



**Protocol Solutions Group**  
3385 Scott Blvd., Santa Clara, CA 95054  
Tel: +1/408.727.6600  
Fax: +1/408.727.6622

# Verification Script Engine

for

LeCroy *PETracer*<sup>TM</sup>

Reference Manual

**Manual Version 5.3**

**For *PETracer* Software Version 5.30**

October 2007

## 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.

LeCroy reserves the right to revise the information presented in this document without notice or penalty.

## Trademarks and Servicemarks

*LeCroy, CATC Trace, PETracer EML, PETracer ML, PETracer, PETrainer EML, PETrainer ML, and PETracer Automation* are trademarks of LeCroy.

*Microsoft* and *Windows* are registered trademarks of Microsoft Inc.

All other trademarks are property of their respective companies.

## Copyright

Copyright © 2007, LeCroy Corporation. All Rights Reserved.

This document may be printed and reproduced without additional permission, but all copies should contain this copyright notice.

## Version

This is version 5.3 the *PETracer* Verification Script Engine Reference Manual. This manual applies to *PETracer* software version 5.30 and higher.

# Table of Contents

<b>1 INTRODUCTION</b> .....	<b>1</b>
<b>2 VERIFICATION SCRIPT STRUCTURE</b> .....	<b>2</b>
<b>3 INTERACTION BETWEEN PETRACER AND A VERIFICATION SCRIPT</b> .....	<b>4</b>
<b>4 RUNNING VERIFICATION SCRIPTS FROM THE PETRACER</b> .....	<b>6</b>
4.1    RUNNING VERIFICATION SCRIPTS .....	8
4.2    VSE GUI SETTINGS .....	10
<b>5 VERIFICATION SCRIPT ENGINE INPUT CONTEXT MEMBERS</b> .....	<b>11</b>
5.1    TRACE EVENT-INDEPENDENT SET OF MEMBERS .....	11
5.2    TRACE EVENT-DEPENDENT SET OF MEMBERS .....	12
5.2.1    All packet/transaction-specific set of members .....	12
5.2.2    DLLP-specific set of members .....	13
5.2.3    TLP-specific set of members.....	14
5.2.4    Ordered Set specific set of members.....	16
5.2.5    Link Condition specific set of members .....	17
5.2.6    Link transaction-specific set of members .....	18
5.2.7    Split transaction-specific set of members .....	19
<b>6 VERIFICATION SCRIPT ENGINE OUTPUT CONTEXT MEMBERS</b> .....	<b>20</b>
<b>7 VERIFICATION SCRIPT ENGINE EVENTS</b> .....	<b>21</b>
7.1    PACKET LEVEL EVENTS .....	21
7.2    LINK TRANSACTION LEVEL EVENTS .....	21
7.3    SPLIT TRANSACTION LEVEL EVENTS.....	21
<b>8 SENDING FUNCTIONS</b> .....	<b>22</b>
8.1    SENDLEVEL() .....	22
8.2    SENDLEVELONLY().....	23
8.3    DONTSENDLEVEL() .....	24
8.4    SENDCHANNEL() .....	25
8.5    SENDCHANNELONLY().....	26
8.6    DONTSENDCHANNEL ().....	27
8.7    SENDALLCHANNELS().....	28
8.8    SENDTRACEEVENT () .....	29
8.9    DONTSENDTRACEEVENT() .....	30
8.10   SENDTRACEEVENTONLY().....	31
8.11   SENDALLTRACEEVENTS().....	32
8.12   SENDDLLPTYPE().....	33
8.13   FILTERDLLPTYPE().....	34
8.14   SENDTLPSTYPE().....	35
8.15   FILTERTLPSTYPE().....	36
8.16   SENDORDEREDSETTYPE() .....	37
8.17   FILTERORDEREDSETTYPE() .....	38
<b>9 TIMER FUNCTIONS</b> .....	<b>39</b>
9.1    VSE TIME OBJECT .....	39
9.2    SETTIMER().....	40
9.3    KILLTIMER() .....	41
9.4    GETTIMERTIME().....	42
<b>10 TIME CONSTRUCTION FUNCTIONS</b> .....	<b>43</b>
10.1   TIME().....	43

<b>11 TIME CALCULATION FUNCTIONS.....</b>	<b>44</b>
11.1  ADDTIME() .....	44
11.2  SUBTRACTTIME().....	45
11.3  MULTIMEBYINT().....	46
11.4  DIVTIMEBYINT().....	47
<b>12 TIME LOGICAL FUNCTIONS.....</b>	<b>48</b>
12.1  ISEQUALTIME() .....	48
12.2  ISLESSTIME() .....	49
12.3  ISGREATERTIME() .....	50
12.4  ISTIMEININTERVAL() .....	51
<b>13 TIME TEXT FUNCTIONS.....</b>	<b>52</b>
13.1  TIMETOTEXT() .....	52
<b>14 OUTPUT FUNCTIONS.....</b>	<b>53</b>
14.1  REPORTTEXT().....	53
14.2  ENABLEOUTPUT().....	54
14.3  DISABLEOUTPUT().....	55
<b>15 INFORMATION FUNCTIONS .....</b>	<b>56</b>
15.1  GETTRACEName() .....	56
15.2  GETSCRIPTNAME().....	57
15.3  GETAPPLICATIONFOLDER().....	58
15.4  GETCURRENTTIME().....	59
15.5  GETEVENTSEGNUMBER().....	60
15.6  GETTRIGGERPACKETNUMBER().....	61
<b>16 NAVIGATION FUNCTIONS.....</b>	<b>62</b>
16.1  GOTOEVENT().....	62
16.2  SETPMARKER().....	63
<b>17 FILE FUNCTIONS.....</b>	<b>64</b>
17.1  OPENFILE().....	64
17.2  CLOSEFILE() .....	65
17.3  WRITESTRING().....	66
17.4  SHOWINBROWSER() .....	67
<b>18 COM/AUTOMATION COMMUNICATION FUNCTIONS.....</b>	<b>68</b>
18.1  NOTIFYCLIENT() .....	68
<b>19 USER INPUT FUNCTIONS.....</b>	<b>69</b>
19.1  MSGBOX() .....	69
19.2  INPUTBOX().....	71
19.3  GETUSERDLGLIMIT().....	73
19.4  SETUSERDLGLIMIT() .....	74
<b>20 STRING MANIPULATION/FORMATING FUNCTIONS .....</b>	<b>75</b>
20.1  FORMATEx() .....	75
<b>21 MISCELLANEOUS FUNCTIONS .....</b>	<b>77</b>
21.1  SCRIPTFORDISPLAYONLY() .....	77
21.2  SLEEP() .....	78
21.3  CONVERTTOHTML() .....	79
21.4  PAUSE().....	80

---

<b>22 THE VSE IMPORTANT SCRIPT FILES .....</b>	<b>81</b>
22.1 <b>EXAMPLE SCRIPT FILES .....</b>	<b>81</b>
<b>HOW TO CONTACT LECROY .....</b>	<b>82</b>

# 1 Introduction

---

This document contains a description of the LeCroy Verification Script Engine (VSE), a new utility in the *PE Tracer™* software that allows users to perform custom analyses of PCI Express™ (PE) traffic, recorded using the new generation of PCI Express protocol analyzers.

VSE allows users to ask the *PE Tracer* application to send some desired “events” (currently defined as packets, link transactions or split transactions) from a PE trace to a verification script written using the CATC 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 PE packet header or link/split transaction, and to make complex timing calculations between different events in a pre-recorded trace. It also allows filtering-in or filtering-out of data with dynamically changing filtering conditions, porting of information to a special output window, saving of data to text files, and sending of data to COM clients connected to a *PE Tracer* application.

## 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 *PE Tracer™* application interacts with running scripts.

The main script file that contains the text of the verification script should have extension **.pevs**, and be located in the subfolder **..\Scripts\VFScripts** of the main *PE Tracer* folder. Some other files might be included in the main script file using directive **%include**. (see the LeCroy *PE Tracer* File Based Decoding user manual for details).

The following schema presents a common structure of a verification script (this is similar to the content of the script template [**VSTemplate.pev\_**] which is included with VSE):

```
# VS1.pevs
#
# 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 initialization 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
# are passed to the script.
#
# For details - 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 completed
# 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;
}

```



## 3 Interaction between *PETracer* and a verification script

---

When a user runs a script against a pre-recorded trace, the following sequence occurs:

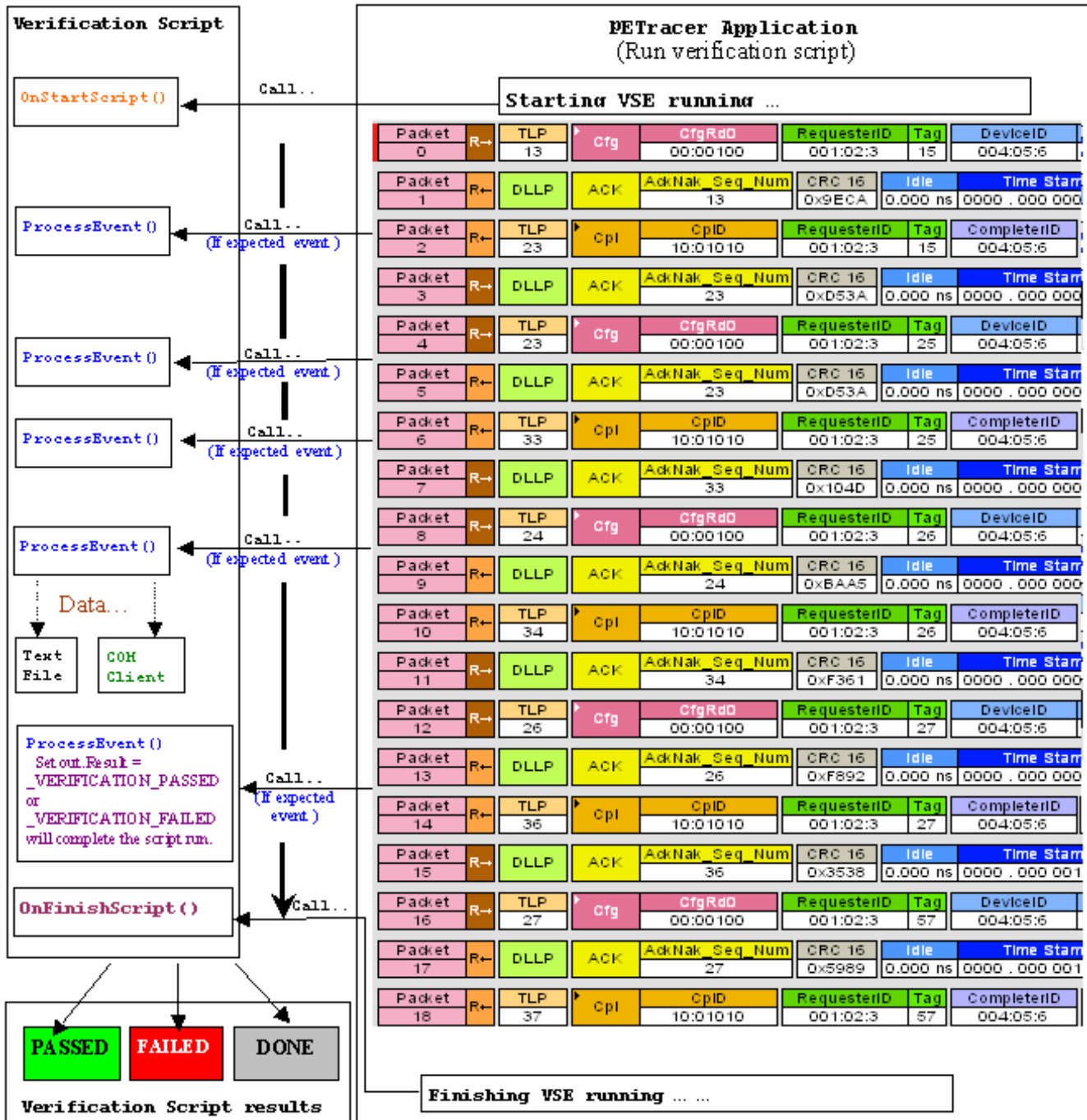
1. Prior to sending information to the script's main processing function **ProcessEvent()**, VSE looks for the function **OnStartScript()** and calls it if it is found. In this function, setup actions are defined, such as specifying the kind of trace events that should be passed to the script and setting up initial values for script-specific global variables.
2. Next, the 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 )
3. **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, tell 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 in this manner:  
**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. (There is a set of script functions allowing you to save extracted data in text files or send it to other applications, via COM/Automation interfaces.)

4. 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.

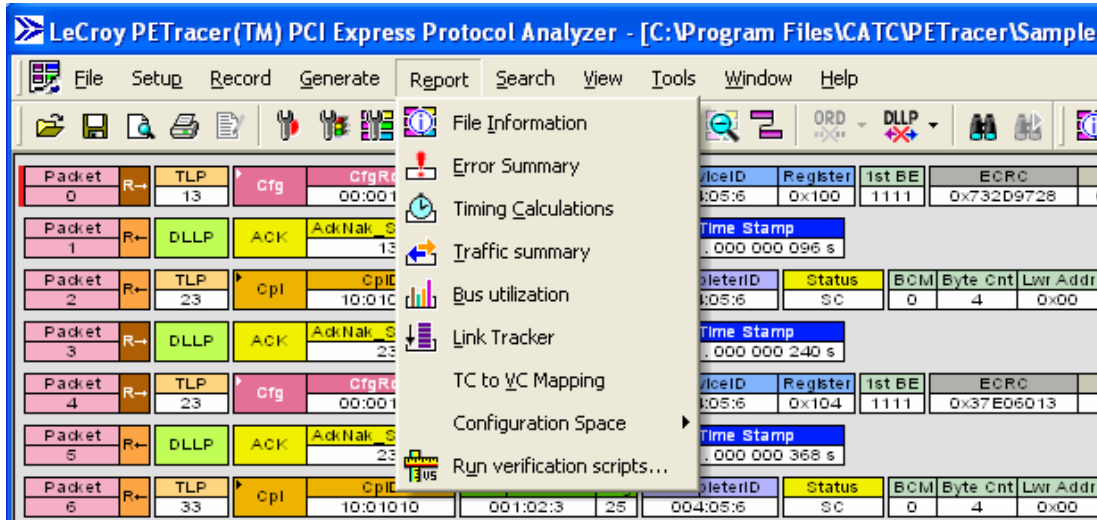
The following figure illustrates the interaction between the PETracer™ application and a running verification script:



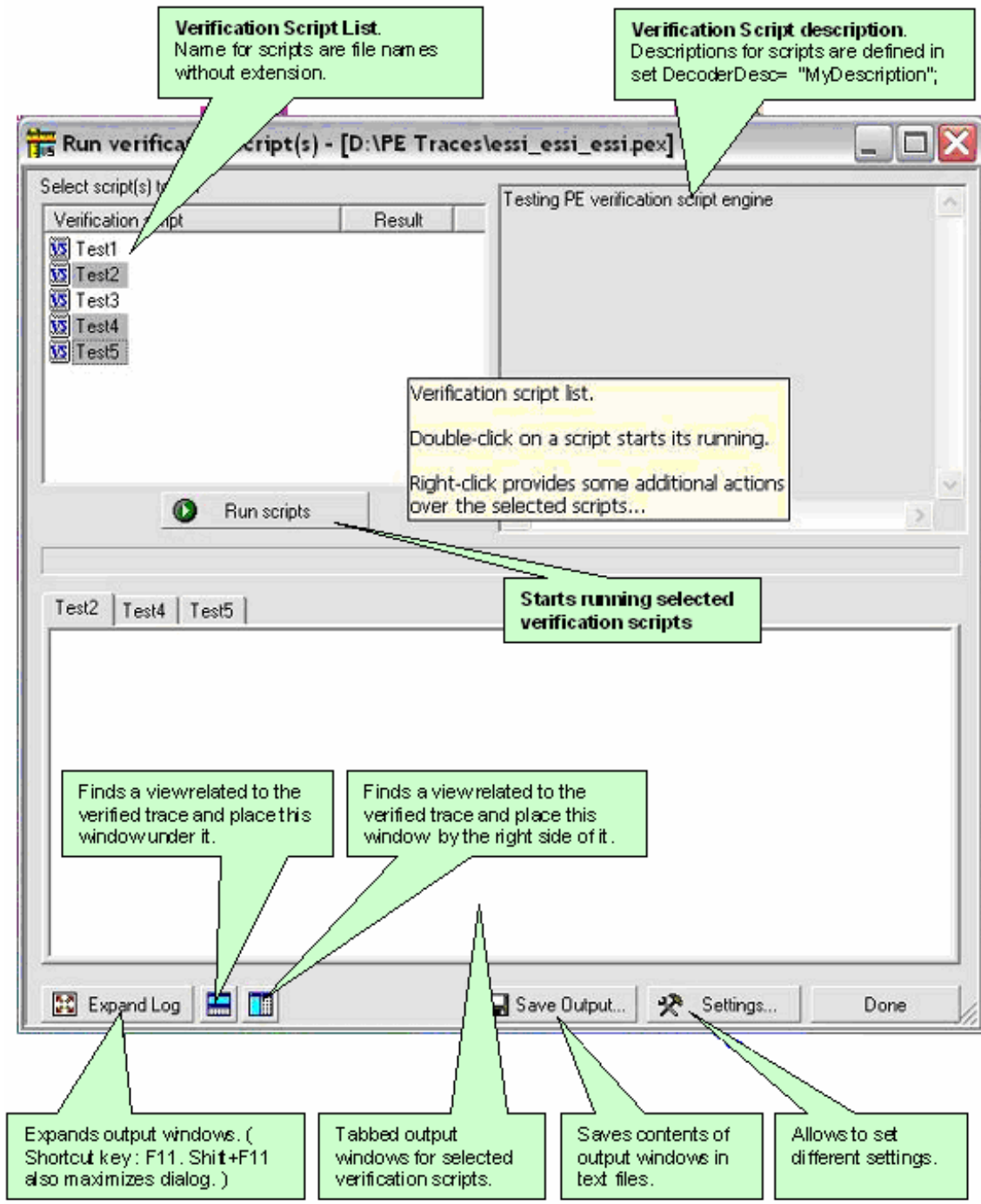
**Note:** The Verification script result "DONE" occurs when the script has been configured to extract and display some information about the trace, but not to display PASSED/FAILED results. To configure a script so that it only displays information – place a call somewhere in your script to the function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

# 4 Running verification scripts from the PETracer

In order to run a verification script over a trace - you need to open the PETracer™ main menu item **Report\Run verification scripts...** or push the icon on the main toolbar if it is not hidden.

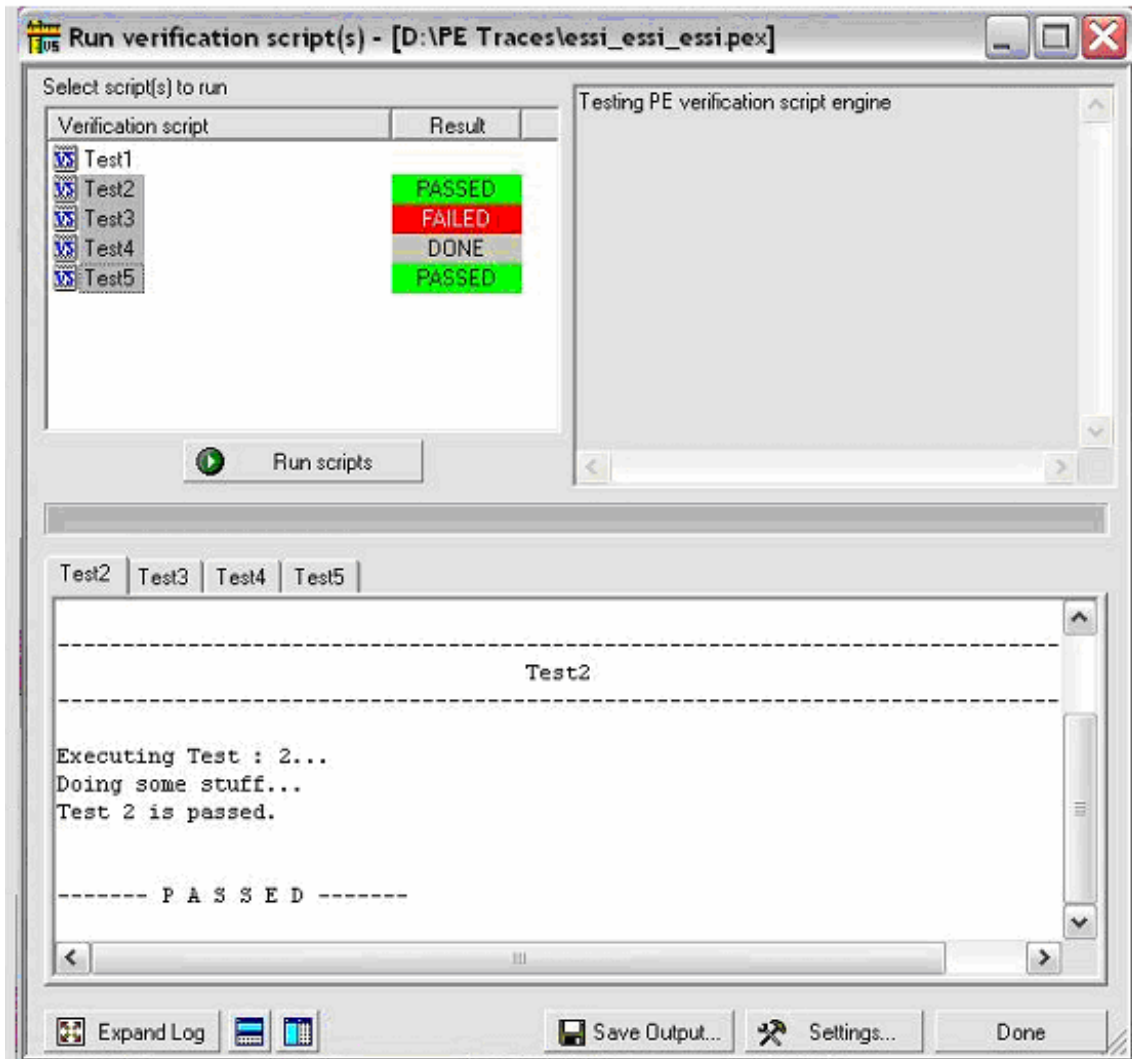


The special dialog opens displaying a list of verifications scripts. You can select one script to run, or several scripts from the list to run in parallel:

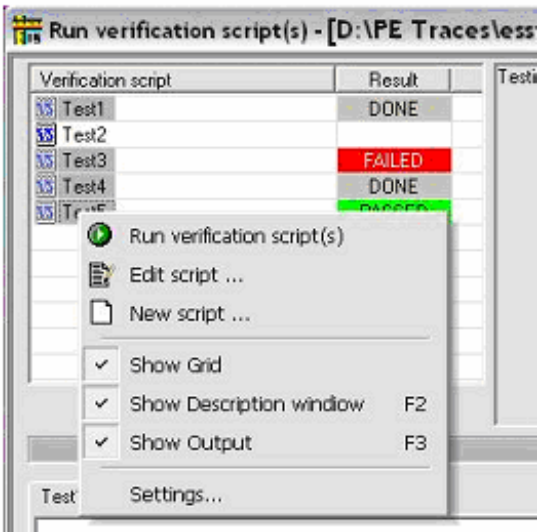


### 4.1 Running verification scripts

Push the button **Run scripts** after you selected the desired script(s) to run. VSE starts running the selected verification script(s), show script report information in the output windows, and present results of verifications in the script list:



Right-click in script list opens 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 whatever editor was specified in Editor settings.
- **New script:** Creates a new script file using the template specified in 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 which allows you to specify different settings for VSE.

## 4.2 VSE GUI Settings

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

The screenshot shows the 'Settings' dialog box with the following sections and options:

- Choose Editor application and editing settings:**
  - Notepad (by default)
  - Other...
  - Path to the editor: [Text Field] [Browse...]
  - Edit all selected scripts in one process
  - Open all included files
  - Launch editor application in full screen
  - Path to the template file for a new script: D:\Projects\PETracer\Debug\Scripts\VFScript [Browse...]
- Display settings:**
  - Show the full path for the trace file in dialog caption
  - Restore (don't maximize) dialog at start
  - Load last output from saved log files when possible
  - Activate dialog after script(s) stop running
  - Remember dialog layout
- Saving settings:**
  - Path to the folder where to save output log files: D:\Projects\PETracer\Debug [Browse...]
  - Save logs automatically after scripts stopped running

Callout boxes provide the following explanations:

- Callout 1:** This option (if set) allows editor applications supporting multi-document interface (MDI) to edit all script files related to the selected scripts in one application instance. Otherwise, a new application instance will be launched for each script file.
- Callout 2:** This option (if set) allows editor applications to edit all included files (extension : \*.inc) along with main verification script files (extension : \*.vse). Otherwise, only main verification script files will be opened for editing.
- Callout 3:** Launches editor application in full screen mode.
- Callout 4:** Full path to the file to be used as a template for a new script.
- Callout 5:** This setting (if set) specifies that the last saved output for selected scripts should be loaded into the output windows.
- Callout 6:** This setting (if set) brings Run VS dialog to foreground when scripts stopped running.
- Callout 7:** This setting (if set) forces the application to save output automatically when the script's stopped running.

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

## 5 Verification Script Engine Input Context Members

---

All verification scripts have input contexts – some special structures whose members are filled by the application and can be used inside of the scripts (for more details about input contexts – please 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:** Transaction level of the trace event (0 = frames, 1= sequences)

**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 the format: 2 sec 125 ns -> [2 , 125]). (See [9.1 VSE Time Object](#) for details)

**in.Channel:** Channel where the event occurred. (may be **\_CHANNEL\_1** (1) or **\_CHANNEL\_2** (2) indicating which direction of the PE 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)



## 5.2 Trace event-dependent set of members

This set of members is defined and can be used only for a specific events or after calling some functions filling out some of the variables:

### 5.2.1 All packet/transaction-specific set of members

Members of this set are valid for any event.

**in.Payload:** Bit source of the frame/sequence payload (you can extract any necessary information using 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.LinkWidth:** Link Width recorded for this packet. Possible values – 1, 2, 4, 8, 16 -- represent the number of lanes on the link.

**Error-related variables.** Used for passing all the detected packet error types to the script.

**in.HasErrors:** Indicates presence of any error type in the current packet. If this variable is set to 1, some of the errors described below are specified. If it is set to zero, no errors are present.

**in.ErrorDisparity:** If set to a non-zero value, indicates presence of Running Disparity error(s) in this packet

**in.ErrorSymbol:** If set to a non-zero value, indicates presence of Symbol (10-bit Code) error(s) in this packet

**in.ErrorDelimiter:** If set to a non-zero value, indicates presence of Delimiter error(s) in this packet

**in.ErrorEndBad:** If set to a non-zero value, indicates presence of an EDB symbol in this packet

**in.ErrorAlignment:** If set to a non-zero value, indicates presence of Alignment error(s) in this packet

**in.ErrorLength:** If set to a non-zero value, indicates presence of Bad Length error(s) in this TLP packet

**in.HasIdleErrors:** Indicates presence of Idle errors in the current packet. If set, one of the following is set, indicating the presensef of error(s) of this type between this packet and the previous packet on this direction of the link:

**in.IdleErrorDisparity:** If set to a non-zero value, indicates presence of Running Disparity error(s)

**in.IdleErrorSymbol:** If set to a non-zero value, indicates presence of Symbol (10-bit Code) error(s)

**in.IdleErrorSkip:** If set to a non-zero value, indicates presence of Skip error(s)

**in.IdleErrorData:** If set to a non-zero value, indicates presence of Logical Idle data pattern error(s)

**Note:** for CRC error variables, see the specific packet type variable sets below.

### 5.2.2 DLLP-specific set of members

Valid for data link layer packets only. Undefined for other events.

**in.DLLPType:** Contains the numeric encoding of the DLLP type. The following possible values are defined by VSE and the corresponding constants can be used by scripts:

DLLP_TYPE_ACK	= 0x0;
DLLP_TYPE_NAK	= 0x1;
DLLP_TYPE_INIT_FC1_P	= 0x4;
DLLP_TYPE_INIT_FC1_NP	= 0x5;
DLLP_TYPE_INIT_FC1_CPL	= 0x6;
DLLP_TYPE_INIT_FC2_P	= 0xC;
DLLP_TYPE_INIT_FC2_NP	= 0xD;
DLLP_TYPE_INIT_FC2_CPL	= 0xE;
DLLP_TYPE_UPDATE_FC_P	= 0x8;
DLLP_TYPE_UPDATE_FC_NP	= 0x9;
DLLP_TYPE_UPDATE_FC_CPL	= 0xA;
DLLP_TYPE_VENDOR	= 0x3;
DLLP_TYPE_PM_ENTER_L1	= 0x10;
DLLP_TYPE_PM_ENTER_L23	= 0x11;
DLLP_TYPE_PM_ACT_STATE_REQUEST_L1	= 0x13;
DLLP_TYPE_PM_REQUEST_ACK	= 0x14;
DLLP_TYPE_INVALID	= 0x7;

**in.AckNak\_SeqNum:** Field value (valid only for Ack and Nak DLLPs), indicating which TLPs are affected by the acknowledgement

**in.VC\_ID:** Virtual Channel ID (valid only for InitFC and UpdateFC DLLPs)

**in.HdrFC:** Credit value for headers of the type indicated by the DLLP type (valid only for InitFC and UpdateFC DLLPs)

**in.DataFC:** Credit value for payload data of the type indicated by the DLLP type (valid only for InitFC and UpdateFC DLLPs)

**in.VendorSpecific:** 3-byte vendor-defined value in a Vendor-specific DLLP

**in.BadCRC:** Set to 1 if the DLLP has bad 16-bit CRC, to 0 otherwise

### 5.2.3 TLP-specific set of members

Valid for TLPs only, undefined for other events.

#### All TLPs

**in.TLPType:** Contains the numeric encoding of the TLP type. The following possible values are defined by VSE and the corresponding constants can be used by scripts:

```
TLP_TYPE_ID_INVALID = 0;
TLP_TYPE_ID_MRD32   = 1;
TLP_TYPE_ID_MRDLK32 = 2;
TLP_TYPE_ID_MWR32   = 3;
TLP_TYPE_ID_MRD64   = 4;
TLP_TYPE_ID_MRDLK64 = 5;
TLP_TYPE_ID_MWR64   = 6;
TLP_TYPE_ID_IORD    = 7;
TLP_TYPE_ID_IOWR    = 8;
TLP_TYPE_ID_CFGRD_0 = 9;
TLP_TYPE_ID_CFGWR_0 = 10;
TLP_TYPE_ID_CFGRD_1 = 11;
TLP_TYPE_ID_CFGWR_1 = 12;
TLP_TYPE_ID_MSG     = 13;
TLP_TYPE_ID_MSGD    = 14;
TLP_TYPE_ID_MSGAS   = 15;
TLP_TYPE_ID_MSGASD  = 16;
TLP_TYPE_ID_CPL     = 17;
TLP_TYPE_ID_CPLD    = 18;
TLP_TYPE_ID_CPLLK   = 19;
TLP_TYPE_ID_CPLDLK  = 20;
```

**in.BadLCRC:** Set to 1 if the TLP has bad LCRC, to 0 otherwise

**in.BadECRC:** Set to 1 if the TLP has bad ECRC (when it should be present), to 0 otherwise

#### Field values for all TLP types:

**in.Type:** Type of TLP field value

**in.Fmt:** Format of TLP field value

**in.PSN:** Packet Sequence Number for this TLP as set by the Data Link Layer

**in.RequesterId:** Requester ID value (Bus, Device and Function Number fields combined)

**in.Tag:** Tag field value

**in.TC:** Traffic Class field value

**in.Length:** Length field value

**in.Snoop:** Snoop attribute bit value

**in.Ordering:** Ordering attribute bit value

**in.TD:** TLP Digest bit value

**in.EP:** Poisoned TLP bit value

**in.LCRC:** LCRC value as set by the Data Link Layer

**in.ECRC:** ECRC value (optional)

**Field values dependant upon TLP type:**

**in.FirstDwBe:** Byte Enable bits for the first DW of the payload (all TLPs except Completions and Messages)

**in.LastDwBe:** Byte Enable bits for the last DW of the payload (all TLPs except Completions and Messages)

**in.Address:** 32-bit Address value for IO, Configuration, and Mem-32 requests

**in.AddressLo:** Low 32 bits of the Address for Mem-64 requests and Messages routed by address

**in.AddressHi:** High 32 bits of the Address for Mem-64 requests and Messages routed by address

**in.Deviceld:** Requester ID value (Bus, Device and Function Number fields combined) for Configuration requests and Messages routed by ID

**in.Register:** Register address (Register Number and Extended Register Number combined) for Configuration requests

**For Completion TLPs only:**

**in.CompleterId:** Completer ID value (Bus, Device and Function Number fields combined)

**in.ComplStatus:** Completion Status field value

**in.BCM:** Byte Count Modified bit value

**in.ByteCount:** Remaining Byte Count field value

**in.LowerAddr:** Lower Address for starting byte of completion field value

**For Message TLPs only:**

**in.MessageCode:** Message Code field value

**in.MessageRoute:** Message Routing field value (from the TLP Type field)

**For Configuration Write Requests and Read Completions:**

**in.RegisterData:** 32-bit value written to or read from a configuration register (for convenience of processing the configuration requests, as it also can be obtained from the Payload)

#### 5.2.4 Ordered Set specific set of members

**in.OrderedSetType:** Contains the numeric encoding of the Ordered Set type. The following possible values are defined by VSE and the corresponding constants can be used by scripts:

```
ORDSET_TYPE_TS1      = 0x02;  
ORDSET_TYPE_TS2      = 0x03;  
ORDSET_TYPE_FTS      = 0x04;  
ORDSET_TYPE_IDLE_SET = 0x05;  
ORDSET_TYPE_SKIP     = 0x07;  
ORDSET_TYPE_PATN     = 0x08;
```

For Training Sequences (TS1 and TS2), the following variables of the list type exist in the input context (the lists are arrays of integers with dimensions equal to the Link Width for the Training Sequence packet).

**in.TS\_LinkNumberList:** Contains the Link Number parameter values for all lanes

**in.TS\_LaneNumberList:** Contains the Lane Number parameter values for all lanes

**in.TS\_N\_FTSList:** Contains the N\_FTS parameter values for all lanes

**in.TS\_TrainingControlList:** Contains the Training Control bitmap parameter values for all lanes

**Note:** For Link Number and Lane Number values the special value of 0x1FF is used to indicate the PAD symbol. Please refer to the **examp\_ordered\_sets.pevs** sample script for an example of how to process ordered Sets and Training Sequences, in particular.

### 5.2.5 Link Condition specific set of members

**in.LinkConditionType:** Contains the numeric encoding of the Link Condition type. The following possible values are defined by VSE and the corresponding constants can be used by scripts:

LINK_CONDITION_LINK_UP	= 1; - "Link Up" Link Condition event
LINK_CONDITION_LINK_DOWN	= 2; - "Link Down" Link Condition event
LINK_CONDITION_SKEW	= 3; - "Deskewing" Link Condition event
LINK_CONDITION_ELECTR_IDLE	= 4; - "Electrical Idle" Link Condition event

## 5.2.6 Link transaction-specific set of members

Valid for Link transactions only, undefined for other events.

All the TLP-specific values are present in the input context for Link transactions, depending upon the type of TLP for this Link transaction. In addition to that, the following value exists:

**in.TransactionStatus**: Status for this Link transaction. Can be one of three values: Implicitly Acknowledged, Explicitly Acknowledged, or Incomplete (Link Layer error). See file **VS\_constants.inc** for encodings.

### Metric values

The following values are defined in input context for Link Transactions that are related to Unit Metrics. To learn more about Unit Metrics, please refer to PE *Tracer*<sup>TM</sup> Help.

**in.Metric\_NumOfPackets**: Metric presenting the total number of packets that compose this Link Transaction, an integer value

**in.Metric\_ResponseTime**: Metric presenting time it took to transmit this Link Transaction on the PE link, from the beginning of the first packet in the transaction to the end of the last packet in the transaction, a VSE time object value (see 9.1 VSE Time Object for details)

**in.Metric\_Throughput**: Metric presenting transaction payload divided by response time, expressed in **kilobytes** per second, an integer value

**in.Metric\_PayloadBytes**: Metric presenting number of data payload bytes this Link Transaction transferred, an integer value

**Notes**: For the incomplete Link Transactions only, the NumOfPackets metric is valid. In case of an incomplete Link Transaction, the ResponseTime metric value is set to `null`.

### 5.2.7 Split transaction-specific set of members

Valid for Split transactions only. Undefined for other events.

All the TLP-specific values **for the request TLP of the split transaction** are present in the input context for Link transactions, depending upon the type of TLP for this Link transaction. Also the common PayloadLength and Payload values reflect the total combined payload for the Split transaction. In addition to that, the following values exist:

**in.CompletionStatus:** Completion Status for this Split transaction. From the last completion of the response.

#### Metric values

The following values are defined in input context for Split Transactions that are related to Unit Metrics. To learn more about Unit Metrics please refer to *PE Tracer Help*.

**in.Metric\_NumOfPackets:** Metric presenting the total number of packets that compose this Link Transaction, an integer value

**in.Metric\_ResponseTime:** Metric presenting time it took to transmit this Split Transaction on the PE link, from the beginning of the first packet in the transaction to the end of the last packet in the transaction, a VSE time object value (see [9.1 VSE Time Object](#) for details)

**in.Metric\_LatencyTime:** Metric presenting time measured from the end of the request transaction to the first completion transmitted in response to the request within this Split Transaction, a VSE time object value (see [9.1 VSE Time Object](#) for details)

**in.Metric\_Throughput:** Metric presenting transaction payload divided by response time, expressed in **kilobytes** per second, an integer value

**in.Metric\_PayloadBytes:** Metric presenting number of data payload bytes this Split Transaction transferred, an integer value.

**Notes:** For the incomplete Link Transactions only, the NumOfPackets metric is valid. In case of an incomplete Link Transaction the ResponseTime metric value is set to `null`.



## 6 Verification Script Engine Output Context Members

---

All verification scripts have output contexts – some special structures whose members are filled by the script and can be used inside of the application. (For more details about output contexts, please 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 the 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 stops running.

If you don't specify any of those values, the result of script execution is set as **\_VERIFICATION\_FAILED** at exit.

**Note:** If you don't care about the results of the script that's running, please call function [ScriptForDisplayOnly\(\)](#) one time before stopping the script. Then the results are **DONE**.

## 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 retrieving and displaying some contained information. The information about the type of event can be seen in **in.TraceEvent**. Please refer to the topic "Sending Functions" in this manual for details about how to specify transaction levels and which events should be sent to verification scripts.

### 7.1 Packet level events

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

Types of Packets	in.TraceEvent
Data Link Layer Packets (DLLP)	_PKT_DLLP
Transaction Layer Packets (TLP)	_PKT_TLP
Ordered Sets	_PKT_ORDERED_SET
Link Conditions	_PKT_LINK_CONDITION

### 7.2 Link Transaction level events

The table below describes the current list of Link Transaction events (transaction level: 1) and value of **in.TraceEvent**:

Types of Link Transactions	in.TraceEvent
Memory transactions	_LINK_MEMORY
IO transactions	_LINK_IO
Configuration transactions	_LINK_CONFIG
Message transactions	_LINK_MESSAGE
Completion transactions	_LINK_COMPLETION

### 7.3 Split Transaction level events

The table below describes the current list of Split Transaction events (transaction level: 2) and value of **in.TraceEvent**:

Types of Split Transactions	in.TraceEvent
Memory transactions	_SPLIT_MEMORY
IO transactions	_SPLIT_IO
Configuration transactions	_SPLIT_CONFIG

## 8 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.

### 8.1 SendLevel()

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

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

**Parameters:**

level	Can have one of following values:
	_PACKET      (value 0) Send Packet level events
	_LINK        (value 1) Send Link Transaction level events
	_SPLIT       (value 2) Send Split Transaction level events

**Note:** Only Packet and Link Transaction level events are available in release 4.6 of PE *Tracer*<sup>TM</sup> software.

**Example:**

```
...
SendLevel( _PACKET); # Send packet level events
```

**Remark:**

If no level was specified, events of Packet level are sent to the script by default.

## 8.2 SendLevelOnly()

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

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

### Parameters:

<code>level</code>	Can have one of following values:
<code>_PACKET</code>	(value 0) Send Packet level events
<code>_LINK</code>	(value 1) Send Link Transaction level events
<code>_SPLIT</code>	(value 2) Send Split Transaction level events

### Example:

```
...
SendLevelOnly( _PACKET ); # Send ONLY packet level events
```

## 8.3 DontSendLevel()

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

**Format:** `DontSendLevel( level )`

### Parameters:

<code>level</code>	Can have one of following values:
<code>_PACKET</code>	(value 0) Send Packet level events
<code>_LINK</code>	(value 1) Send Link Transaction level events
<code>_SPLIT</code>	(value 2) Send Split Transaction level events

### Example:

```
...  
DontSendLevel( _LINK); # DO NOT send link transaction level events
```

## 8.4 SendChannel()

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

**Format:** `SendChannel( channel )`

### Parameters:

<code>channel</code>	Can have one of following values: <code>_CHANNEL_1</code> (= 1) Send events from Upstream direction of the link (channel 1) <code>_CHANNEL_2</code> (= 2) Send events from Downstream direction of the link (channel 2)
----------------------	---

### Example:

```
...  
SendChannel(_CHANNEL_1); # Send events from Upstream direction of the link
```

## 8.5 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` Can have one of following values:  
\_CHANNEL\_1 (= 1) Send events from Upstream direction of the link (channel 1)  
\_CHANNEL\_2 (= 2) Send events from Downstream direction of the link (channel 2)

### Example:

```
...  
SendChannelOnly( _CHANNEL_1 ); # Send ONLY events from Upstream  
                               # direction of the link
```

## 8.6 DontSendChannel ()

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

**Format:** `DontSendChannel ( channel )`

### Parameters:

<code>channel</code>	Can have one of following values: _CHANNEL_1 (= 1) Send events from Upstream direction of the link (channel 1) _CHANNEL_2 (= 2) Send events from Downstream direction of the link (channel 2)
----------------------	---

### Example:

```
...
DontSendChannel ( _CHANNEL_1 ); # DO NOT send events from Upstream
                                # direction of the link
```



## 8.7 SendAllChannels()

This function specifies that events that have occurred on ALL channels should be sent to the script.

**Format:** `SendAllChannels ()`

**Example:**

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

## 8.8 SendTraceEvent ()

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

**Format:** `SendTraceEvent( event )`

### Parameters:

`event` Can have one of the following values:

#### Packet level events:

Event value	Description
<code>_PKT_DLLP</code>	Data Link Layer Packets (DLLP)
<code>_PKT_TLP</code>	Transaction Layer Packets (TLP)
<code>_PKT_ORDERED_SET</code>	Ordered Sets
<code>_PKT_LINK_CONDITION</code>	Link Conditions

#### Link Transaction level events:

Event value	Description
<code>_LINK_MEMORY</code>	Memory transactions
<code>_LINK_IO</code>	IO transactions
<code>_LINK_CONFIG</code>	Configuration transactions
<code>_LINK_MESSAGE</code>	Message transactions
<code>_LINK_COMPLETION</code>	Completion transactions

### Example:

```

...
SendTraceEvent( _PKT_TLP );
...

SendLevel( _LINK );
SendTraceEvent ( _LINK_MEMORY ); # Send memory Read and Write request
                                   # transactions to the script

```

## 8.9 DontSendTraceEvent()

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

**Format:** `DontSendTraceEvent ( event )`

### Parameters:

event            See **SendTraceEvent()** for all possible values.

### Example:

```
...
SendLevel( _LINK );           # Send Link Transaction level events
SendTraceEvent ( _LINK_CONFIG ); # Send Configuration transactions
SendTraceEvent ( _LINK_COMPLETION ); # Send Completion transactions
SendTraceEvent ( _LINK_MESSAGE ); # Send Message transactions
...
if( SomeCondition )
{
DontSendTraceEvent ( _LINK_CONFIG ); # Don't send Cfg Request transactions
DontSendTraceEvent ( _LINK_COMPLETION ); # Don't send Completion transactions

# Only Message transactions are sent.
}
```

## 8.10 SendTraceEventOnly()

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

**Format:** `SendTraceEventOnly( event )`

### Parameters:

event            See **SendTraceEvent()** for all possible values.

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

### Example:

```
...
SendLevel( _LINK );           # Send Link Transaction level events
SendTraceEvent ( _LINK_CONFIG );   # Send Configuration transactions
SendTraceEvent ( _LINK_COMPLETION ); # Send Completion transactions
SendTraceEvent ( _LINK_MESSAGE );  # Send Message transactions
...
if( SomeCondition )
{
    SendTraceEventOnly ( _LINK_MEMORY );
    # Only Memory read/write request transactions are sent.
}
```

## 8.11 SendAllTraceEvents()

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

**Format:** `SendAllTraceEvents ()`

### Example:

```
...
SendLevel( _PACKET ); # Send packet level events
SendAllTraceEvents ( ); # All TLP, DLLP and Ordered Set packets
                        # are sent to the script
```

## 8.12 SendDllpType()

This function specifies more precise tuning (filtering in) for sending DLLP packets to the script.

**Format:** `SendDllpType( dllp_type )`

### Parameters:

`dllp_type` Encoding of the DLLP type. This parameter may be one of the following values:

**DLLP** type values:

Constant	DLLP type
<code>_DLLP_TYPE_ACK</code>	Ack
<code>_DLLP_TYPE_NAK</code>	Nak
<code>_DLLP_TYPE_INIT_FC1_P</code>	InitFC1-P
<code>_DLLP_TYPE_INIT_FC1_NP</code>	InitFC1-NP
<code>_DLLP_TYPE_INIT_FC1_CPL</code>	InitFC1-Cpl
<code>_DLLP_TYPE_INIT_FC2_P</code>	InitFC2-P
<code>_DLLP_TYPE_INIT_FC2_NP</code>	InitFC2-NP
<code>_DLLP_TYPE_INIT_FC2_CPL</code>	InitFC2-Cpl
<code>_DLLP_TYPE_UPDATE_FC_P</code>	UpdateFC-P
<code>_DLLP_TYPE_UPDATE_FC_NP</code>	UpdateFC-NP
<code>_DLLP_TYPE_UPDATE_FC_CPL</code>	UpdateFC-Cpl
<code>_DLLP_TYPE_PM</code>	All Power Management DLLP types
<code>_DLLP_TYPE_INVALID</code>	Invalid DLLP types
<code>_DLLP_TYPE_INIT_FC</code>	All InitFC DLLP types
<code>_DLLP_TYPE_UPDATE_FC</code>	All UpdateFC DLLP types
<code>_ANY_TYPE</code>	All possible DLLP types

### Example:

```
SendDllpType(_DLLP_TYPE_ACK );      # Send Ack DLLPs to the script
...
SendDllpType(_DLLP_TYPE_UPDATE_FC); # Send all UpdateFC DLLPs
```

## 8.13 FilterDllpType()

This function specifies more precise tuning (filtering out) for sending DLLP packets to the script.

**Format:** `FilterDllpType( dllp_type )`

### Parameters:

`dllp_type`      Encoding of the DLLP type.  
This parameter may be one of the values defined for the **SendDllpType()** function.

### Example:

```
SendDllpType(_DLLP_TYPE_INIT_FC);            # Send all InitFC DLLPs to the script
FilterDllpType(_DLLP_TYPE_INIT_FC1_CPL); # Don't send InitFCs for Completions
FilterDllpType(_DLLP_TYPE_INIT_FC2_CPL);

# Only InitFC DLLPs for Posted and Non-posted requests are sent to the script
```

## 8.14 SendTlpType()

This function specifies more precise tuning (filtering in) for sending TLP packets to the script.

**Format:** `SendDtlpType( tlp_type )`

### Parameters:

`tlp_type` Encoding of the TLP type. This parameter may be one of the following values:

TLP type values:

Constant	TLP type
<code>_TLP_TYPE_INVALID</code>	Invalid TLP types
<code>_TLP_TYPE_MRDLK32</code>	Memory Read Request, 32-bit address format
<code>_TLP_TYPE_MRD32</code>	Memory Read Request - Locked, 32-bit address format
<code>_TLP_TYPE_MWR32</code>	Memory Write Request, 32-bit address format
<code>_TLP_TYPE_MRDLK64</code>	Memory Read Request – Locked, 64-bit address format
<code>_TLP_TYPE_MRD64</code>	Memory Read Request, 64-bit address format
<code>_TLP_TYPE_MWR64</code>	Memory Write Request, 64-bit address format
<code>_TLP_TYPE_IORD</code>	I/O Read Request
<code>_TLP_TYPE_IOWR</code>	I/O Write Request
<code>_TLP_TYPE_CFGRD_0</code>	Configuration Read Type 0
<code>_TLP_TYPE_CFGWR_0</code>	Configuration Write Type 0
<code>_TLP_TYPE_CFGRD_1</code>	Configuration Read Type 1
<code>_TLP_TYPE_CFGWR_1</code>	Configuration Write Type 1
<code>_TLP_TYPE_MSG</code>	Message Request
<code>_TLP_TYPE_MSGD</code>	Message Request with Data payload
<code>_TLP_TYPE_MSGAS</code>	Message for Advanced Switching
<code>_TLP_TYPE_MSGASD</code>	Message for Advanced Switching with Data
<code>_TLP_TYPE_CPL</code>	Completion
<code>_TLP_TYPE_CPLD</code>	Completion with Data
<code>_TLP_TYPE_CPLLK</code>	Completion for Locked Memory Read
<code>_TLP_TYPE_CPLDLK</code>	Completion for Locked Memory Read with Data
<code>_TLP_TYPE_MEMORY</code>	All Memory Request TLP types
<code>_TLP_TYPE_IO</code>	All I/O Request TLP types
<code>_TLP_TYPE_CONFIG</code>	All Configuration Request TLP types
<code>_TLP_TYPE_MESSAGE</code>	All Message Request TLP types
<code>_TLP_TYPE_COMPLETION</code>	All Completion TLP types
<code>_ANY_TYPE</code>	All possible TLP types

### Example:

```
SendTlpType( _TLP_TYPE_ID_MSG ); # Send Message Request TLPs to the script
...
SendTlpType( _TLP_TYPE_MEMORY ); # Send all Memory Request TLPs to the script
```



## 8.15 FilterTlpType()

This function specifies more precise tuning (filtering out) for sending TLP packets to the script.

**Format:** `FilterTlpType( tlp_type )`

### Parameters:

<code>tlp_type</code>	Encoding of the TLP type. This parameter may be one of the values defined for the <b>SendTlpType()</b> function.
-----------------------	---

### Example:

```
SendTlpType(_TLP_TYPE_CONFIG); # Send all Configuration Request TLPs
                                # to the script
FilterTlpType(_TLP_TYPE_ID_CFGRD_1); # Don't send Type 1 requests
FilterTlpType(_TLP_TYPE_ID_CFGWR_1);

# Only Type 0 Configuration Request TLPs are sent to the script
```

## 8.16 SendOrderedSetType()

This function specifies more precise tuning (filtering in) for sending Ordered Set packets to the script.

**Format:** `SendOrderedSetType( set_type )`

### Parameters:

`set_type` Encoding of the Ordered Set type. This parameter may be one of the following values:

### Ordered Set type values:

Constant	DLLP type
<code>_ORDSET_TYPE_TS1</code>	Training Sequence Type 1
<code>_ORDSET_TYPE_TS2</code>	Training Sequence Type 2
<code>_ORDSET_TYPE_FTS</code>	Fast Training Sequence
<code>_ORDSET_TYPE_IDLE_SET</code>	Idle Set
<code>_ORDSET_TYPE_IDLE_GLC</code>	Idle Glc Set
<code>_ORDSET_TYPE_SKIP</code>	Skip
<code>_ORDSET_TYPE_PATN</code>	Pattern
<code>_ANY_TYPE</code>	All possible Ordered Set types

### Example:

```
SendOrderedSetType(_ORDSET_TYPE_FTS); # Send Fast Training Sequences
                                        # to the script
...
```

## 8.17 FilterOrderedSetType()

This function specifies more precise tuning (filtering out) for sending Ordered Set packets to the script.

**Format:** `FilterOrderedSetType( set_type )`

### Parameters:

<code>set_type</code>	Encoding of the Ordered Set type. This parameter may be one of the values defined for the <b>SendOrderedSetType()</b> function.
-----------------------	--

### Example:

```
FilterOrderedSetType(_ORDSET_TYPE_SKIP); # Don't send Skip packets.
```

## 9 Timer Functions

---

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

### 9.1 VSE Time Object

A VSE time object is a special object that presents time intervals in verification scripts. From point of view of the **CSL**, the verification script time object is a “list”-object of two elements. (Please see the *CSL Manual* for more details about CSL types.)

[\[seconds, nanoseconds\]](#)

**Note:** The best way to construct a VSE time object is to use the **Time()** function (see below).

## 9.2 SetTimer()

Starts the timing calculation from the event where this function was called.

**Format:** `SendTimer( timer_id = 0)`

### Parameters:

timer\_id      Unique timer identifier

### Example:

```
SetTimer();    # 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 for the same timer ID, it resets the timer and starts timing calculations again from the point where it was called.

## 9.3 KillTimer()

Stops timing calculation for a specific timer and frees related resources.

**Format:** `KillTimer( timer_id = 0)`

### Parameters:

timer\_id      Unique timer identifier

### Example:

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

## 9.4 GetTimerTime()

Retrieve the timing interval from the specific timer.

**Format:** `GetTimerTime ( timer_id = 0 )`

### Parameters:

timer\_id      Unique timer identifier

### Return values:

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.

# 10 Time Construction Functions

---

This group of functions are used to construct 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]**  
Second function returns **[seconds, nanoseconds]**

### Parameters:

<code>nanoseconds</code>	Number of nanoseconds in specified time
<code>seconds</code>	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`.



# 11 Time Calculation Functions

---

This group of functions covers VSE capability to work with “time” – VSE time objects.

## 11.1 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:

<code>time_1</code>	VSE time object representing the first time interval
<code>time_2</code>	VSE time object representing the second time interval

### Example:

```
t1 = Time(100);  
t2 = Time(2, 200);  
t3 = AddTime( t1, t2 ) # Returns VSE time object = 2 sec 300 ns.
```

## 11.2 SubtractTime()

Subtract 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:

<code>time_1</code>	VSE time object representing the first time interval
<code>time_2</code>	VSE time object representing the second time interval

### Example:

```
t1 = Time(100);  
t2 = Time(2, 200);  
t3 = SubtractTime ( t2, t1 ) # Returns VSE time object = 2 sec 100 ns.
```

## 11.3 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:

<code>time</code>	VSE time object
<code>mult</code>	multiplier, integer value

### Example:

```
t = Time(2, 200);  
t1 = MulTimeByInt ( t, 2 ) # Returns VSE time object = 4 sec 400 ns.
```

## 11.4 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:

<code>time</code>	VSE time object
<code>div</code>	divisor, integer value

### Example:

```
t = Time(2, 200);  
t1 = DivTimeByInt ( t, 2 ) # Returns VSE time object = 1 sec 100 ns.
```

## 12 Time Logical Functions

---

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

### 12.1 IsEqualTime()

Verifies that 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:**

<code>time_1</code>	VSE time object representing the first time interval
<code>time_2</code>	VSE time object representing the second time interval

**Example:**

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

## 12.2 IsLessTime()

Verifies that 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:

<code>time_1</code>	VSE time object representing the first time interval
<code>time_2</code>	VSE time object representing the second time interval

### Example:

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

## 12.3 IsGreaterTime()

Verifies that 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:

<code>time_1</code>	VSE time object representing the first time interval
<code>time_2</code>	VSE time object representing the second time interval

### Example:

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

## 12.4 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:

<code>time_1</code>	VSE time object representing the first time interval
<code>time_2</code>	VSE time object representing the second time interval

### Example:

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



## 13 Time Text Functions

---

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

### 13.1 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 below details for ReportText() function
```

## 14 Output Functions

---

This group of functions covers VSE capability to present information in the output window.

### 14.1 ReportText()

Outputs text in the output window related to the verification script.

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

**Parameters:**

text            Text variable, constant, or literal

**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 = 0xB BBB;
text = FormatEx( "x = 0x%04X, y = 0x%04X", x, y );
ReportText( "Text : " + text );
...
```

## 14.2 EnableOutput()

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

**Format:** `EnableOutput ( )`

**Example:**

```
EnableOutput ( );
```

## 14.3 DisableOutput()

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

**Format:** `DisableOutput ()`

**Example:**

```
DisableOutput ();
```

# 15 Information Functions

---

## 15.1 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 segment being processed.

**Format:** `GetTraceName( filepath_compatible )`

### Parameters:

<code>filepath_compatible</code>	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 Traces\Data.pex
# GetTraceName(1) returns - "D_Some Traces_Data.pex"
```

## 15.2 GetScriptName()

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

**Format:** `GetScriptName()`

**Example:**

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

## 15.3 GetApplicationFolder()

This function returns the full path of the folder where the PE *Tracer™* application was started.

**Format:** `GetApplicationFolder()`

**Example:**

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

## 15.4 GetCurrentTime()

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

**Format:** `GetCurrentTime()`

**Example:**

```
ReportText( GetCurrentTime() ); # Yields "February 10, 2004, 5:49 PM"
```



## 15.5 GetEventSegNumber()

In case 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 **\*.pem**) 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 have different segment numbers.

**Format:** `GetEventSegNumber()`

### Example:

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

## 15.6 GetTriggerPacketNumber()

This function returns the number of the trigger packet in the trace. 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 Navigation Functions

---

## 16.1 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:

<code>level</code>	Transaction level of the event to jump to (possible values: <code>_PACKET</code> , <code>_LINK</code> , <code>_SPLIT</code> )
<code>index</code>	Transaction index of the event to jump to
<code>segment</code>	Segment index of the event to jump to. If omitted, the current segment index is used.

### Remarks:

If no parameters were specified, the application jumps 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 (extension: `*.pem`). For regular traces it is ignored.

If wrong parameters were specified (like an index exceeding the maximum index for the specified transaction level), the function does nothing and an error message is 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 );
}
```

## 16.2 SetMarker()

This function sets a marker for some trace event.

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

### Parameters:

<code>marker_text</code>	Text of the marker
<code>level</code>	Transaction level of the event to jump to (possible values: <code>_PACKET</code> , <code>_LINK</code> , <code>_SPLIT</code> )
<code>index</code>	Transaction index of the event to jump to
<code>segment</code>	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 (extension: **\*.pem**). For regular traces it is ignored.

If wrong parameters were specified (like an index exceeding the maximum index for a specified transaction level), the function does nothing and an error message is 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 );
}

```

# 17 File Functions

---

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

## 17.1 OpenFile()

This function opens a file for writing.

**Format:** `OpenFile( file_path, append )`

### Parameters:

<code>file_path</code>	Full path to the file to open. (For '\ ' use '\\')
<code>append</code>	This parameter (if present and not equal to 0) specifies that VSE should append to the contents of the file. Otherwise, the contents of the file are overwritten.

### Return Values:

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

### 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
...
# Opens file, the following file operations append to contents of the file.
file_handle = OpenFile( GetApplicationFolder() + "Log.txt", _APPEND );
```

## 17.2 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
...
```

## 17.3 WriteString()

This function writes a text string to the file.

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

### Parameters:

<code>file_handle</code>	File "handle"
<code>text_string</code>	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
...
```

## 17.4 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 is launched. For instance, if Internet Explorer is registered to open files with extensions \*.htm and the file handle passed to **ShowInBrowser()** function belongs to a file with such an extension, this file is opened in the Internet Explorer.

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

### Parameters:

<code>file_handle</code>	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 );
...
```



# 18 COM/Automation Communication Functions

---

This group of functions covers VSE capabilities to communicate with COM/Automation clients connected to the *PETracer™* application. (Please refer to the *PETracer Automation Manual* for the details on how to connect to the *PETracer* application and VSE)

## 18.1 NotifyClient()

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

**Format:** `NotifyClient( param_list )`

### Parameters:

`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, "CHANNEL 2", "TLP",
                    TimeToText( in.Time ) ] );
}
...
# Here we sent 2 parameters to clients applications :
# 2 ( integer ),
# [ in.Index, in.Level, "CHANNEL 2", "TLP", TimeToText( in.Time ) ] ( list )
```

### Remark:

See an example of handling this notification by client applications and parsing code in the *PE Automation* document.

# 19 User Input Functions

---

## 19.1 MsgBox()

Displays a 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:

<code>prompt</code>	Required. String expression displayed as the message in the dialog box.
<code>type</code>	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 <b>_MB_OK</b> . (See the list of possible values in the table below)
<code>title</code>	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.

The **type** argument values are:

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

**Return Values:**

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

Constant	Description
_MB_OK	<b>OK</b> button was clicked.
_MB_CANCEL	<b>Cancel</b> button was clicked.
_MB_YES	<b>Yes</b> button was clicked.
_MB_NO	<b>No</b> button was clicked.
_MB_RETRY	<b>Retry</b> button was clicked.
_MB_IGNORE	<b>Ignore</b> button was clicked.
_MB_ABORT	<b>Abort</b> 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 *PE Tracer™* 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 doesn't show dialog boxes if a customizable limit was exceeded (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...
}

```

## 19.2 InputBox()

Displays a 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:

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

The **return\_type** argument values are:

Constant	Value	Description
<code>_IB_LIST</code>	0	CSL list object is returned ( by Default ).
<code>_IB_STRING</code>	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 as it is.

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"
```

Produces a CSL list object of 5 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 as 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 PE Tracer 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 doesn't show dialog boxes if a customizable limit was exceeded (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 );
}

```

## 19.3 GetUserDlgLimit()

This function returns the current limit of user dialogs allowed in the verification script. If the script reaches this limit, no user dialogs are shown and the script does 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
...
```

## 19.4 SetUserDlgLimit()

This function sets the current limit of user dialogs allowed in the verification script. If the script reaches this limit, no user dialogs are shown and script does not stop. 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
...
```

## 20 String Manipulation/Formatting Functions

---

### 20.1 FormatEx()

Write formatted data to a string. **FormatEx()** is used to control the way that arguments print out. 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 are used to define the conversion specifications.

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

#### Parameters:

<code>format_string</code>	Format-control string
<code>argument_list</code>	Optional list of arguments to fill in the format string

#### Return Values:

Formatted string .

Format conversion characters:

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 **0X** 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 expands 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 Miscellaneous Functions

---

### 21.1 ScriptForDisplayOnly()

Specifies that the script is designed for displaying information only and that its author doesn't care about verification script result. Such a script has a result of **DONE** after execution.

**Format:** `ScriptForDisplayOnly ()`

**Example:**

```
ScriptForDisplayOnly();
```

## 21.2 Sleep()

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.
```

## 21.3 ConvertToHTML()

This function replaces spaces with “&nbsp;” and carriage return symbols with “<br>” in a text string.

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

### Parameters:

<code>text_string</code>	Text string
--------------------------	-------------

### Example:

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

html_str = ConvertToHTML ( str );
# html_string = "Hello&nbsp;world&nbsp;!!!<br>How&nbsp;pare&nbsp;pyou&nbsp;today?"
```

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

## 21.4 Pause()

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

**Format:** `Pause()`

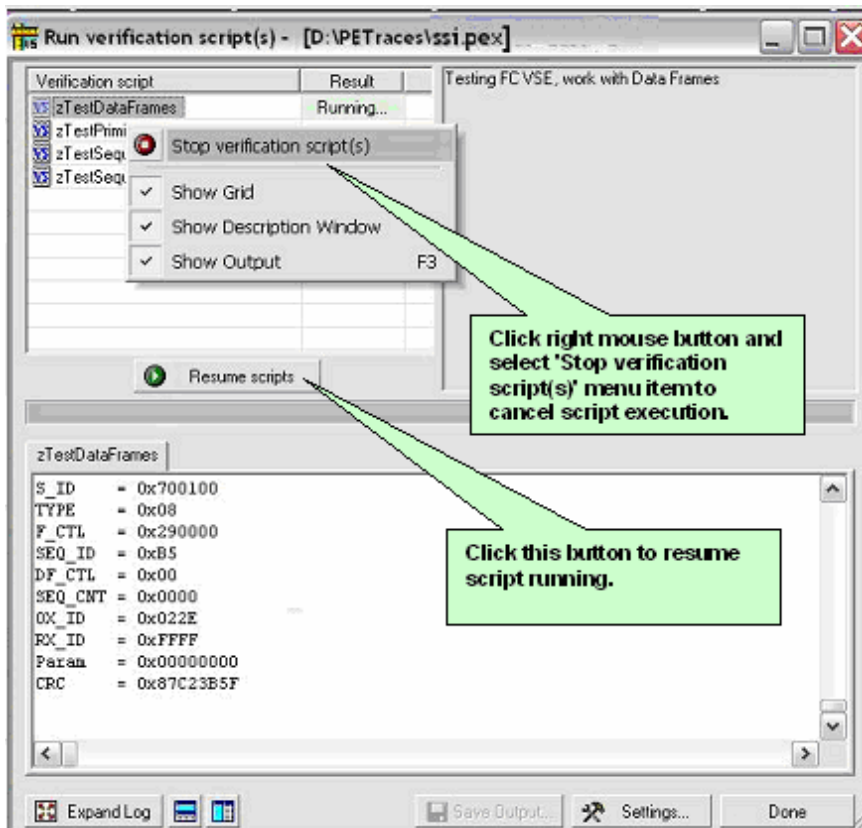
**Example:**

```
...
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 The VSE Important Script Files

The VSE working files are located in the `..\Scripts\VFScripts` subfolder of the main *PE Tracer™* folder. The current version of VSE includes the following files:

File	Description
<b>VSTools.inc</b>	Main VSE file containing definitions of some generic and PCI Express-specific VSE script functions provided by LeCroy (must be included in every script)
<b>VS_constants.inc</b>	File containing definitions of some important generic and PCI Express-specific VSE global constants
<b>VSTemplate.pev_</b>	Template file for new verification scripts.
<b>VSUser_globals.inc</b>	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.)

### 22.1 Example Script Files

The VSE example files are located in the `..\Scripts\VFScripts\Samples` subfolder of the main *PE Tracer* folder. The current version of VSE includes the following files:

File	Description
examp_tlp_data.inc	Sample include file containing definitions and functions used by some other sample scripts
examp_dllps.pevs	Sample processing script that outputs information about DLLP packets present in the trace
examp_tlps.pevs	Sample processing script that outputs information about TLP packets present in the trace
examp_ordered_sets.pevs	Sample processing script that outputs information about Ordered Set and Link Condition packets present in the trace
examp_check_errors.pevs	Sample PASS/FAIL script that checks all packets in the trace for all the errors VSE exports and fails in case any error is found
examp_link_transactions.pevs	Sample processing script that outputs information about Link Transactions present in the trace
examp_split_transactions.pevs	Sample processing script that outputs information about Split Transactions present in the trace
examp_metrics.pevs	Sample processing script that outputs information about Memory Write Link Transaction metrics and all Split Transaction metrics

# How to Contact LeCroy

<b>Type of Service</b>	<b>Contact</b>
Call for technical support...	US and Canada: 1 (800) 909-2282 Worldwide: 1 (408) 653-1260
Fax your questions...	Worldwide: 1 (408) 727-6622
Write a letter ...	LeCroy Corporation Protocol Solutions Group Customer Support 3385 Scott Blvd. Santa Clara, CA 95054-3115
Send e-mail...	<a href="mailto:psgsupport@lecroy.com">psgsupport@lecroy.com</a>
Visit LeCroy web site...	<a href="http://www.LeCroy.com/">http://www.LeCroy.com/</a>