITH SSC of Initiator Phy-Capabilities*

In.I_PHYCAP_G3WITHOUTSSC — *Value of the field G3 WITHOUTSSC of Initiator Phy-Capabilities*

In.I_PHYCAP_G2WITHSSC — *Value of the field G2 WITH SSC of Initiator Phy-Capabilities*

In.I_PHYCAP_G2WITHOUTSSC — *Value of the field G2 WITHOUTSSC of Initiator Phy-Capabilities*

In.I_PHYCAP_G1WITHSSC — *Value of the field G1 WITH SSC of Initiator Phy-Capabilities*

In.I_PHYCAP_G1WITHOUTSSC — *Value of the field G1 WITHOUTSSC of Initiator Phy-Capabilities*

In.I_PHYCAP_PARITY — *Value of the field PARITY of Initiator Phy-Capabilities*

In.T_PHYCAP_REQUESTEDLOGICALLINKRATE — *Value of the field REQUESTED LOGICAL LINK RATE of Target Phy-Capabilities*

In.T_PHYCAP_TXSSCTYPE — *Value of the field TXSSCTYPE of Target Phy-Capabilities*

In.T_PHYCAP_START — *Value of the field START of Target Phy-Capabilities*

In.T_PHYCAP_G4WITHSSC — *Value of the field G4 WITH SSC of Target Phy-Capabilities*

In.T_PHYCAP_G4WITHOUTSSC	*Value of the field G4 WITHOUT SSC of Target Phy-Capabilities*
In.T_PHYCAP_G3WITHSSC	*Value of the field G3 WITH SSC of Target Phy-Capabilities*
In.T_PHYCAP_G3WITHOUTSSC	*Value of the field G3 WITHOUT SSC of Target Phy-Capabilities*
In.T_PHYCAP_G2WITHSSC	*Value of the field G2 WITH SSC of Target Phy-Capabilities*
In.T_PHYCAP_G2WITHOUTSSC	*Value of the field G2 WITHOUT SSC of Target Phy-Capabilities*
In.T_PHYCAP_G1WITHSSC	*Value of the field G1 WITH SSC of Target Phy-Capabilities*
In.T_PHYCAP_G1WITHOUTSSC	*Value of the field G1 WITHOUT SSC of Target Phy-Capabilities*
In.T_PHYCAP_PARITY	*Value of the field PARITY of Target Phy-Capabilities*

Example:

```
if(in.Level == _OOB_SEQ)
{
ReportText( FormatEx( "\t Start => %d",  in.I_PhyCap_Start));
ReportText( FormatEx( "\t TX SSC Type => %d", in.I_PhyCap_TXSSCType));
}
```

Note: For SATA/STP frame, if trace has been captured with M6-4, M6-2 or M6-1, “Host” and “Device” are coming sequentially one after other: [[Host], [Device], [Host], [Device], ….]

5.2 Trace Event-dependent Set of Members

This set of members is defined and can be used only for specific events or after calling some functions and filling out some of the variables:
Note: As a general rule for all fields' values, if the field length is less than or equal to 4 bytes (32 bits), it is returned as an integer value, but if the field length is larger than 4 bytes, it is returned as a raw data type.

For example:
- In.CRC: 4-Byte value is returned as an Integer.
- in.Lnk_SourceSASAddress: 8-Byte value is returned as Raw Data.

For more information about Raw Data values, please see section 5.3 "Raw Data field Value" section.

5.2.1 Primitives

in.Primitive	4-byte value of the Primitive
in.PrimCode	Primitive Code (refer to Appendix A)
in.PrimCount	Primitive count

5.2.2 OOB

in.OOBType OOB Types are defined as:
SAS_OOB_COMWAKE
SAS_OOB_COMRESET
SAS_OOB_COMSAS
SAS_OOB_KEEP_ALIVE_ACTIVITY

in.OOBBurstIdleList List of pairs (<burst_or_idle>, <count>),
where <burst_or_idle> is 0 for idle and 1 for burst. For example:

```
oob_burst_idle_list = in.OOBBurstIdleList;

for ( i = 0; i < sizeof( oob_burst_idle_list ); i++ )
{
  oob_burst_idle = oob_burst_idle_list[ i ];
  report += repIndent;
  select
{
oob_burst_idle[ 0 ] == SAS_OOB_ELEMENTTYPE_IDLE  :
report += Format( "OOBIdle  :  %ld oobi\n", i, oob_burst_idle[ 1 ] );
oob_burst_idle[ 0 ] == SAS_OOB_ELEMENTTYPE_BURST :
report += Format( "OOBBurst :  %ld oobi\n", i, oob_burst_idle[ 1 ] );
};
}
```

5.2.3 SAS/SATA Generic Frame members

in.Payload	Bit source of the frame/sequence payload. You can extract any necessary information using the **GetNBits(), NextNBits(),** or **PeekNBits()** functions. Please refer to the *CSL Manual* for details about these functions.
in.PayloadLength	Length in bytes of the retrieved payload
in.SOF, ***in.StartOfFrame***	4-byte value of Start of frame primitive
in.EOF, ***in.EndOfFrame***	4-byte value of End of frame primitive
in.CRC	CRC value as transmitted

5.2.4 How to Access Frame Members

IMPORTANT: Non-Compatibility with SASTracer VSE
The old SASTracer VSE used host-to-device register fields as ATA command fields.
The current VSE has changed that. Now, to get information of ATA of any event, use **IN.App.Command**, where “Command” is a field of the ATA command, not a host-to-device register (Transport layer).
The parameters of the “IN” context have changed to the SAS/SATA packet's field.
For example, to show the “Interrupt bit” on a device-to-host register, you now use "IN.I", not the old "IN.Interrupt".
In general, you must use field names based on the Sierra software, not field names based on the SAS*Tracer* software.

For all function-specific fields of any frame/command type, refer to the actual field name in the trace file and access the same as follows:

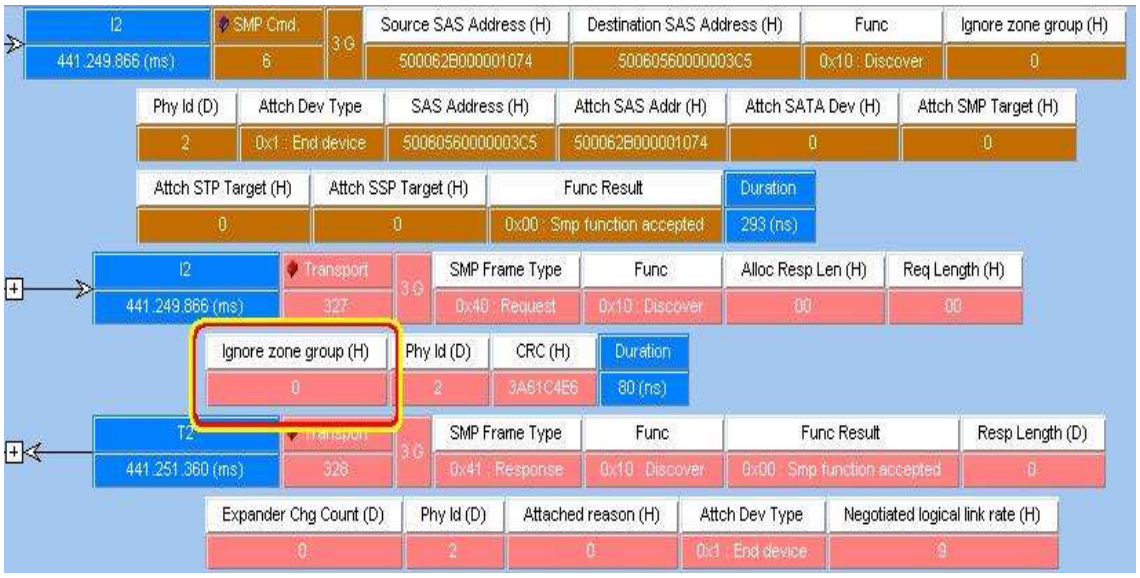
in.<Level>_<FieldName without spaces>

For <Level>, use:
- _**App** for Application Layer Commands
- _**Trp** for all Transport Level Events
- _**Lnk** for all Link Level Events

For <Field Name>, include field name of the frame WITHOUT SPACES.
For example, for "Write Data Len", use <FieldName> = 'WriteDataLen'.

Example

Following the above, to access a SMP Transport -> 'Ignore Zone Group' field, use **In.Trp_IgnoreZoneGroup**



SAS Open Address Frame members

Note: All members return the content of the matching field name ("in.xyz" returns the value of field "xyz") based on the SAS specification.

in.Lnk_AddressFrameType
in.Lnk_Protocol
in.Lnk_InitiatorPort
in.Lnk_ConnectionRate
in.Lnk_Features
in.Lnk_InitiatorConnectionTag
in.Lnk_DestinationSASAddress
in.Lnk_SourceSASAddress
in.*Lnk_SourceZoneGroup*
in.Lnk_PathwayBlockedCount
in.Lnk_ArbitrationWaitTime
in.Lnk_MoreCompatibleFeatures

5.2.5 SAS Identify Address Frame members

Note: All members return the content of the matching field name ("in.xyz" returns the value of field "xyz") based on the SAS specification.

in.Lnk_AddressFrameType
in.Lnk_DeviceType
in.Lnk_Reason
in.Lnk_Restricted
in.Lnk_SMPInitiatorPort
in.Lnk_STPInitiatorPort
in.Lnk_SSPInitiatorPort
in.Lnk_SMPTargetPort
in.Lnk_STPTargetPort
in.Lnk_SSPTargetPort
in.Lnk_DeviceName
in.Lnk_SASAddress

in.Lnk_PhyIdentifier
in.Lnk_Break_ReplyCapable
in.Lnk_RequestedInsideZPSDS
in.Lnk_InsideZPSDSPersistent
in.Lnk_PARTIALCAPABLE
in.Lnk_SLUMBERCAPABLE

5.2.6 SAS12 Training Sequence members

in.Lnk_ControlField
in.Lnk_CoefficientSettings
in.Lnk_PatternType
in.Lnk_Coefficient1Request
in.Lnk_Coefficient2Request
in.Lnk_Coefficient3Request
in.Lnk_StatusField
in.Lnk_BALANCE
in.Lnk_TXInit
in.Lnk_TRAINCOMP
in.Lnk_Coefficient1Status
in.Lnk_Coefficient2Status
in.Lnk_Coefficient3Status

5.2.7 SAS SSP Frame members

Note: All members return the content of the matching field name ("in.xyz" returns the value of field "xyz") based on the SAS specification.

in.Trp_SSPFrameType
in.Trp_HashedDestSASAddr
in.Trp_HashedSrcSASAddr
in.Trp_ChangingDataPointer
in.Trp_ReTransmit
in.Trp_RetryDataFrames
in.Trp_TLRCONTROL
in.Trp_NumOfFillBytes
in.Trp_Tag
in.Trp_TargetPortTransferTag
in.Trp_DataOffset

5.2.7.1 XFER_RDY SSP Frame

in.Trp_SSPFrameType is SSP_FRAME_TYPE_ XFER_RDY (0x05)

in.Trp_ReqOffset
in.Trp_WriteDataLen

5.2.7.2 COMMAND SSP Frame

in.Trp_SSPFrameType is SSP_FRAME_TYPE_COMMAND (0x06)

in.Trp_LogicalUnitNumber
in.Trp_TaskAttribute
in.Trp_TaskPriority
in.Trp_EnableFirstBurst
in.Trp_AddiCDBLen
in.Trp_CDB

5.2.7.3 RESPONSE SSP Frame

in.Trp_SSPFrameType is SSP_FRAME_TYPE_RESPONSE (0x07)

in.Trp_STATUSQUALIFIER
in.Trp_DataPresent
in.Trp_Status
in.Trp_SenseDataLen
in.Trp_RespDataLen

5.2.7.4 TASK SSP Frame

in.Trp_SSPFrameType is SSP_FRAME_TYPE_TASK (0x016)

in.Trp_LogicalUnitNumber
in.Trp_TaskMngFunc
in.Trp_TagofTasktobeMng

5.2.7.5 Data SSP Frame

in.Trp_SSPFrameType is SSP_FRAME_TYPE_DATA (0x01)

in.Trp_Data

5.2.8 SAS SMP Frame members

Note: All members return the content of the matching field name ("in.xyz" returns the value of field "xyz") based on the SAS specification.

5.2.8.1 Common SMP Frame members

in.Trp_SMPFrameType
in.Trp_Func

5.2.8.2 SMP Request Frame members

in.Trp_AllocRespLen
in.Trp_ReqLength

5.2.8.2.1 Report General Frame members

in.Trp_ReqLength

5.2.8.2.2 Report Manufacturer Info Frame members

in.Trp_ReqLength

5.2.8.2.3 Discover Frame members

in.Trp_Ignorezonegroup
in.Trp_PhyId

Note: Follow the same procedure explained in Section 5.2.4 for the remaining Command/Function/OpCode specific frame members:

5.2.8.2.4 Read GPIO register

5.2.8.2.5 Report self-configuration status

5.2.8.2.6 Report zone permission table members

5.2.8.2.7 Report zone manager password members

5.2.8.2.8 Report broadcast members

5.2.8.2.9 Report phy error log members

5.2.8.2.10 Report phy SATA members

5.2.8.2.11 Report route information members

5.2.8.2.12 Report phy event members

5.2.8.2.13 Discover list members

5.2.8.2.14 Report phy event list members

5.2.8.2.15 Report expander route table list members

5.2.8.2.16 Configure general members

5.2.8.2.17 Enable disable zoning members

5.2.8.2.18 Write GPIO register members

5.2.8.2.19 Zoned broadcast members

5.2.8.2.20 Zone lock members

5.2.8.2.21 Zone activate members

5.2.8.2.22 Zone unlock members

5.2.8.2.23 Configure zone manager password members

5.2.8.2.24 Configure zone phy information members

5.2.8.2.25 Configure zone permission table members

5.2.8.2.26 Configure route information members

5.2.8.2.27 Phy control members

5.2.8.2.28 PHY test function members

5.2.8.2.29 Configure phy event members

5.2.8.2.30 Configure zone permission members

5.2.8.3 SMP Response Frames members

in.Trp_FuncResult
in.Trp_RespLength

5.2.8.3.1 Report General Frame members

in.Trp_ExpanderChgCount
in.Trp_ExpanderRouteIdx
in.Trp_LongResponse
in.Trp_NumofPhys
in.Trp_Externallyconfigurableroutetable
in.Trp_Config
in.Trp_Configuresothers
in.Trp_OpenRejectRetrySupported
in.Trp_ContinueAWT
in.Trp_SelfConfiguring
in.Trp_ZoneConfiguring
in.Trp_Tabletotablesupported
in.Trp_EnclosureLogicalIdentifier
in.Trp_businactivitytimelimit
in.Trp_maximumconnecttimelimit
in.Trp_SMPI_Tnexuslosstime
in.Trp_Zoningenabled
in.Trp_Zoningsupported
in.Trp_Physicalpresenceasserted
in.Trp_Physicalpresencesupported
in.Trp_ZoneLocked
in.Trp_Numberofzonegroups
in.Trp_Savingzoningenabledsupported
in.Trp_Savingzonepermissiontablesupported
in.Trp_Savingzonephyinformationsupported
.
.
.
And so on

Note: Follow the same procedure explained in Section 5.2.4 for the remaining Command/Function/OpCode specific frame members:

5.2.8.3.1.1.1.1 Report Manufacturer Info Frame members

5.2.8.3.1.1.1.2 Discover Frame members

5.2.8.3.1.1.1.3 Read GPIO register

5.2.8.3.1.1.1.4 Report self-configuration status

5.2.8.3.1.1.1.5 Report zone permission table members

5.2.8.3.1.1.1.6 Report zone manager password members

5.2.8.3.1.1.1.7 Report broadcast members

5.2.8.3.1.1.1.8 Report phy error log members

5.2.8.3.1.1.1.9 Report phy SATA members

5.2.8.3.1.1.1.10 Report route information members

5.2.8.3.1.1.1.11 Report phy event members

5.2.8.3.1.1.1.12 Discover list members

5.2.8.3.1.1.1.13 Report phy event list members

5.2.8.3.1.1.1.14 Report expander route table list members

5.2.8.3.1.1.1.15 Configure general members

5.2.8.3.1.1.1.16 Enable disable zoning members

5.2.8.3.1.1.1.17 Write GPIO register members

5.2.8.3.1.1.1.18 Zoned broadcast members

5.2.8.3.1.1.1.19 Zone lock members

5.2.8.3.1.1.1.20 Zone activate members

5.2.8.3.1.1.1.21 Zone unlock members

5.2.8.3.1.1.1.22 Configure zone manager password members

5.2.8.3.1.1.1.23 Configure zone phy information members

5.2.8.3.1.1.1.24 Configure zone permission table members

5.2.8.3.1.1.1.25 Configure route information members

5.2.8.3.1.1.1.26 Phy control members

5.2.8.3.1.1.1.27 PHY test function members

5.2.8.3.1.1.1.28 Configure phy event members

5.2.8.3.1.1.1.29 Configure zone permission members

5.2.9 SATA/STP Frame Members

Note: All members return the content of the matching field name ("in.xyz" returns the value of field "xyz") based on the SATA specification.

5.2.9.1 Common SATA/STP Frame Members

in.Trp_STPFrameType
in.Trp_PMPort

5.2.9.2 Register Host To Device Frame Members

in.Trp_C
in.Trp_Command
in.Trp_Features
in.Trp_SectorNumber
in.Trp_CylLow
in.Trp_CylHigh
in.Trp_DevHead
in.Trp_SectorNumExp
in.Trp_CylLowExp
in.Trp_CylHighExp
in.Trp_FeaturesExp
in.Trp_SectorCount
in.Trp_SectorCountExp
in.Trp_Control

5.2.9.3 DMA Setup Frame Members

in.Trp_D
in.Trp_I
in.Trp_A
in.Trp_DMABufferidLow
in.Trp_DMABufferidHi
in.Trp_DMABufferOffset
in.Trp_DMABufferTransferCount

Note: Follow the same procedure explained in Section 5.2.4 for the remaining Command/Function/OpCode specific frame members:

5.2.9.4 Register Device To Host Frame Members

5.2.9.5 Set Device Bits Frame Members

5.2.9.6 DMA Activate Frame Members

5.2.9.7 Bist Activate Frame Members

5.2.9.8 PIO Setup Frame Members

5.2.9.9 DATA Frame Members

5.2.10 SCSI Command Frame Members

Note: All members return the content of the matching field name ("in.xyz" returns the value of field "xyz") based on the SAS specification.

in.App_SourceAddress
in.App_DestinationAddress
in.App_OperationCode
in.App_Control
in.App_TaskAttribute
in.App_Tag
in.App_Status
in.App_LUN

5.2.10.1 Read (10) Frame Members

in.App_RelAdr
in.App_FUA
in.App_DPO
in.App_LogicalBlockAddress
in.App_TransferLength
in.App_PayloadData

5.2.10.2 Mode Sense (6) Frame Members

in.App_DBD
in.App_ModePageCode
in.App_PC
in.App_SubpageCode
in.App_AllocationLength
in.App_ModeParameterList
in.App_SenseKey
in.App_ASCASCQ
in.App_SenseData

Note: Follow the same procedure explained in Section 5.2.4 for the remaining Command/Function/OpCode specific frame members:

5.2.10.3 Write (10) Frame Members

5.2.10.4 Inquiry Frame Members

5.2.10.5 Report LUNS Frame Members

5.2.10.6 Read Capacity (10) Frame Members

5.2.10.7 Mode Select (6) Frame Members

5.2.10.8 Set Read Ahead Frame Members

5.2.10.9 Read Attribute Frame Members

And so on.

5.2.11 ATA Commands Frame Members

Note: All members return the content of the matching field name ("in.xyz" returns the value of field "xyz") based on the SATA specification.

in.App_SourceSASAddress
in.App_DestinationSASAddress
in.App_Command
in.App_Input
in.App_PMPort
in.App_Protocol
in.App_Status

5.2.11.1 Read DMA Ext Frame Members

in.App_SecCount
in.App_LBA
in.App_DEV
in.App_LBAMode

5.2.11.2 Write FPDMA Queued Frame Members

in.App_SecCount
in.App_Tag
in.App_PRIO
in.App_LBA
in.App_ICC
in.App_DEV
in.App_LBAMode
in.App_FUA

Note: Follow the same procedure explained in Section 5.2.4 for the remaining Command/Function/OpCode specific frame members:

5.2.11.3 Write DMA Ext Frame Members

5.2.11.4 Identify Device Frame Members

5.2.11.5 Read Buffer Frame Members

5.2.11.6 Read Multiple Frame Members

5.2.11.7 Write Buffer Frame Members

5.2.11.8 Write Multiple Frame Members

5.2.11.9 Write Log Ext Frame Members

5.2.11.10 Write Port Multiplier Frame Members

5.2.11.11 Read Log DMA Ext

5.2.11.12 Write DMA FUA Ext

And so on.

5.2.12 Metrics Fields for SCSI and ATA Commands (ONLY)

Note: All members return the content of the matching field name ("in.xyz" returns the value of field "xyz") based on the SAS and SATA specification.

in.TrpNo Number of Transports: total number of transports that compose this exchange

Type: *integer*

in.RespTime Response Time: time it took to transmit this exchange on the link(s)

Type: *time (for more information, Please see chapter 10 to 14)*

in.PayloadLength Payload: number of Payload Bytes this operation transferred

Type: *integer*

in.Latency Latency: time measured from the transmission of SCSI/ ATA Command to the first data transmitted to this SCSI/ ATA IO Operation

Type: *time (for more information, Please see chapter 10 to 14)*

in.DataStatTime Data to Status Time: time between end of data transmission for this SCSI/ATA Command and the status frame

Type: *time (for more information, Please see chapter 10 to 14)*

in.Throughput Data Throughput: Payload divided by Response time in KB/s.

Type: *integer*
Format: *KB/s (multiple by 1024 to get MB/s)*

5.3 Raw Data field Value

As a general rule for all fields' values, if the field length is less than or equal to 4 bytes (32 bits), it is returned as an integer value, but if the field length is larger than 4 bytes, it is returned as a raw data type. If a field has variable size such as LBA, sizeof() must be used to verified the field length.

For example, LBA has variable length that depends on its command and it must be checked like this:

```
IsGreaterLBA(LeftLBA, RightLBA)
{
        # It is a raw data.
        If(sizeof(LeftLBA) >4)
        {
            # This method has been implemented as an example of Raw Data in the following.
            Return IsGreaterRawData (LeftLBA, RightLBA);
        }

        # It is an integer value
        return (LeftLBA > RightLBA)
}
```

The Raw Data is actually a "Raw Bytes" value format which has been defined in the CSL_RefManual document.

It is a literal notation, which is supported using single quotes:

'00112233445566778899AABBCCDDEEFF'

This represents an array of 16 bytes with values starting at 0x00 and ranging up to 0xFF. These values can only be hexadecimal digits. Each digit represents a nibble (four bits), and if there is not an even number of nibbles specified, an implicit zero is added to the first byte.

For example:
'FFF' is interpreted as '0FFF'
Raw data field values are stored as Little Endian.

For example, if the value of a field is:
SASAddress = 0x1122334455667788

It is stored as
‘8877665544332211’

And each byte value is accessible by “[index]” such as following:
SASAddress[0] = 0x88 (Least Significant Byte)
SASAddress[1] = 0x77
SASAddress[7] = 0x11 (Most Significant Byte)

Here is an example of how to use raw data, which returns a 1 if the first passed RawData is greater than the second one, otherwise it returns 0:

```
IsGreaterRawData (LeftRaw, RightRaw)
{
    # if both Raw Data are equal
    if(LeftRaw == RightRaw)
            return 0;

    #Get Size of raw data in bytes
    LeftRawSize     = sizeof(LeftRaw);
    RightRawSize    = sizeof(RightRaw);

    # If size of the Left Raw Data is greater than Right Raw Data, the value is greater too.
    if(LeftRawSize > RightRawSize)
            return 1;
```

```
	# If the size of the Left Raw Data is less than the Right Raw Data, the value is less too.
	if(LeftRawSize < RightRawSize)
		return 0;

	# If both size are equal
	if(LeftRawSize == RightRawSize)
	{
		# Because it is saved in Little Endian format, it must start comparing from the last index.
		for(index = RightRawSize - 1; index >= 0; index--)
		{
			if(LeftRaw[index] > RightRaw[index])
				return 1;

			if(LeftRaw[index] < RightRaw[index])
				return 0;
		}
	}

	return 0;
}
```

To format raw to string, please see 22.6 FormatRawBytes(RawBytes)

Chapter 6: Verification Script Engine Output Context Members

All verification scripts have output contexts –special structures whose members are filled by the script and can be used inside of the application. (For more details about output contexts, refer to the *CATC Script Language(CSL) Manual.*) The verification script output contexts have only one member:

out.Result Result of the whole verification program defined in the verification script

This member is supposed to have three values:
_VERIFICATION_PROGRESS (set by default when script starts running)
_VERIFICATION_PASSED
_VERIFICATION_FAILED

The last two values should be set if you decide that the recorded trace does (or does not) satisfy the imposed verification conditions. In both cases, the verification script will stop running.

If you do not specify any of those values, the result of script execution is set as **_VERIFICATION_FAILED** at exit.

Note: If you do not care about the results of the script that's running, call the function **ScriptForDisplayOnly()** once before stopping the script. Then the result will be **DONE**.

Chapter 7: Verification Script Engine Events

VSE defines a large group of trace "events", on packet, link, and split transaction levels, that can be passed to a verification script for evaluation or for retrieving and displaying some contained information. The information about the type of event can be seen in **in.TraceEvent**.

Refer to the topic Sending functions for details about how to specify transaction levels and which events should be sent to verification scripts.

7.1 Frame Level Events

The table below describes the current list of Frame Level events (transaction level 0) and value of **in.TraceEvent**:

Types of Frames	**in.TraceEvent**
Primitive	_FRM_PRIMITIVE
OOB Signal Frame	_FRM_OOB_SIGNAL
Open Address Frame	_FRM_AF_OPEN
Identify Address Frame	_FRM_AF_IDENTIFY
SSP Frame	_FRM_SSP
SMP Frame	_FRM_SMP
STP Frame	_FRM_STP
Incomplete STP Frame	_FRM_STP_INCOMPELETE
Training Sequence Frame	_FRM_TRAINING_SEQUENCE

Chapter 8: Declarations

8.1 Array Declarations

Scripts can declare an array object.

Format: **Array(source, size, default_value)**

Parameters

source (optional) List (collection or stand-alone value) of values to fill array.

size (optional) Size of array

default_value (optional) Default value to fill array, if size is greater than size of source.

Examples

```
set MyArray = Array( [1, 2, 3] );
# Creates an array of 3 elements from list : [1,2,3].

set MyArray = Array( [1, 2, 3, 4, 5, 6] );
# Creates an array  of 6 elements from list : [1,2,3,4,5,6 ].

set MyArray = Array( [1, 2, 3, 4, 5, 6], 20, 23 );
# Creates an array  of 20 elements. Fills first 6 elements from
# list : [1,2,3,4,5,6 ]. Aftewards, fills remainder with 23.

set MyArray = Array( 64 );
# Creates an array of one element [64].

set  MyArray = Array();
# Creates an empty array.
```

Remarks

Arrays can be iterated using the same syntax as lists (but more efficiently):

```
for( i = 0; i < sizeof(MyArray); i++ )
{
    ReportText( FormatEx( "MyArray[%d] = %d", i, MyArray[ i ] ) );
}
```

Arrays can use the operators **sizeof(), first(), more(), next(), prev(),** and **last()**.
Therefore, the example array code above can be written:

```
i = 0;
for( item = first(MyArray); more(MyArray); item = next(MyArray) )
{
    ReportText( FormatEx( "MyArray[%d] = %d", i, item ) );
    i++;
}
```

Arrays can grow and change dynamically through the functions: **InsertAt, RemoveAt,** and **SetAt:**

```
set MyArray = Array( [1, 2, 3, 4, 5, 6] );
```

```
InsertAt ( MyArray, 0, 0 ); # [0, 1, 2, 3, 4, 5, 6]
RemoveAt( MyArray, 0 );     # [1, 2, 3, 4, 5, 6]
SetAt( MyArray, 8, 8);      # [1, 2, 3, 4, 5, 6, null, 8]

# Use operators +  and +=.
set MyArray = Array( [1, 2, 3, 4, 5, 6] );
MyArray += [7, 8]; # [1, 2, 3, 4, 5, 6, 7, 8]
```

8.2 Map Declaration

Scripts can declare a map object.

Format: **Map(source, hash_size)**

Parameters

source — (optional) List (collection or stand-alone value) of values to fill map. The source paremeter (if present) MUST be a list of sublists, where the first item of the sublist MUST be either an integer or string and becomes a 'key', whereas the sublist itself becomes a 'value'.

hash_size — (optional) Hash size of map (default = 29)

Example 1

```
set WiNetFrameTable =
Map(
[
#-------------------------------------------------------------------
# Type , Type Name, Full Type name, Color, Decoder function|
#-------------------------------------------------------------------
 [0xFFFF, null, null , null, null ],
 [0, "DATA", "Standard Data", WN_DATA_COLOR, "ShowWNDataFrame" ],
 [1, "ABBR DATA", "Abbreviated Data", WN_ABBR_DATA_COLOR,
"ShowWNAbbrDataFrame" ],
 [2, "CONTROL", "Control Frame", WN_CONTROL_COLOR, "ShowCtrlFrame"],
 [3, "ASSOC", "Association Frame", WN_ASSOC_COLOR, "ShowAssocFrame"]
]
   );
```

The above example creates a map (hash-table) with the following key/value pairs:

```
key :    Value:
0xFFFF : [0xFFFF, null, null ,null, null ]
0      : [0, "DATA", "Standard Data", WN_DATA_COLOR,"ShowWNDataFrame"]
1      : [1, "ABBR DATA", "Abbreviated Data", WN_ABBR_DATA_COLOR,
"ShowWNAbbrDataFrame" ]
2      : [2, "CONTROL", "Control Frame", WN_CONTROL_COLOR,
"ShowCtrlFrame"]
3     : [3, "ASSOC", "Association Frame", WN_ASSOC_COLOR,
"ShowAssocFrame"]
```

This syntax can transform lists used like maps into map objects in existing scripts.

Example 2

```
# Example of map having mixed types of keys - both integer and string.

set MyMap = Map(
                 [
                     [0, 1, 2, 3],
                     [1, 3, 4, 5],
                     [2, 6, 7, 8],
                     ["JoJo", 4, 8, 9],
                     ["Papa", 14, 18, 19]
                 ]
                );
```

```
# Keys in the map object are : 0, 1, 2, "JoJo", "PaPa"

set MyMap = Map( ); # empty map
```

Example: Setting and getting values to or from map object

```
MyMap["JoJo"] = 12;     # Set pair: key : "JoJo", value : 12.
entry = MyMap["JoJo"]; # Get value at key : "JoJo".

MyMap[7] = [7, 8, 9, 10]; # Set pair: key : 7, value : [7, 8, 9, 10].
entry = MyMap[7];          # Get value at key : 7.
```

Remarks

Maps can use the operators **sizeof(), first(), more(),** and **next()**.

```
# Iterate through all elements of the map.

for( item = first(MyMap); more(MyMap); item = next(MyMap) )
    {
        ReportText("-------------------------------------------");

        if( IsString( item[0] ) )
            ReportText( FormatEx( "Item[0] = %s", item[0] ) );
        else
            ReportText( FormatEx( "Item0] = %d", item[0] ) );
    }
```

Maps can use the functions **IsNull(), IsInt(), IsString(), IsList(), IsArray(),** and **IsMap().** These functions determine a type of script object inside a script.

Chapter 9: Sending Functions

This topic contains information about the special group of VSE functions designed to specify which events the verification script should expect to receive.

NOTE: Performance of the script execution can be improvised drastically by providing/sending user-requirement-specific functions with specific options.
Avoid sending **ALL[Levels/Channels/Events]** when it is not required.

9.1 SendLevel()

This function specifies that events of the specified transaction level should be sent to the script. All layers and IDLE filtering are included by the **SendLevel(level)** method.

Format : **SendLevel(*level*)**

Parameters

level — This parameter can be one of following values:

_LINK	Send Link level events.
_TRANSPORT	Send Transport level events.
_IDLE	Send Idle events.
_OOB_SEQ	Send OOB Sequence events.
_ATA	Send ATA Command level events.
_SCSI	Send SCSI Command events.
_SMP	Send SMP Command events.
_TASK	Send Task Management Command events.

Example

```
SendLevel( _LINK);  # Send Link level events.
```

Remark

If NO level-sending-function was specified, no events will be sent to the script by default, and it generates an error.

9.2 SendLevelOnly()

This function specifies that ONLY events of the specified transaction level should be sent to the script.

Format : **SendLevelOnly(*level*)**

Parameters

level	This parameter can be one of the values of the **SendLevel()** command (see above).

Example

```
…
SendLevelOnly( _LINK ); # Send ONLY Link level events.
```

9.3 DontSendLevel()

This function specifies that events of the specified transaction level should NOT be sent to the script.

Format : DontSendLevel(*level*)

Parameters

level	This parameter can be one of the values of the **SendLevel()** command (see above).

Example

```
…
DontSendLevel( _LINK ); # DO NOT send Link level events.
```

9.4 SendChannel()

This function specifies that events that have occurred on the specified channel should be sent to script.

Format : SendChannel(*channel*)

Parameters

channel — This parameter can be one of following values:

_I1	(= 0)	Send events from Channel I1.
_T1	(= 1)	Send events from Channel T1.
_I1_1	(= 2)	Send events from Channel I1 MUX.
_T1_1	(= 3)	Send events from Channel T1 MUX.
_I2	(= 4)	Send events from Channel I2.
_T2	(= 5)	Send events from Channel T2.
_I2_1	(= 6)	Send events from Channel I2 MUX.
_T2_1	(= 7)	Send events from Channel T2 MUX.
_I3	(= 8)	Send events from Channel I3.
_T3	(= 9)	Send events from Channel T3.
_I3_1	(= 10)	Send events from Channel I3 MUX.
_T3_1	(= 11)	Send events from Channel T3 MUX.

and so on up to _I36, _T36, _I36_1, _T36_1.

Example

```
…
SendChannel(_I1); # Send events from Channel I1.
```

Remark

If NO channel-sending-function was specified, no events will be sent to the script by default, and it generates an error.

9.5 SendChannelOnly()

This function specifies that ONLY events that have occurred on the specified channel should be sent to the script.

Format : SendChannelOnly(*channel*)

Parameters

channel	This parameter can be one of the values of the **SendChannel()** command (see above).

Example

```
…
SendChannelOnly( _I1 ); # Send ONLY events from Channel I1.
```

9.6 DontSendChannel()

This function specifies that events that have occurred on the specified channel should NOT be sent to the script.

Format : DontSendChannel (*channel*)

Parameters

channel	This parameter can be one of the values of the **SendChannel()** command (see above).

Example

```
…
DontSendChannel ( _I1 );  # DO NOT send events from Channel I1.
```

9.7 SendAllChannels()

This function specifies to send all events with any channel.

.

Format : SendAllChannels ()

Example

```
…
SendAllChannels ();  # Send events from ALL channels.
```

9.8 SendTraceEvent()

This function specifies the events to be sent to the script.

Format : SendTraceEvent(*event*)

Parameters

event — Refer to Verification Script Engine Events for all possible values.

_FRM_PRIMITIVE	Primitve event
_FRM_OOB_SIGNAL	OOB Sequence Event
_FRM_AF_OPEN	Open Address Frame Event
_FRM_AF_IDENTIFY	Identify Address Frame Event
_FRM_SSP	SSP Frame Event
_FRM_SMP	SMP Frame Event
_FRM_STP	STP Frame Event
_FRM_STP_INCOMPELETE	Incomplete Frame Event
_FRM_DEVICE__SLEEP	Device Sleep Event

Example

```
…
SendTraceEvent( _FRM_AF_OPEN );
```

9.9 DontSendTraceEvent()

This function specifies that the event specified in this function should not be sent to script.

Format : DontSendTraceEvent (*event*)

Parameters

event Values are the same as for **SendTraceEvent()** (see above).

Example

```
…
if( SomeCondition )
{
      DontSendTraceEvent( _FRM_AF_OPEN );
}
```

9.10 SendTraceEventOnly()

This function specifies that ONLY the event specified in this function will be sent to the script.

Format : SendTraceEventOnly(*event*)

Parameters

event Values are the same as for **SendTraceEvent()** (see above).

Remark
This function may be useful when many events are to be sent, yet you need to send only one kind of event and turn off the rest.

Example

```
…
if( SomeCondition )
{
        SendTraceEventOnly (_FRM_AF_OPEN );
}
```

9.11 SendAllTraceEvents()

This function specifies that ALL trace events relevant for the selected transaction level will be sent to the script.

Format : SendAllTraceEvents ()

Example

```
…
SendAllTraceEvents ( );
```

9.12 SendProtocolError()

This function specifies that a selected Protocol Error event be sent to the script along with any other events (if sent any already)

Format : SendProtocolError(Protocol Error)

Parameters

Protocol Error	Possible values are as follows

```
_ANY
OOB_SEQUENCE_ERROR
SYMBOL_VIOLATION_ERROR
DISPARITY_ERROR
ALIGN_NOTIFY_ERROR
STP_SIGNALING_LATENCY_ERROR
STP_UNEXPECTED_PRIMITIVE_ERROR
STP_PRIMITIVE_RESPONSE_TIMEOUT_ERROR
FRAME_TYPE_ERROR
FRAME_LENGTH_ERROR
FRAME_DIRECTION_ERROR
CRC_ERROR
ACK_NAK_TIMEPOT_ERROR
```

Example

```
…
SendProtocolErrorOnly( OOB_SEQUENCE_ERROR );
SendProtocolError ( CRC_ERROR );
# This sends OOB_Sequence_Error events and CRC_Error events
# only
```

9.13 SendProtocolErrorOnly()

This function specifies that ONLY the selected Protocol Error event to be sent to the script and no other events to be sent.

Format : SendProtocolError(PError)

Parameters

Protocol Error	Possible values are the same as for **SendProtocolError()** (see above).

Example

```
…
SendProtocolErrorOnly( OOB_SEQUENCE_ERROR );
# This sends OOB_Sequence_Error events only.
```

Remark

If you send two or more events using **SendProtocolErrorOnly()**, then only the later function call will be sent to the script. Previous calls will be ignored. For example:

```
SendProtocolErrorOnly(CRC_ERROR);
SendProtocolErrorOnly(OOB_SEQUENCE_ERROR);.
```

Then only **OOB-Sequence_Error** events will be sent to the script.

9.14 DontSendProtocolError()

This function specifies NOT to send the selected Protocol Error event to the Verification script.

Format : DontSendProtocolError(Protocol Error)

Parameters

Protocol Error	Possible values are the same as for **SendProtocolError()** (see above).

Example

```
…
SendAllProtocolError();
DontSendProtocolError ( CRC_ERROR );
# Sends all protocol error events except CRC_Error Events.
```

9.15 SendPairSASAddress()

This function specifies a selected Pair SAS Address event to send to the script, along with any other events specified/sent before this.

Format : SendPairSASAddress(SourceSASAddress, DestinationSASAddress)

Parameters

SourceSASAddress [**format: raw data**]
DestinationSASAddress [**format: raw data**]

Pair-values can be any possible combinations of the following [except (_ANY, _ANY), which has no meaning]:

```
_ANY
'5000C500001047B5'
'5000E85000000001'
'5000C50000103D91'
… and so on
```

Example

```
…
SendPairSASAddressOnly ('5000E85000000001', '5000C50000103D91');
SendPairSASAddress ('5000C500001047B5', '5000C50000103D91');
# This sends Pair-1 & Pair-2 SAS Address events only.
```

9.16 SendPairSASAddressOnly()

This function specifies that ONLY the selected Pair-SAS-Address event be sent to the script, and no other events be sent.

Format : SendPairSASAddressOnly(Source SAS Address, Destination SAS Address)

Parameters

Source SAS Address [**format: raw data**] See **SendPairSASAddress()** above.
Destination SAS Address [**format: raw data**]

Example

```
…
SendPairSASAddressOnly ('5000E85000000001', '5000C50000103D91');
# Send only events with the above specified SAS Address Pair.
```

Remark

If you send two or more events using **SendPairSASAddressOnly()**, then only the later function call will be sent to the script. Previous calls will be ignored. For example,

```
SendPairSASAddressOnly('5000E85000000001', '5000C50000103D91');
# Pair – 1
SendPairSASAddressOnly('500062B000001074', '50060560000003C4');
# Pair – 2
```

Then only Pair-2 events will be sent to the script.

9.17 DontSendPairSASAddress()

This function specifies NOT to send a selected Pair-SAS-Address event to the script.

Format : DontSendPairSASAddress(Source SAS Address, Destination SAS Address)

Parameters

Source SAS Address [**format: raw data**] See **SendPairSASAddress()** above.
Destination SAS Address [**format: raw data**]

Example

```
…
SendAllPairSASAddress();
DontSendPairSASAddress('5000C500001047B5', '5000C50000103D91');
# This sends all events to the script except the one with a pair
# SAS address mentioned inside DontSendPairSASAddress()
```

9.18 SendAllPairSASAddress()

This function specifies to send events with all possible combinations of the above specified set of SAS Addresses.

Format : SendAllPairSASAddress()

Example

```
…
SendAllPairSASAddress();
```

9.19 SendPrimitive()

This function specifies to send a selected Primitive event to the script for verification.

Format : SendPrimitive(Primitive Code)

Parameters

Primitive Code	Works only for the following Primitive Codes: PRIM_CODE_ERROR PRIM_CODE_AIP_NORMAL (For a complete list of Primitive Codes, refer to Appendix-A.

Example

```
…
SendPrimitive(PRIM_CODE_DONE_NORMAL);
# Sends all Done-Normal Primitive event.
```

9.20 SendSSPTransport()

This function specifies to send a selected SSP Transport event to the script.

Format : **SendSSPTransport(Frame Type);**
SendSSPTransport(Frame Type, Field Name, Field Value, Nexus Info);

Parameters

Frame Type	Frame type of the SSP Transport event: _ANY SSP_FRAME_TYPE_COMMAND SSP_FRAME_TYPE_XFER_RDY SSP_FRAME_TYPE_RESPONSE SSP_FRAME_TYPE_TASK SSP_FRAME_TYPE_DATA SSP_FRAME_TYPE_VENDOR
Field Name	String form of any particular field name of the frame. For example: “Tag” / “HashedDestSASAddr” (Field names should be WITHOUT SPACES. NO need to Prefix or Suffix the field name with any of event or level-specific keywords, such as Trp_, Lnk_, or App_.)
Field Value	Value of the chosen field of the frame. Provide valid acceptable values according to the field type. For example: For string values, provide value surrounded by double-quotation marks. For Raw Data, surround the value using single-quotation marks. For Decimal or Hex-Decimal, provide the value as it is.

Nexus Info	Information about source and destination SAS address and Tag/Port. according to the command types
	Structure is:
	[Source SAS Address, Destination SAS Address, Tag/ Port]
	All fields are optional subject to usage of succeeding fields.
	
	For example:
	To set a value for destination address alone, set source address value to _ANY.
	To set only Tag/Port value, set preceding fields (Source SAS Address and Destination SAS Address) to _ANY.
	
	Tag / Port — Use Tag value only for SCSI Commands/SSP Frames
	Use Port value only for ATA Commands/STP Frames
	
	Valid structures for Nexus Info are:
	`NexusInfo = ['5000C500001047B5'];`
	`NexusInfo = ['5000C500001047B5', '5000E85000000001'];`
	`NexusInfo = [_ANY, '5000E85000000001'];`
	`NexusInfo = ['5000C500001047B5','5000E85000000001',10];`
	`NexusInfo = [_ANY, '5000E85000000001', 10];`
	`NexusInfo = ['5000C500001047B5', _ANY, 10];`
	`NexusInfo = [_ANY, _ANY, 10];`

Example

```
…
SendSSPTransport (SSP_FRAME_TYPE_XFER_RDY);

SendSSPTransport (SSP_FRAME_TYPE_XFER_RDY, "NumOfFillBytes", 0);

SendSSPTransport (SSP_FRAME_TYPE_XFER_RDY, "", 0,
                  '5000C500001047B5');

NexusInfo = ['5000C500001047B5', '5000E85000000001', 10];
# Or any other possible combinations of the Nexus Info

SendSSPTransport (SSP_FRAME_TYPE_XFER_RDY, "", 0, NexusInfo);
```

9.21 SendSTPTransport()

This function specifies to send a selected STP Transport event to the script.

Format : **SendSTPTransport(Frame Type);**
SendSTPTransport(Frame Type, Field Name, Field Value, Nexus Info);

Parameters

Frame Type	Frame type of the STP Transport event: _ANY STP_FIS_TYPE_REG_H2D STP_FIS_TYPE_REG_D2H STP_FIS_TYPE_DEV_BITS STP_FIS_TYPE_DMA_ACTIVATE STP_FIS_TYPE_DMA_SETUP STP_FIS_TYPE_BIST_ACTIVATE STP_FIS_TYPE_PIO_SETUP STP_FIS_TYPE_DATA STP_FIS_TYPE_VENDOR
Field Name	Refer to SendSSPTransport ().
Field Value	Refer to SendSSPTransport ().
Nexus Info	Refer to SendSSPTransport ().

Example

```
…
SendSTPTransport (STP_FIS_TYPE_DATA);

SendSTPTransport (STP_FIS_TYPE_DATA, "", 0, '5000C500001047B5');

NexusInfo = ['5000C500001047B5', '5000E85000000001', 10];
# Or any other possible combinations of the Nexus Info

SendSTPTransport (STP_FIS_TYPE_DATA, "", 0, NexusInfo);
```

9.22 SendSMPTransport()

This function specifies to send a selected SMP Transport event to the script.

Format : **SendSMPTransport(Frame Type);**
SendSMPTransport(Frame Type, Field Name, Field Value, Nexus Info);

Parameters

Frame Type	Frame type of the SMP Transport event: _ANY SMP_FRAME_TYPE_REQUEST SMP_FRAME_TYPE_RESPONSE
Field Name	Refer to SendSSPTransport ().
Field Value	Refer to SendSSPTransport ().
Nexus Info	Refer to SendSSPTransport ().

Example

```
…
SendSMPTransport (SMP_FRAME_TYPE_REQUEST);

SendSMPTransport (SMP_FRAME_TYPE_REQUEST, "PhyId", 1,
                  '5000C500001047B5');

NexusInfo = ['5000C500001047B5', '5000E85000000001', 10];
# Or any other possible combinations of the Nexus Info

SendSMPTransport (SMP_FRAME_TYPE_REQUEST, "PhyId", 1, NexusInfo);
```

9.23 SendSCSICommand()

This function specifies to send a selected SCSI Command event to the script.

Format : **SendSCSICommand(OpCode, ServiceAction, Field Name, Field Value, Nexus Info);**

Parameters

OpCode	Option Code of the SCSI Command event: _ANY SPC3_REPORT_LUNS : 0xA0 MMC4_READ_10 : 0x28 For a complete list of SCSI Command codes, refer to Appendix C.

ServiceAction:	Service Action of the SCSI Command event. The Value combines with the OpCode value and results in the formation of different combinations of SCSI Commands.
	OpCode + Service Action Value-1 = SCSI Command-1
	OpCode + Service Action Value-2 = SCSI Command-2
	Example:
	`0xA3 + 0x10 = MANAGEMENT_PROTOCOL_IN`
	`0xA3 + 0x0E = REPORT_PRIORITY`
	`0xA3 + 0x0F = REPORT_TIME_STAMP`

Field Name	Refer to SendSSPTransport ().
Field Value	Refer to SendSSPTransport ().
Nexus Info	Refer to SendSSPTransport ().

Example

```
…
SendSCSICommand (_ANY);

SendSCSICommand (0xA0); # Report LUN.

SendSCSICommand (0x28, _ANY, “”, 0, '500062B000001074');
# MMC4_READ_10
```

9.24 SendATACommand()

This function specifies to send a selected ATA Command event to the script.

Format : **SendATACommand(Command, Feature, Field Name, Field Value, Nexus Info);**

Parameters

Command	Command code of the ATA Command event: _ANY 0x25 : "READ DMA EXT"; 0x35 : "WRITE DMA EXT"; 0xD1 : "CHECK MEDIA CARD TYPE"; For complete list of ATA Command Codes, refer to Appendix-B.
Feature:	Feature value of the chosen command of ATA Command event. The Value combines with the Command code and results in the formation of different combinations of ATA Commands. Command Code + Feature Value-1 = ATA Command-1 Command Code + Feature Value-2 = ATA Command-2 For example: 0xE4 + 00 = 0xE400 READ_BUFFER 0xE4 + FF = 0xE4FF READ_PORT_MULTIPLIER
Field Name	Refer to SendSSPTransport ().
Field Value	Refer to SendSSPTransport ().
Nexus Info	Refer to SendSSPTransport ().

Example

```
…
SendATACommand (_ANY);

SendATACommand (0x35);  # Write DMA Ext.

SendATACommand (0x25, _ANY, "", 0, '500062B000001074',
                  '50060560000003C4', 0); # Read DMA Ext.
```

9.25 SendSMPCommand()

This function specifies to send a selected SMP Command event to the script.

Format : **SendSMPCommand(Function, Field Name, Field Value, Nexus Info);**

Parameters

Function	Function code of the SMP Command event: _ANY SMP_REPORT_GENERAL = 0x00; SMP_REPORT_MANUFACTURER_INFO = 0x01; SMP_DISCOVER = 0x10; SMP_REPORT_PHY_ERROR_LOG = 0x11; SMP_REPORT_PHY_SATA = 0x12; SMP_REPORT_ROUTE_INFO = 0x13; For a complete list of SMP Function and function-result codes, refer to Appendix D.
Field Name	Refer to SendSSPTransport ().
Field Value	Refer to SendSSPTransport ().
Nexus Info	Refer to SendSSPTransport ().

Example

```
…
SendSMPCommand (_ANY);

SendSMPCommand (0x00);  # Report general.

SendSMPCommand (0x10);  # Discover.

SendSMPCommand (0x11, "", 0, '500062B000001074');
# Report phy error log.
```

9.26 SendTaskMgmCommand()

This function specifies to send a selected Task Management Command event to the script.

Format : **SendTaskMgmCommand(Function, Field Name, Field Value, Nexus Info);**

Parameters

Function	Function code of the Task Management Command event: _ANY 0x01 : Abort Task 0x02 : Abort Task Set 0x04 : Clear Task Set 0x08 : Logical Unit Reset 0x10 : IT Nexus Reset 0x40 : Clear ACA 0x80 : Query Task 0x81 : Query Task Set 0x82 : Query Asynchronous Event
Field Name	Refer to SendSSPTransport ().
Field Value	Refer to SendSSPTransport ().
Nexus Info	Refer to SendSSPTransport ().

Example

```
…
SendTaskMgmCommand (_ANY);

SendTaskMgmCommand (0x01); # Abort task.
```

Chapter 10: Time Construction Functions

This group of functions constructs VSE time objects.

10.1 Time()

Constructs a verification script time object.

Format : **Time(*nanoseconds*)**
Time(*seconds, nanoseconds*)

Return values

First function returns **[0, nanoseconds],** and second returns **[seconds, nanoseconds].**

Parameters

nanoseconds Number of nanoseconds in specified time

seconds Number of seconds in specified time

Example

```
Time ( 50 * 1000 ); # Create time object of 50 microseconds.
Time (3, 100); # Create time object of 3 seconds and 100 nanoseconds.
Time( 3 * MICRO_SECS ); # Create time object of 3 microseconds.
Time( 4 * MILLI_SECS ); # Create time object of 4 milliseconds.
```

Note: MICRO_SECS and MILLI_SECS are constants defined in ***VS_constants.inc***.

Chapter 11: Timer Functions

This group of functions covers VSE capability to work with timers –-- internal routines that repeatedly measure a timing intervals between different events.

Using VSE Timer Functions is a stage-by-stage process of 'Set the Timer', followed by 'Get the Timer Time' and finally 'Kill the Timer'. All three stages are interdependent on each other using a unique timer identifier, which can be any user-defined positive integer.

Prototypes for each of the functions, along with their corresponding usage explanations, are provided below.

11.1 SetTimer()

Sets/starts timer for timing calculation from the event where this function was called.

Format : SendTimer(*timer_id* = 0)

Parameter

timer_id Unique timer identifier (user-defined positive integer)

Example

```
SetTimer(0);        # Start timing for timer with id = 0;
SetTimer(23);       # Start timing for timer with id = 23;
```

Remark

If this function is called a second time with the same **timer id**, it resets the timer and starts timing calculations again from the point where it was called.

11.2 GetTimerTime()

Calculates and returns the timing interval (VSE Time object) between the Time-at **SetTimer()** and Current Time. For example, if
T1 is the time when SetTimer(<timer_id>) is being called for an event; and
T2 is the current time when GetTimerTime(<timer_id>) is called.
Then
T = T2 – T1 is the timing interval returned by the **GetTimerTime(<timer_id>)** function, where **T** is of type **VSE Time object,** which can be converted into a string using the **TimeToText()** function (refer to Chapter 13 for detailed explanation).

Format : GetTimerTime (*timer_id* = 0)

Parameters

timer_id Unique timer identifier

Return value

Returns VSE time object from timer with **id = timer_id**.

Example

```
GetTimerTime ();   # Retrieve timing interval for timer with id = 0;
GetTimerTime (23); # Retrieve timing interval for timer with id = 23;
```

Remark

This function, when called, does not reset the timer.

11.3 KillTimer()

Kills/stops a specific running Timer (with Timer-ID = <timer_id> used by **SetTimer()**) and frees its corresponding resources.

Format : KillTimer(*timer_id* = 0)

Parameter

timer_id	Unique timer identifier

Example

```
KillTimer();          # Stop timing for timer with id = 0;
KillTimer(23);        # Stop timing for timer with id = 23;
```

Chapter 12: Time Calculation Functions

This group of functions covers VSE capability to work with "time" — VSE time objects.

12.1 AddTime()

Adds two VSE time objects.

Format : **AddTime(*time1*, *time2*)**

Return values

Returns VSE time object representing the time interval equal to the sum of **time_1** and **time_2.**

Parameters

time_1 VSE time object representing the first time interval

time_2 VSE time object representing the second time interval

Example

```
t1 = Time(100);      # 100 nanoseconds
t2 = Time(2, 200);   # 2 seconds 200 nanoseconds
t3 = AddTime( t1, t2 ) # Returns T1 + t2 = 2 sec 300 ns.
```

12.2 SubtractTime()

Subtracts two VSE time objects.

Format : **SubtractTime (*time1, time2*)**

Return values

Returns VSE time object representing the time interval equal to the difference between **time_1** and **time_2.**

Parameters

time_1 VSE time object representing the first time interval

time_2 VSE time object representing the second time interval

Example

```
t1 = Time(100);      # 100 nanoseconds
t2 = Time(2, 200);   # 2 seconds 200 nanoseconds
t3 = SubtractTime ( t2, t1 ) # Returns t2 - t1 = 2 sec 100 ns.
```

12.3 MulTimeByInt()

Multiplies VSE time object by integer value.

Format : **MulTimeByInt (*time, mult*)**

Return values

Returns VSE time object representing the time interval equal to the product of **time * mult.**

Parameters

time	VSE time object
mult	multiplier, integer value

Example

```
t    = Time(2, 200);          # 2 seconds 200 nanoseconds
t1 = MulTimeByInt ( t, 2 )   # Returns t * 2 = 4 sec 400 ns.
```

12.4 DivTimeByInt()

Divides VSE time object by integer value.

Format : **DivTimeByInt (*time, div*)**

Return values

Returns VSE time object representing the time interval equal to the quotient of **time / div.**

Parameters

time	VSE time object
div	divisor, integer value

Example

```
t   = Time(2, 200);         # 2 seconds 200 nanoseconds
t1 = DivTimeByInt ( t, 2 ) # Returns t / 2 = 1 sec 100 ns.
```

Chapter 13: Time Logical Functions

This group of functions covers VSE capability to compare VSE time objects.

13.1 IsEqualTime()

Verifies whether one VSE time object is equal to the other VSE time object.

Format : **IsEqualTime (*time1*, *time2*)**

Return values

Returns 1 if **time_1** is equal to **time_2**. Returns 0 otherwise.

Parameters

time_1 VSE time object representing the first time interval

time_2 VSE time object representing the second time interval

Example

```
t1 = Time(100);  t2 = Time(500);
If( IsEqualTime( t1, t2 ) ) DoSomething();
```

13.2 IsLessTime()

Verifies whether or not one VSE time object is less than the other VSE time object

Format : **IsLessTime (*time1, time2*)**

Return values

Returns 1 if **time_1** is less than **time_2**. Returns 0 otherwise.

Parameters

time_1 VSE time object representing the first time interval

time_2 VSE time object representing the second time interval

Example

```
t1 = Time(100);  t2 = Time(500);
If( IsLessTime ( t1, t2 ) ) DoSomething();
```

13.3 IsGreaterTime()

Verifies whether or not one VSE time object is greater than the other VSE time object.

Format : **IsGreaterTime (*time1, time2*)**

Return values

Returns 1 if **time_1** is greater than **time_2**. Returns 0 otherwise.

Parameters

time_1	VSE time object representing the first time interval
time_2	VSE time object representing the second time interval

Example

```
t1 = Time(100);  t2 = Time(500);
If( IsGreaterTime ( t1, t2 ) ) DoSomething();
```

13.4 IsTimeInInterval()

Verifies that a VSE time object is greater than some VSE time object and less than the other VSE time object.

Format : **IsTimeInInterval(min_time, time, max_time)**

Return values

Returns 1 if **min_time** <= time <= **max_time**. Returns 0 otherwise.

Parameters

min_time VSE time object representing the first time interval

max_time VSE time object representing the second time interval

Example

```
t1 = Time(100);
t  = Time(400);
t2 = Time(500);
If( IsTimeInInterval ( t1, t, t2 ) ) DoSomething();
```

Chapter 14: Time Text Functions

This group of functions covers VSE capability to convert VSE time objects into text strings.

14.1 TimeToText()

Converts a VSE time object into text.

Format : **TimeToText (*time*)**

Return values

Returns a text representation of VSE time object.

Parameters

time VSE time object

Example

```
t = Time(100);
ReportText( TimeToText( t ) ); # See ReportText() function below.
```

Chapter 15: Output Functions

This group of functions covers VSE capability to Enable, Disable, and Display of information/results in the output window.

15.1 EnableOutput()

Enables showing information in the output window and sending COM reporting notifications to COM clients.

Format : **EnableOutput ()**

Example

```
EnableOutput ( );
```

15.2 DisableOutput()

Disables showing information in the output window and sending COM reporting notifications to COM clients.

Format : **DisableOutput ()**

Example

```
DisableOutput ();
```

15.3 ReportText()

Reports/Displays textual information in the output window of verification script engine.

Format : **ReportText (*text*)**

Parameters

text	string variable or a constant string

Example

```
…
ReportText ( "Some text" );
…
t = "Some text"
ReportText ( t );
…
num_of_frames = in.NumOfFrames;
text = Format( "Number of frames : %d",  num_of_frames );
ReportText ( text );
…
x = 0xAAAA;
y = 0xBBBB;
text = FormatEx( "x = 0x%04X,  y = 0x%04X", x, y );
ReportText( "Text : " + text );
…
```

Chapter 16: Information Functions

16.1 GetTraceName()

This function returns the filename of the trace file being processed by VSE.
If the script is being run over a multi-segmented trace, this function returns the path to the current segment being processed.

Format : **GetTraceName(filepath_compatible)**

Parameters

filepath_compatible	If this parameter is present and not equal to 0, the returned value may be used as part of the filename.

Example

```
ReportText( "Trace name : " + GetTraceName() );
…
File = OpenFile( "C:\\My Files\\" +  GetTraceName(1) + "_log.log" );

#  For trace file with path    -  D:\Some SAS Traces\Data.sas
#  GetTraceName(1) will return – "D_Some SAS Traces_Data.sas"
```

16.2 GetScriptName()

This function returns the name of the verification script where this function is called.

Format : **GetScriptName()**

Example

```
ReportText( "Current script : " + GetScriptName() );
```

16.3 GetApplicationFolder()

This function returns the full path of the folder where the SAS*Suite*/SATA*Suite* application was started.

Format : **GetApplicationFolder()**

Example

```
ReportText( "SASTracer folder : " + GetApplicationFolder () );
```

16.4 GetCurrentTime()

This function returns the string representation of the current system time.

Format : GetCurrentTime()

Example

```
ReportText( GetCurrentTime() ); # For example, will report
                                #"February 10, 2006 5:49 PM"
```

16.5 GetEventSegNumber()

If a multi-segmented trace is being processed, this function returns the index of the segment for the current event.

Note: When a multi-segmented trace file (extension ***.smt** or ***.smat**) is processed by VSE, different trace events in different segments of the same trace file may have the same indexes (value stored in **in.Index** input context members), but they will have different segment numbers.

Format : **GetEventSegNumber()**

Example

```
ReportText( Format( "Current segment =  %d", GetEventSegNumber() ) );
```

16.6 GetTriggerPacketNumber()

This function returns the index of a link packet for which a Trigger was set. In case no trigger event was recorded in the trace, a value of 0xFFFFFFFF is returned.

Format : **GetTriggerPacketNumber()**

Example

```
ReportText( FormatEx( "Trigger packet # : %i",
                      GetTriggerPacketNumber() );
```

16.7 GetTraceStartPacketTime()

This function returns time stamp of the first packet in trace file.

Format : **GetTriggerPacketNumber()**

Return Value

Returns VSE time object.

Example

```
Time = GetTraceStartPacketTime();
ReportText(GetFullTimestamp(Time)); #print time stamp
```

16.8 GetTraceEndtPacketTime()

This function returns time stamp of the last packet in trace file.

Format : **GetTriggerPacketNumber()**

Return Value

Returns VSE time object.

Example

```
Time = GetTraceEndPacketTime();
ReportText(GetFullTimestamp(Time)); #print time stamp
```

Chapter 17: Navigation Functions

17.1 GotoEvent()

This function forces the application to jump to some trace event and show it in the main trace view.

Format : **GotoEvent(level, index, segment)**
GotoEvent()

Parameters

level	Transaction level of the event to jump to
index	Transaction index of the event to jump to
segment	Segment index of the event to jump to. If omitted, the current segment index is used.

Remarks

If no parameters were specified, the application will jump to the current event being processed by VSE.
The **segment** parameter is used only when the verification script is running over a multi-segmented trace (extensions: ***.smt, *.samt**). For regular traces, it is ignored.
If wrong parameters were specified (such as an index exceeding the maximum index for the specified transaction level), the function will do nothing, and an error message will be sent to the output window.

Example

```
…
if( Something == interesting ) GotoEvent(); # Go to the current event.
…
if( SomeCondition )
 {
    interesting_segment = GetEventSegNumber();
    interesting_level   =  in.Level;
    interesting_index   =  in.Index;
 }
 …
 OnFinishScript()
{
    …
     # Go to the interesting event.
     GotoEvent( interesting_level, interesting_index,
                interesting_segment );
}
```

17.2 SetMarker()

This function sets a marker (bookmark) for some trace event.

Format : **SetMarker(marker_text)**
SetMarker(marker_text, level, index, segment)

Parameters

marker_text	Text of the marker
level	Transaction level of the event to jump to
index	Transaction index of the event to jump to
segment	Segment index of the event to jump to. If omitted, the current segment index is used.

Remarks
If no parameters were specified, other than **marker_text**, the application sets a marker to the current event being processed by VSE.
The **segment** parameter is used only when a verification script is running over a multi-segmented trace (extensions: ***.smt, *.samt)**. For regular traces, it is ignored.
If wrong parameters were specified (such as an index exceeding the maximum index for a specified transaction level), the function will do nothing, and an error message will be sent to the output window.

Example

```
…
# Set marker to the current event.
if( Something == interesting ) SetMarker( "!!! Something cool !!!" );
…
if( SomeCondition )
 {
    interesting_segment = GetEventSegNumber();
    interesting_level   =  in.Level;
    interesting_index   =  in.Index;
 }
 …
 OnFinishScript()
{
…
# Set marker to the interesting event.
SetMarker( " !!! Cool Marker !!! ", interesting_level,
          interesting_index, interesting_segment );

# Go to the interesting event.
GotoEvent(interesting_level, interesting_index, interesting_segment);
}
```

Chapter 18: File Functions

This group of functions covers VSE capabilities to work with the external files.

18.1 OpenFile()

This function opens a read-write file for writing or appending content.

Format : **OpenFile(*file_path, file_creation_type = 0, file_open_type = 0*)**

Parameters

file_path	Full path of the file to open (for '\' use '\\') If the path is not specified, the file is in the USER folder.
file_creation_type	_FC_RW_CREATE Writes new content to the file by replacing the previous content. **Note:** _FC_RW_CREATE ERASEs the file content (if any) while it is opening. _FC_RW_APPEND Appends/adds new content at the end of the previous file content without replacing the same.
file_open_type	_FO_TEXT opens as a Text document (*.txt). _FO_BINARY opens as a Binary file.

Return Values

The "handle" to the file to be used in other file functions.

Example

```
…
set file_handle = 0;

# Opens file with default options (“<path>”, 0, 0), to open as
# a text document and to overwrite its previous contents.
file_handle  = OpenFile( "D:\\Log.txt" );
…
# Write text string to file.
WriteString(file_handle, "Some Text1" );
# Write text string to file.
WriteString(  file_handle, "Some Text2" );
…
CloseFile( file_handle ); # Closes file.

# Opens file in Binary mode with _APPEND functionality.
file_handle  = OpenFile("Sample.bin", _FC_RW_APPEND, _FO_BINARY
);
…
# Write text string to file.
WriteBin(  file_handle, "Some Text1" );
# Write integer to file.
WriteBin(  file_handle, 0x1234ABCD);
…
CloseFile( file_handle );   # Closes file.
```

18.2 WriteString()

This function writes a text string to the file.

Format : **WriteString(*file_handle, text_string*)**

Parameters

file_handle File "handle"

text_string Text string

Example

```
…
set file_handle = 0;
…
file_handle  = OpenFile( "D:\\Log.txt" ); # Opens file.
                                    # The previous contents are erased.
…
WriteString( file_handle, "Some Text1" ); # Write text string to file.
WriteString( file_handle, "Some Text2" ); # Write text string to file.
…
CloseFile( file_handle ); # Closes file.
…
```

18.3 CloseFile()

This function closes an opened file.

Format : **CloseFile(*file_handle*)**

Parameters

file_handle File "handle"

Example

```
…
set file_handle = 0;
…
file_handle  = OpenFile( "D:\\Log.txt" ); # Opens file.
                                    # The previous contents are erased.
…
WriteString( file_handle, "Some Text1" ); # Write text string to file.
WriteString( file_handle, "Some Text2" ); # Write text string to file.
…
CloseFile( file_handle ); # Closes file.
…
```

18.4 WriteBin ()

This function writes content to a binary file.

Format : **WriteBin(*bin_file_handle, variable*)**

Parameters

bin_file_handle	File handle for the binary file (only opened using _FO_BINARY), to which the contents are to be written.
variable	Any variable type (integer, string, list, and raw string)

Return Value

_BF_RW_OK	Read/Write binary returns succesfully
_BF_RW_INVALID_HANDLE (1)	Invalid File Handle
_BR_RW_INALID_VALUE	Invalid Value

Example

```
…
# Opens file in binary mode with _CREATE functionality.
bin_file_handle = OpenFile("Sample.bin", _FC_RW_CREATE,
                           _FO_BINARY );
…
raw_data = '00112233445566778899AABBCCDD';
list_val = ["one", 2, "three", [4, [5, [6]]]];

# Write text string to file.
WriteBin( bin_file_handle, "Some Text1" );
# Write integer to file.
WriteBin( bin_file_handle, 0x1234ABCD);
# Write raw data to file.
WriteBin( bin_file_handle, raw_data);
# Write list to file.
WriteBin( bin_file_handle, list_val);
…
CloseFile( bin_file_handle ); # Closes file.
```

18.5 ReadBin ()

This function reads content from a binary file.

Case 1: Open and Read the content from a binary file
The file should be opened ONLY in _FC_RW_APPEND mode (because _FC_RW_CREATE mode erases all the content of the file while it is opening). Because APPEND mode drags the file pointer to the end of the file, use the **SeekToBegin()** function ito start reading from the beginning of the file.

Case 2: Reading the just-written content from an already opened binary file.
The file might have already been opened in CREATE or APPEND mode (to write the content), so to read the written content, use the **SeekToBegin()** function, as explained in case 1 above, and start reading sequentially.

Before reading a binary file, you should know the content inside the file in terms of its order of data types. The **ReadBin** function should be called in the exact sequence of the **WriteBin** function. (Refer to the example of **WriteBin()** to compare the sequence of data read/write from/to file.)

Format : **ReadBin(*bin_file_handle*)**

Parameters

bin_file_handle	File handle for the binary file (only opened using _FO_BINARY) from which the contents are to be read. **Note:** Opening the file in _FC_RW_CREATE mode erases content (if any) of the file.

Return Value
Content of the opened binary file, which can be any data type (integer, string, raw data, and so on).

Example

```
# This example demonstrates CASE 1, how to open and read data.
# Opens file in binary mode with _APPEND functionality,
# to read data.
bin_file_handle = OpenFile("Sample.bin", _FC_RW_APPEND,
                           _FO_BINARY );
…
# Start reading from beginning of file.
SeekToBegin( bin_file_handle );
# "Some Text1"
string_val = ReadBin( bin_file_handle);
# 0x1234ABCD
integer_val= ReadBin( bin_file_handle);
# '00112233445566778899AABBCCDD'
raw_data = ReadBin( bin_file_handle,);
# ["one", 2, "three", [4, [5, [6]]]]
list_val = ReadBin( bin_file_handle);
…
CloseFile( bin_file_handle ); # Closes file.
```

……

```
# This example demonstrates CASE 2, how to read data from an
# already opened and written file.
# Opens file in binary mode with _CREATE functionality.
bin_file_handle = OpenFile("Sample.bin", _FC_RW_CREATE,
                           _FO_BINARY );
…
raw_data = '00112233445566778899AABBCCDD';
list_val = ["one", 2, "three", [4, [5, [6]]]];

# Write text string to file.
WriteBin( bin_file_handle, "Some Text1" );
# Write integer to file.
WriteBin( bin_file_handle, 0x1234ABCD);
# Write raw data to file.
WriteBin( bin_file_handle, raw_data);
# Write list to file.
WriteBin( bin_file_handle, list_val);
…
# Read the just-written content of this binary file.
# Start reading from beginning of file.
SeekToBegin( bin_file_handle );

# "Some Text1"
string_val = ReadBin( bin_file_handle);
# 0x1234ABCD
integer_val= ReadBin( bin_file_handle);
# '00112233445566778899AABBCCDD'
raw_data1 = ReadBin( bin_file_handle,);
# ["one", 2, "three", [4, [5, [6]]]]
list_val1 = ReadBin( bin_file_handle);
…
CloseFile( bin_file_handle ); # Closes file.
```

18.6 SeekToBegin ()

This function seeks the file read-write pointer from current position to the BEGIN of the file.

Format : **SeekToBegin(*file_handle*)**

Parameters

file_handle	Handle of the file (Text or Binary) on which the Seek funcitonality has to be performed

Example

```
…
SeekToBegin (file_handle);
…
```

18.7 SeekToEnd()

This function seeks the file read-write pointer from current position to the END of the file.

Format : **SeekToEnd(*file_handle*)**

Parameters

file_handle	Handle of the file (Text or Binary) on which the Seek funcitonality has to be performed

Example

```
…
SeekToEnd (file_handle);
…
```

18.8 ShowInBrowser()

This function allows you to open a file in the Windows Explorer. If the extension of the file has the application registered to open files with such extensions, it will launch.

For example, if Internet Explorer is registered to open files with extension ***.htm,** and the file handle passed to the **ShowInBrowser()** function belongs to a file with such an extension, this file opens in Internet Explorer.

Format : **ShowInBrowser (*file_handle*)**

Parameters

file_handle File "handle"

Example

```
…
set html_file = 0;
…
html_file = OpenFile( "D:\\Log.htm" );
…
WriteString(  html_file, "<html><head><title>LOG</title></head>" );
WriteString(  html_file, "<body>" );
…
WriteString(  html_file, "</body></html>" );
ShowInBrowser( html_file ); # Opens the file in Internet Explorer.
CloseFile( html_file );
…
```

Chapter 19: COM/Automation Communication Functions

This group of functions covers VSE capabilities to communicate with COM/Automation clients connected to the SAS*Suite*/SATA*Suite* application. (Refer to the SAS*Suite*/SATA*Suite* *Automation* manual for the details on how to connect to the SAS*Suite*/SATA*Suite* application and VSE.)

19.1 NotifyClient()

This function allows you to send information to COM/Automation client applications in a custom format. The client application will receive a VARIANT object, which it is supposed to parse.

Format : **NotifyClient(*event_id, param_list*)**

Parameters

event_id	Event identifier
param_list	List of parameters to be sent to the client application. Each parameter might be an integer, string, or list. (See *CSL Manual* for details about data types available in CSL.) Because the list itself may contain integers, strings, or other lists, it is possible to send complicated messages. (Lists should be treated as arrays of VARIANTs.)

Example

```
…
if( SomeCondition() )
{
      NotifyClient( 2, [ in.Index, in.Level,
                  TimeToText( in.Time )] );
}
…
# Here we sent 2 parameters to client applications :
# 2 ( integer ) and
# [ in.Index, in.Level, TimeToText( in.Time )] ( list )
```

Remark

See an example of handling this notification by client applications and parsing code in the SAS*Suite*/SATA*Suite Automation* document.

Chapter 20: User Input Functions

20.1 MsgBox()

Displays a dialog-based message in a dialog box, waits for the user to click a button, and returns an Integer indicating which button the user clicked.

Format : **MsgBox(*prompt, type, title*)**

Parameters

prompt — Required. String expression displayed as the message in the dialog box.

type — Optional. Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box.
If omitted, the default value for buttons is **_MB_OK**.
(See the list of possible values in the table below.)

The **type** argument values are:

Constant	Description
_MB_OKONLY	Display **OK** button only (by Default).
_MB_OKCANCEL	Display **OK** and **Cancel** buttons.
_MB_RETRYCANCEL	Display **Retry** and **Cancel** buttons.
_MB_YESNO	Display **Yes** and **No** buttons.
_MB_YESNOCANCEL	Display **Yes**, **No**, and **Cancel** buttons.
_MB_ABORTRETRYIGNORE	Display **Abort**, **Retry**, and **Ignore** buttons.
_MB_EXCLAMATION	Display **Warning Message** icon.
_MB_INFORMATION	Display **Information Message** icon.
_MB_QUESTION	Display **Warning Query** icon.
_MB_STOP	Display **Critical Message** icon.
_MB_DEFBUTTON1	First button is default.
_MB_DEFBUTTON2	Second button is default.
_MB_DEFBUTTON3	Third button is default.
_MB_DEFBUTTON4	Fourth button is default.

title — Optional. String expression displayed in the title bar of the dialog box.
If you omit the title, the script name is placed in the title bar.

Return Values

This function returns an integer value indicating which button the user clicked.

Constant	**Description**

_MB_OK	**OK** button was clicked.
_MB_CANCEL	**Cancel** button was clicked.
_MB_YES	**Yes** button was clicked.
_MB_NO	**No** button was clicked.
_MB_RETRY	**Retry** button was clicked.
_MB_IGNORE	**Ignore** button was clicked.
_MB_ABORT	**Abort** button was clicked.

Remark

This function works only for VS Engines controlled via the GUI. For VSEs controlled by COM/Automation clients, it does nothing.

This function "locks" the SAS*Suite*/SATA*Suite* application, which means that there is no access to other application features until the dialog box is closed. In order to prevent too many MsgBox calls, in the case of a script not written correctly, VSE keeps track of all function calls demanding user interaction and does not show dialog boxes if a customizable dialog limit was exceeded (and returns **_MB_OK** in this case).

Example

```
…
if( Something )
{
…
str = "Something happened!!!\nShould we continue?"
result = MsgBox( str , _MB_YESNOCANCEL | _MB_EXCLAMATION,
                 "Some Title" );

if( result != _MB_YES )
ScriptDone();
… # Go on…
}
```

20.2 InputBox()

Displays a dialog-based input prompt in a dialog box, waits for the user to input text or click a button, and returns a CSL list object (see the CSL Manual for details about list objects) or a string containing the contents of the text box.

Format : **InputBox(*prompt, title, default_text, return_type*)**

Parameters

prompt	Required. String expression displayed as the message in the dialog box.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit the title, the script name is placed in the title bar.
default_text	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit default_text, the text box is displayed empty.
return_type	Optional. Specifies the contents of the return object.

The **return_type** argument values are:

Constant	Value	Description
_IB_LIST	0	CSL list object will be returned (by default).
_IB_STRING	1	String input as it was typed in the text box

Return Values

Depending upon the **return_type** argument, this function returns either a CSL list object or the text typed in the text box.

In case of **return_type = _IB_LIST** (by default), the text in the text box is considered as a set of list items delimited by ',' (only hexadecimal, decimal, and string items are currently supported).

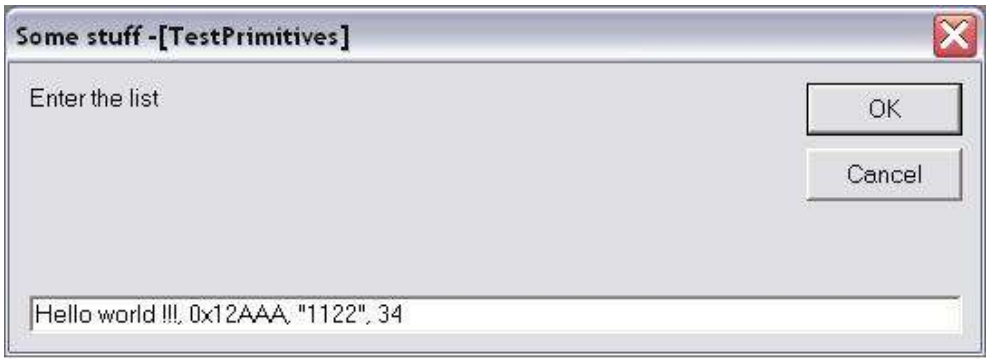
Text example

Hello world !!!, 12, Something, 0xAA, 10, "1221"

Will produce a CSL list object of five items:

```
list = [ "Hello world !!!", 12, "Something", 0xAA, 10, "1221" ];
list [0] = "Hello world !!!"
list [1] = 12
list [2] = "Something"
list [3] = 0xAA
list [4] = 10
list [5] = "1212"
```

Note: Although the dialog box input text parser tries to determine a type of list item automatically, a text enclosed in quote signs "" is always considered a string.



Remark

This function works only for VS Engines controlled via the GUI. For VSEs controlled by COM/Automation clients, it does nothing.

This function "locks" the SAS*Suite*/SATA*Suite* application, which means that there is no access to other application features until the dialog box is closed. In order to prevent too many **InputBox** calls, in the case of a script not written correctly, VSE keeps track of all function calls demanding user interaction and does not show dialog boxes if a customizable limit was exceeded (and returns **null** object in that case).

Example

```
…
if( Something )
{
…
v = InputBox( "Enter the list", "Some stuff", "Hello world !!!,
               0x12AAA, Some, 34" );
ReportText ( FormatEx( "input = %s, 0x%X, %s, %d", v[0], v[1], v[2],
             v[3] ) );
… # Go on…

str =  InputBox( "Enter the string", "Some stuff", "<your string>",
                 _IB_STRING  );
ReportText( str );
}
```

20.3 GetUserDlgLimit()

This function returns the current limit of user dialogs (message box or input box) allowed in the verification script. If the script reaches this limit, no user dialogs will be shown, and the script will not stop. By default, this limit is set to 20.

Format : **GetUserDlgLimit()**

Example

```
…
result = MsgBox( Format( "UserDlgLimit = %d", GetUserDlgLimit() ),
                 _MB_OKCANCEL | _MB_EXCLAMATION, "Some Title !!!" );

SetUserDlgLimit( 2 ); # Set the limit to 2.
…
```

20.4 SetUserDlgLimit()

This function sets the current limit of user dialogs allowed in the verification script. If the script reaches this limit, no user dialogs will pop up, and script will continue with its next instruction statement. By default, this limit is set to 20.

Format : **SetUserDlgLimit()**

Example

```
…
result = MsgBox( Format( "UserDlgLimit = %d", GetUserDlgLimit() ),
                 _MB_OKCANCEL | _MB_EXCLAMATION, "Some Title !!!" );

SetUserDlgLimit( 2 ); # Set the limit to 2.
…
```

Chapter 21: String Manipulation/Formatting Functions

21.1 FormatEx()

This function writes formatted data to a string. **FormatEx()** controls the way that arguments print. The format string may contain conversion specifications that affect the way in which the arguments in the value string are returned. Format conversion characters, flag characters, and field-width modifiers define the conversion specifications.

Format : **FormatEx (*format_string, argument_list*)**

Parameters

format_string Format-control string

argument_list Optional list of arguments to fill in the format string

Return Values

Formatted string.

Conversion Characters

Format conversion characters are:

Code	Type	Output
c	Integer	Character
d	Integer	Signed decimal integer
i	Integer	Signed decimal integer
o	Integer	Unsigned octal integer
u	Integer	Unsigned decimal integer
x	Integer	Unsigned hexadecimal integer, using "abcdef."
X	Integer	Unsigned hexadecimal integer, using "ABCDEF."
s	String	String

Remark

A conversion specification begins with a percent sign (%) and ends with a conversion character. The following optional items can be included, in order, between the % and the conversion character, to further control argument formatting:

Flag characters are used to further specify the formatting. There are five flag characters:

- A minus sign (-) causes an argument to be left-aligned in its field. Without the minus sign, the default position of the argument is right-aligned.

- A plus sign (+) inserts a plus sign before a positive signed integer. This only works with the conversion characters **d** and **i**.

- A space inserts a space before a positive signed integer. This only works with the conversion characters **d** and **i**. If both a space and a plus sign are used, the space flag is ignored.

- A hash mark (#) prepends a 0 to an octal number when used with the conversion character **o.** If # is used with **x** or **X**, it prepends *0x* or 0*X* to a hexadecimal number.

- A zero (*0*) pads the field with zeros instead of with spaces.

Field width specification is a positive integer that defines the field width, in spaces, of the converted argument. If the number of characters in the argument is smaller than the field width, then the field is padded with spaces. If the argument has more characters than the field width has spaces, then the field will expand to accommodate the argument.

Example

```
str = "String";
i = 12;
hex_i = 0xAABBCCDD;
…
formatted_str = FormatEx( "%s, %d, 0x%08X", str, i, hex_i );

# formatted_str = "String, 12, 0xAABBCCDD"
```

21.2 Left(), Right(), and Mid()

The **Left(), Right()**, and **Mid()** functions extract substrings.

Format : **Left ( )**
Right ( )
Mid ( )

21.3 Get and Set Characters in Strings

You can set and get characters at positions in CSL-script-file string objects.

Format : **str [index]**

Parameter

index	Position in string

Example

```
str = "Brush your teeth before going to bed";

# Get character at some index.
c = str[ 6 ];  # c = "y"

# Set characters or substring at some index.
str[ 11 ] = "hands"; # str = "Brush your hands before going to bed";

str[ 33 ] = "sleep"; # str = "Brush your hands before going to sleep";
```

Chapter 22: Miscellaneous Functions

22.1 ScriptForDisplayOnly()

This function specifies that the script is designed for displaying information only and that its author does not care about the verification script result. Such a script has a result of **DONE** after execution.

Format : **ScriptForDisplayOnly ()**

Example

```
ScriptForDisplayOnly();
#Use this function call anywhere in the OnStartScript() function.
```

22.2 Sleep()

This function asks VSE not to send any events to a script until the timestamp of the next event is greater than the timestamp of the current event plus sleeping time.

Format : **Sleep(*time*)**

Parameters

time	VSE time object specifying sleep time

Example

```
Sleep ( Time(1000) ); # Don't send any event occurred during 1 ms
                      # from the current event.
```

22.3 ConvertToHTML()

This function replaces spaces with " " and carriage return symbols with "
" in a text string.

Format : **ConvertToHTML(*text_string*)**

Parameters
text_string Text string

Example

```
str  =  "Hello world !!!\n";
str += "How are you today?";

html_str = ConvertToHTML ( str );
# html_string =
"Hello&nbspworld !!!<br>How&nbspare&nbspyou&nbsptoday?"
```

Note: Some other useful miscellaneous functions can be found in the file **VSTools.inc**.

22.4 Pause()

Pauses a running script. Later, script execution can be resumed or cancelled.

Format : **Pause()**

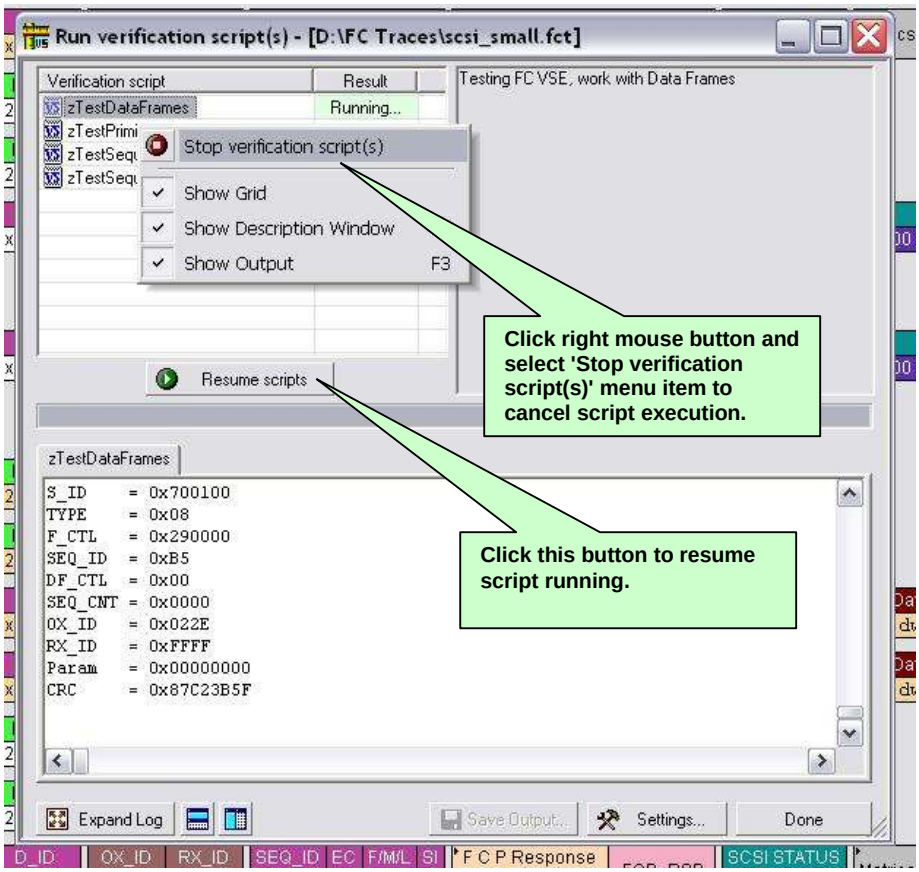
Example

```
…
If( Something_Interesting() )
{
  GotoEvent();   # Jump to the trace view.
  Pause();       # Pause script execution.
}
…
```

Remark

This function works only for VS Engine controlled via a GUI. For VSEs controlled by COM/Automation clients, it does nothing.

When script execution is paused, the Run Verification Script window looks like:



22.5 GetPacketData()

This function returns packet's data in unscramble 8 bits format as a list of DWORDs. Each item of the list includes one DWORD of packet. It also returns 0xFFFFFFFF for XXXX.

Format : **GetPacketData();**

Example

```
ListOfData =  GetPacketData();

for(i=0;i<sizeof(ListOfData);i++)

{

  ReportText(FormatEx("[%d]=0x%x\t", i, ListOfData[i]));

}
```

22.6 FormatRawBytes(RawBytes)

To convert any raw data field to a string. It is a utility method in VSTools.inc file, which can be included and used.

Format : FormatRawBytes(RawBytes)

Parameters

RawBytes: any raw data value.
}

Return Values

The converted RawBytes to string in Hex format.

Example

```
%include "VSTools.inc" # must be included
....
LBA = "0x" + FormatEx("%s",FormatRawBytes(in.App_LogicalBlockAddress));
ReportText(LBA);
```

22.7 SetMaximumIterations

It is used to change maximum number of iterations that can be executed by “for” and “while” loops. Its default value is 200000 iterations.

Format : SetMaximumIterations(max_iteration)

Parameters

max_iteration: New maximum iteration number.

Example

SetMaximumIterations(1000000); // change it to 1 million

Chapter 23: The VSE Important Script Files

The VSE working files are located in the **..\SASVFScripts** or **..\SATAVFScripts** subfolder of the main *SAS Protocol Suite* or *SATA Protocol Suite* folder. The current version of VSE includes the following files:

File	Description
VSTools.inc	Main VSE file containing definitions of some generic and SAS/SATA-specific VSE script functions provided by Teledyne LeCroy (must be included in every script). **NOTE**: The files VS_constants.inc and VS_Primitives.inc are included.
VS_constants.inc	File containing definitions of some important generic and SAS/SATA-specific VSE global constants
VSTemplate.sasv_	Template file for new verification scripts.
VSUser_globals.inc	File of user global variable and constant definitions (In this file, it is useful to enter definitions of constants, variables, and functions to be used in many scripts you write.)
VS_OOB.inc	OOB definitions and utilities
VS_Primitives.inc	Primitives definitions and utilities
VS_SMPFrames.inc	SMP Frames definitions and utilities
VS_SSPFrames.inc	SSP Frames definitions and utilities
VS_STPFrames.inc	STP Frames definitions and utilities
VS_AddressFrames.inc	Address Frames definitions and utilities
VS_ATACommands.inc	ATA Commands definitions and utilities
VS_SCSICommands.inc	SCSI Commands definitions

23.1 Example Script Files

The VSE example files are located in the **..\SASVFScripts\Examples** or **..\SATAVFScripts\Examples** subfolder of the main *SAS Protocol Suite* or *SATA Protocol Suite* folder. The current version of VSE includes the following files:

File	Description
sample_ata_commands.sasvs	Sample script that explains how to handle ATA command events
sample_ssp_protocol.sasvs	Sample script that explains how to manage SSP Protocol event Information
SMP_DiscoverAndReporTracking.sasvs	Sample script that outputs SMP Functional specific information
SampleViewer_ALL.sasvs	Sample script to output all trace events
sample_event_count.sasvs	Sample script to show how count number of events for a category, sub-category and their corresponding fields
TimeLogicalFunc.sasvs	Sample script that shows how to work with Time Logical functions of VSE
ProtocolError.sasvs	Sample script to detect protocol errors

Appendix A – Primitive Codes

```
PRIM_CODE_ERROR
PRIM_CODE_AIP_NORMAL
PRIM_CODE_AIP_RESERVED_0
PRIM_CODE_AIP_RESERVED_1
PRIM_CODE_AIP_RESERVED_2
PRIM_CODE_AIP_RESERVED_WAITING_ON_PARTIAL
PRIM_CODE_AIP_WAITING_ON_CONNECTION
PRIM_CODE_AIP_WAITING_ON_DEVICE
PRIM_CODE_AIP_WAITING_ON_PARTIAL
PRIM_CODE_BROADCAST_CHANGE
PRIM_CODE_BROADCAST_SES
PRIM_CODE_BROADCAST_EXPANDER
PRIM_CODE_BROADCAST_ASYNC_EVENT
PRIM_CODE_BROADCAST_RESERVED_3
PRIM_CODE_BROADCAST_RESERVED_4
PRIM_CODE_BROADCAST_RESERVED_CHANGE_0
PRIM_CODE_BROADCAST_RESERVED_CHANGE_1
PRIM_CODE_CLOSE_CLEAR_AFFILIATION
PRIM_CODE_CLOSE_NORMAL
PRIM_CODE_CLOSE_RESERVED_0
PRIM_CODE_CLOSE_RESERVED_1
PRIM_CODE_HARD_RESET
PRIM_CODE_OOB_IDLE
PRIM_CODE_OPEN_REJECT_BAD_DESTINATION
PRIM_CODE_OPEN_REJECT_CONNECTION_RATE_NOT_SUPPORTED
PRIM_CODE_OPEN_REJECT_NO_DESTINATION
PRIM_CODE_OPEN_REJECT_PATHWAY_BLOCKED
PRIM_CODE_OPEN_REJECT_PROTOCOL_NOT_SUPPORTED
PRIM_CODE_OPEN_REJECT_ZONING_VIOLATION
PRIM_CODE_OPEN_REJECT_RESERVED_ABANDON_1
PRIM_CODE_OPEN_REJECT_RESERVED_ABANDON_2
PRIM_CODE_OPEN_REJECT_RESERVED_ABANDON_3
PRIM_CODE_OPEN_REJECT_RESERVED_CONTINUE_0
PRIM_CODE_OPEN_REJECT_RESERVED_CONTINUE_1
PRIM_CODE_OPEN_REJECT_RESERVED_INITIALIZE_0
PRIM_CODE_OPEN_REJECT_RESERVED_INITIALIZE_1
PRIM_CODE_OPEN_REJECT_RESERVED_STOP_0
PRIM_CODE_OPEN_REJECT_RESERVED_STOP_1
PRIM_CODE_OPEN_REJECT_RETRY
PRIM_CODE_OPEN_REJECT_STP_RESOURCES_BUSY
PRIM_CODE_OPEN_REJECT_WRONG_DESTINATION
PRIM_CODE_OPEN_ACCEPT
PRIM_CODE_ACK
PRIM_CODE_CREDIT_BLOCKED
PRIM_CODE_DONE_ACK_NAK_TIMEOUT
PRIM_CODE_DONE_CREDIT_TIMEOUT
PRIM_CODE_DONE_NORMAL
PRIM_CODE_DONE_RESERVED_0
PRIM_CODE_DONE_CLOSE
PRIM_CODE_DONE_RESERVED_TIMEOUT_0
PRIM_CODE_DONE_RESERVED_TIMEOUT_1
PRIM_CODE_NAK_CRC_ERROR
PRIM_CODE_NAK_RESERVED_0
PRIM_CODE_NAK_RESERVED_1
PRIM_CODE_NAK_RESERVED_2
```

```
PRIM_CODE_RRDY_NORMAL
PRIM_CODE_RRDY_RESERVED_0
PRIM_CODE_RRDY_CLOSE
PRIM_CODE_NOTIFY_ENABLE_SPINUP
PRIM_CODE_NOTIFY_POWER_FAILURE_EXPECTED
PRIM_CODE_NOTIFY_RESERVED_1
PRIM_CODE_NOTIFY_RESERVED_2
PRIM_CODE_BREAK
PRIM_CODE_BREAK_REPLY
PRIM_CODE_TRAIN
PRIM_CODE_TRAIN_DONE
PRIM_CODE_MUX_0
PRIM_CODE_MUX_1
PRIM_CODE_PS_REQ_PARTIAL
PRIM_CODE_PS_REQ_SLUMBER
PRIM_CODE_PS_ACK
PRIM_CODE_PS_NAK
PRIM_CODE_SATA_SOF
PRIM_CODE_SATA_EOF
PRIM_CODE_SATA_CONT
PRIM_CODE_SATA_DMAT
PRIM_CODE_SATA_HOLD
PRIM_CODE_SATA_HOLDA
PRIM_CODE_SATA_PMACK
PRIM_CODE_SATA_PMNAK
PRIM_CODE_SATA_PMREQ_P
PRIM_CODE_SATA_PMREQ_S
PRIM_CODE_SATA_R_ERR
PRIM_CODE_SATA_R_IP
PRIM_CODE_SATA_R_OK
PRIM_CODE_SATA_R_RDY
PRIM_CODE_SATA_SYNC
PRIM_CODE_SATA_WTRM
PRIM_CODE_SATA_X_RDY
PRIM_CODE_SATA_ERROR
PRIM_CODE_D10_2
PRIM_CODE_EXTEND_CONNECTION_NORMAL
PRIM_CODE_EXTEND_CONNECTION_CLOSE
PRIM_CODE_PWR_ACK
PRIM_CODE_PWR_DONE
PRIM_CODE_PWR_GRANT
PRIM_CODE_PWR_REQ
```

Appendix B – ATA Commands

See the include file “VS_ATA_Commands.inc“ in the **..\SASVFScripts** or **..\SATAVFScripts** subfolder of the main *SAS Protocol Suite* or *SATA Protocol Suite* folder for defined constants for use in scripts.

Note: This file needs to be included in the VSE Script to be able to use the defined constants.
Note: The SCSI commands listed below are defined with the following prefix ATA_CMD_ (e.g. CFA_ERASE_SECTORS would be ATA_CMD_ CFA_ERASE_SECTORS)

CFA_ERASE_SECTORS	= 0xC0
CFA_REQUEST_EXTENDED_ERROR_CODE	= 0x03
CFA_TRANSLATE_SECTOR	= 0x87
CFA_WRITE_MULTIPLE_WITHOUT_ERASE	= 0xcd
CFA_WRITE_SECTORS_WITHOUT_ERASE	= 0x38
CHECK_MEDIA_CARD_TYPE	= 0xD1
CHECK_POWER_MODE	= 0xE5
DEVICE_CONFIGURATION	= 0xB1
DEVICE_RESET	= 0x08
DOWNLOAD_MICROCODE	= 0x92
EXECUTE_DEVICE_DIAGNOSTIC	= 0x90
FLUSH_CACHE	= 0xE7
FLUSH_CACHE_EXT	= 0xEA
GET_MEDIA_STATUS	= 0xDA
IDENTIFY_DEVICE	= 0xEC
IDENTIFY_PACKET_DEVICE	= 0xA1
IDLE	= 0xE3
IDLE_IMMEDIATE	= 0xE1
MEDIA_EJECT	= 0xED
MEDIA_LOCK	= 0xDE
MEDIA_UNLOCK	= 0xDF
NOP	= 0x00
PACKET	= 0xA0
READ_BUFFER	= 0xE4
READ_DMA	= 0xC8
READ_DMA_EXT	= 0x25
READ_DMA_QUEUED	= 0xC7
READ_DMA_QUEUED_EXT	= 0x26
READ_LOG_EXT	= 0x2F
READ_MULTIPLE	= 0xC4
READ_MULTIPLE_EXT	= 0x29
READ_NATIVE_MAX_ADDRESS	= 0xF8
READ_NATIVE_MAX_ADDRESS_EXT	= 0x27
READ_SECTORS	= 0x20
READ_SECTORS_WITHOUT_RETRY	= 0x21
READ_SECTORS_EXT	= 0x24
READ_VERIFY_SECTORS	= 0x40
READ_VERIFY_SECTORS_WITHOUT_RETRY	= 0x41
READ_VERIFY_SECTORS_EXT	= 0x42
SECURITY_DISABLE_PASSWORD	= 0xF6
SECURITY_ERASE_PREPARE	= 0xF3
SECURITY_ERASE_UNIT	= 0xF4
SECURITY_FREEZE_LOCK	= 0xF5
SECURITY_SET_PASSWORD	= 0xF1
SECURITY_UNLOCK	= 0xF2
SEEK	= 0x70

SERVICE	= 0xA2
SET_FEATURES	= 0xEF
SET_MAX	= 0xF9
SET_MAX_ADDRESS_EXT	= 0x37
SET_MULTIPLE_MODE	= 0xC6
SLEEP	= 0xE6
SMART	= 0xB0
STANDBY	= 0xE2
STANDBY_IMMEDIATE	= 0xE0
WRITE_BUFFER	= 0xE8
WRITE_DMA	= 0xCA
WRITE_DMA_EXT	= 0x35
WRITE_DMA_QUEUED	= 0xCC
WRITE_DMA_QUEUED_EXT	= 0x36
WRITE_LOG_EXT	= 0x3F
WRITE_MULTIPLE	= 0xC5
WRITE_MULTIPLE_EXT	= 0x39
WRITE_SECTORS	= 0x30
WRITE_SECTORS_EXT	= 0x34
INITIALIZE_DEVICE_PARAMETERS	= 0x91
RECALIBRATE	= 0x10
READ_FPDMA_QUEUED	= 0x60
WRITE_FPDMA_QUEUED	= 0x61
READ_STREAM_DMA	= 0x2A
READ_STREAM_PIO	= 0x2B
WRITE_STREAM_DMA	= 0x3A
WRITE_STREAM_PIO	= 0x3B
WRITE_DMA_FUA_EXT	= 0x3D
WRITE_DMA_QUEUED_FUA_EXT	= 0x3E
WRITE_MULTIPLE_FUA_EXT	= 0xCE
CONFIGURE_STREAM	= 0x51
NCQ_QUEUE_MANAGEMENT	= 0x63
DATA_SET_MANAGEMENT	= 0x06
SOFT_RESET	= 0xFF
READ_LOG_DMA_EXT	= 0x47
TRUSTED_NON_DATA	= 0x5B
WRITE_LOG_DMA_EXT	= 0x57
WRITE_UNCORRECTABLE_EXT	= 0x45
WRITE_MULTIPLE_C3	= 0xC3
NV_CACHE	= 0xB6
READ_PORT_MULTIPLIER	= 0xE4
WRITE_PORT_MULTIPLIER	= 0xE8
TRUSTED_RECEIVE	= 0x5C
TRUSTED_RECEIVE_DMA	= 0x5D
TRUSTED_SEND	= 0x5E
TRUSTED_SEND_DMA	= 0x5F
DOWNLOAD MICROCODE DMA	= 0X93
READ BUFFER DMA	= 0xE9
WRITE BUFFER DMA	= 0xEB
RECEIVE FPDMA QUEUED	= 0x65
SEND FPDMA QUEUED	= 0x64
BLOCK ERASE EXT	= 0xB4
CRYPTO SCRAMBLE EXT	= 0xB4
OVERWRITE EXT	= 0xB4
REQUEST SENSE DATA EXT	= 0x0B
SANITIZE FREEZE LOCK EXT	= 0xB4
SANITIZE STATUS EXT	= 0xB4

Appendix C – SCSI Commands

See the include file “VS_SCSICommands.inc“ in the **..\SASVFScripts** or **..\SATAVFScripts** subfolder of the main *SAS Protocol Suite* or *SATA Protocol Suite* folder for defined constants for use in scripts.

Note: This file needs to be included in the VSE Script to be able to use the defined constants.
Note: The SCSI commands listed below are defined with the following prefix SCSI_CMD_ (e.g. SBC_FORMAT_UNIT would be SCSI_CMD_ SBC_FORMAT_UNIT)

```
//SBC
SBC_FORMAT_UNIT                 =   0x04
SBC_LOCK_UNLOCK_CACHE10         =   0x36
SBC_LOCK_UNLOCK_CACHE16         =   0x92
SBC_PRE_FETCH10                 =   0x34
SBC_PRE_FETCH16                 =   0x90
SBC_READ6                       =   0x08
SBC_READ10                      =   0x28
SBC_READ12                      =   0xA8
SBC_READ16                      =   0x88
SBC_READ_CAPACITY10             =   0x25
SBC_9E                          =   0x9E
SBC_READ_DEFECT_DATA10          =   0x37
SBC_READ_DEFECT_DATA12          =   0xB7
SBC_READ_LONG                   =   0x3E
SBC_REASSIGN_BLOCKS             =   0x07
SBC_REBUILD16                   =   0x81
SBC_7F                          =   0x7F
SBC_REGENERATE16                =   0x82
SBC_REZERO_UNIT                 =   0x01
SBC_SEEK6                       =   0x0B
SBC_SEEK10                      =   0x2B
SBC_SET_LIMITS10                =   0x33
SBC_SET_LIMITS12                =   0xB3
SBC_START_STOP_UNIT             =   0x1B
SBC_SYNCHRONIZE_CACHE10         =   0x35
SBC_SYNCHRONIZE_CACHE16         =   0x91
SBC_VERIFY10                    =   0x2F
SBC_VERIFY12                    =   0xAF
SBC_VERIFY16                    =   0x8F
SBC_WRITE6                      =   0x0A
SBC_WRITE10                     =   0x2A
SBC_WRITE12                     =   0xAA
SBC_WRITE16                     =   0x8A
SBC_WRITE_AND_VERIFY10          =   0x2E
SBC_WRITE_AND_VERIFY12          =   0xAE
SBC_WRITE_AND_VERIFY16          =   0x8E
SBC_WRITE_LONG                  =   0x3F
SBC_WRITE_SAME10                =   0x41
SBC_WRITE_SAME16                =   0x93
SBC_XDREAD10                    =   0x52
SBC_XDWRITE10                   =   0x50
SBC_XDWRITEREAD10               =   0x53
SBC_XDWRITEEXTENDED16           =   0x80
SBC_XPWRITE10                   =   0x51
SBC_ERASE10                     =   0x2C
```

SBC_ERASE12 = 0xAC
SBC_MEDIUM_SCAN = 0x38
SBC_READ_GENERATION = 0x29
SBC_READ_UPDATED_BLOCK = 0x2D
SBC_UPDATE_BLOCK = 0x3D
SBC_ORWRITE = 0x8B
SBC_PREVENT_ALLOW_MEDIUM_REMOVAL = 0x1E
SBC_WRITE_LONG16 = 0x9F
SBC_SKIP_READ = 0xE8
SBC_SKIP_WRITE = 0xEA
SBC_UNMAP = 0x42

//MMC
MMC_BLANK = 0xA1
MMC_CLOSE_TRACKSESSION = 0x5B
MMC_GET_CONFIGURATION = 0x46
MMC_GET_EVENT_STATUS_NOTIFICATION = 0x4A
MMC_GET_PERFORMANCE = 0xAC
MMC_LOAD_UNLOAD_MEDIUM = 0xA6
MMC_MECHANISM_STATUS = 0xBD
MMC_PAUSE_RESUME = 0x4B
MMC_PLAY_AUDIO10 = 0x45
MMC_PLAY_AUDIO12 = 0xA5
MMC_PLAY_AUDIO_MSF = 0x47
MMC_READ_BUFFER_CAPACITY = 0x5C
MMC_SET_STREAMING = 0xB6
MMC_SYNCHRONIZE_CACHE = 0x35
MMC_STOP_PLAYSCAN = 0x4E
MMC_SEND_CUE_SHEET = 0x5D
MMC_RESERVE_TRACK = 0x53
MMC_READ_CD = 0xBE
MMC_READ_CD_MSF = 0xB9
MMC_READ_DISC_INFORMATION = 0x51
MMC_READ_DVD_STRUCTURE = 0xAD
MMC_READ_FORMAT_CAPACITIES = 0x23
MMC_READ_TOC_PMA_ATIP = 0x43
MMC_READ_SUB_CHANNEL = 0x42
MMC_READ_TRACK_INFORMATION = 0x52
MMC_REPAIR_TRACK = 0x58
MMC_REPORT_KEY = 0xA4
MMC_SEND_KEY = 0xA3
MMC_SEND_DVD_STRUCTURE = 0xBF
MMC_SCAN = 0xBA
MMC_SEND_OPC_INFORMATION = 0x54
MMC_SET_CD_SPEED = 0xBB
MMC_SET_READ_AHEAD = 0xA7

//SSC
SSC_ERASE16 =0x93
SSC_READ16 =0x88
SSC_READ_REVERSE16 =0x81
SSC_VERIFY16 =0x8F
SSC_WRITE16 =0x8A
SSC_WRITE_FILEMARKS16 =0x80
SSC_ERASE6 =0x19
SSC_LOCATE10 =0x2B

SSC_READ6 =0x08
SSC_READ_REVERSE6 =0x0F
SSC_SPACE6 =0x11
SSC_VERIFY6 =0x13
SSC_WRITE6 =0x0A
SSC_WRITE_FILEMARKS6 =0x10
SSC_FORMAT_MEDIUM =0x04
SSC_LOAD_UNLOAD =0x1B
SSC_LOCATE16 =0x92
SSC_READ_BLOCK_LIMITS =0x05
SSC_READ_POSITION =0x34
SSC_RECOVER_BUFFERED_DATA =0x14
SSC_REPORT_DENSITY_SUPPORT =0x44
SSC_REWIND =0x01
SSC_SET_CAPACITY =0x0B
SSC_SPACE16 =0x91
SSC_REQUEST_BLOCK_ADDRESS = 0x02
SSC_SEEK_BLOCK = 0x0C

//SCC
SCC_SPARE_IN = 0x22
SCC_MAINTENANCE_IN = 0xA3
SCC_VOLUME_SET_IN = 0xBE
SCC_VOLUME_SET_OUT = 0xBF

//OSD
OSD_7F =0x7F

//ADC
ADC_9F = 0x9F
ADC_A9 = 0xA9

//SMC
SMC_EXCHANGE_MEDIUM = 0xA6
SMC_INITIALIZE_ELEMENT_STATUS = 0x07
SMC_INITIALIZE_ELEMENT_STATUS_WITH_RANGE = 0x37
SMC_MOVE_MEDIUM = 0xA5
SMC_MOVE_MEDIUM_ATTACHED = 0xA7
SMC_POSITION_TO_ELEMENT = 0x2B
SMC_READ_ATTRIBUTE = 0x8C
SMC_READ_ELEMENT_STATUS = 0xB8
SMC_READ_ELEMENT_STATUS_ATTACHED = 0xB4
SMC_REQUEST_VOLUME_ELEMENT_ADDRESS = 0xB5
SMC_SEND_VOLUME_TAG = 0xB6
SMC_WRITE_ATTRIBUTE = 0x8D
SMC_OPEN_CLOSE_IMPORT_EXPORT_ELEMENT = 0x1B
SMC_REPORT_VOLUME_TYPES_SUPPORTED = 0x44
SMC_REQUEST_DATA_TRANSFER_ELEMENT_INQUIRY = 0xA3
SMC_PREVENT_ALLOW_MEDIUM_REMOVAL = 0x1E

//SPC
SPC_ACCESS_CONTROL_IN = 0x86
SPC_ACCESS_CONTROL_OUT = 0x87
SPC_A4 = 0xA4
SPC_EXTENDED_COPY = 0x83

SPC_INQUIRY	= 0x12
SPC_LOGSELECT	= 0x4C
SPC_LOGSENSE	= 0x4D
SPC_MODESELECT6	= 0x15
SPC_MODESELECT10	= 0x55
SPC_MODESENSE6	= 0x1A
SPC_MODESENSE10	= 0x5A
SPC_MOVE_MEDIUM_ATTACHED	= 0xA7
SPC_PERSISTENT_RESERVE_IN	= 0x5E
SPC_PERSISTENT_RESERVE_OUT	= 0x5F
SPC_PREVENT_ALLOW_MEDIUM_REMOVAL	= 0x1E
SPC_READ_ATTRIBUTE	= 0x8C
SPC_READ_BUFFER	= 0x3C
SPC_READ_ELEMENT_STATUS_ATTACHED	= 0xB4
SPC_READ_MEDIA_SERIAL_NUMBER	= 0xAB
SPC_RECEIVE_COPY_RESULT	= 0x84
SPC_RECEIVE_DIAGNOSTIC_RESULTS	= 0x1C
SPC_A3	= 0xA3
SPC_REPORT_LUNS	= 0xA0
SPC_REQUEST_SENSE	= 0x03
SPC_SEND_DIAGNOSTIC	= 0x1D
SPC_TEST_UNIT_READY	= 0x00
SPC_WRITE_ATTRIBUTE	= 0x8D
SPC_WRITE_BUFFER	= 0x3B
SPC_RESERVE6	= 0x16
SPC_RELEASE6	= 0x17
SPC_RESERVE10	= 0x56
SPC_RELEASE10	= 0x57
SPC_SECURITY_PROTOCOL_IN	= 0xA2
SPC_SECURITY_PROTOCOL_OUT	= 0xB5
SPC_ATA_PASS_THROUGH_16	= 0x85
SPC_ATA_PASS_THROUGH_12	= 0xA1

Appendix D – SMP Commands

```
# Management Function definitions
const SMP_REPORT_GENERAL            = 0x00;
const SMP_REPORT_MANUFACTURER_INFO  = 0x01;
const SMP_DISCOVER                  = 0x10;
const SMP_REPORT_PHY_ERROR_LOG      = 0x11;
const SMP_REPORT_PHY_SATA           = 0x12;
const SMP_REPORT_ROUTE_INFO         = 0x13;
const SMP_CONFIGURE_ROUTE_INFO      = 0x90;
const SMP_PHY_CONTROL               = 0x91;
const SMP_PHY_TEST_FUNCTION         = 0x92;
const SMP_CONFIGURE_PHY_ZONE        = 0x93;
const SMP_CONFIGURE_ZONE_PERM       = 0x83;
const SMP_REPORT_ZONE_PERM          = 0x03;
const SMP_REPORT_ZONE_ROUTE_TABLE   = 0x14;

# Management Function Results
const SMP_FUNCTION_ACCEPTED         = 0x00;
const SMP_UNKNOWN_SMP_FUNCTION      = 0x01;
const SMP_FUNCTION_FAILED           = 0x02;
const SMP_INVALID_REQUEST_LENGTH    = 0x03;
const SMP_PHY_DOES_NOT_EXIST        = 0x10;
const SMP_INDEX_DOES_NOT_EXIST      = 0x11;
const SMP_PHY_DOES_NOT_SUPPORT_SATA = 0x12;
const SMP_UNKNOWN_PHY_OPERATION     = 0x13;
```

Appendix E – Limitation of For and While Loops

To prevent infinite loops in improperly written script code there is a maximum limitation of 40,000 iterations in the case of “For” and “While” loops.
Use the following nested loop get around this limitation:

```
for( i = 0; i <= 40000; i++ )
{
for( j = 0; j <= 40000; j++ )
{
k = i * 40000 + j; # k will iterate from 0 to 1600040000 thus overriding 40000
limitation
... # Do something with k
}
```

Appendix F:
How to Contact Teledyne LeCroy

Send e-mail…	psgsupport@teledynelecroy.com
Contact support…	teledynelecroy.com/support/contact
Visit Teledyne LeCroy's web site…	teledynelecroy.com
Tell Teledyne LeCroy…	Report a problem to Teledyne LeCroy Support via e-mail by selecting **Help > Tell Teledyne LeCroy** from the application toolbar. This requires that an e-mail client be installed and configured on the host machine.