Computer Access Technology Corporation

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

Verification Script Engine

for

CATC FC*Tracer*

Reference Manual

Manual Version 1.10

August 16, 2004

Document Version

This is manual version 1.10 and is based on FCTracer software version 2.01.

Table of Contents

1 Introduction

This document contains a description of the CATC's Verification Script Engine (VSE), a new utility in the FCTracer software that allows users to perform custom analyses of Fibre Channel (FC) traffic, recorded using the new generation of CATC's Fibre Channel protocol analyzers.

VSE allows users to ask the FCTracer application to send some desired "events" (currently defined as frames and sequences) from an FC trace to a verification script written using 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 is designed to allow users to easily retrieve information about any field in FC frame headers or FCP SSCI sequences, and to make complex timing calculations between different events in recorded trace. It also allows the filtering-in or filtering-out of data with dynamically changing filtering conditions, the porting of information to a special output window, the saving of data to text files, and the sending of data to COM clients connected to FCTracer application.

2 Verification Script Structure

Writing verification scripts is easy so long as you follow a few rules and have some understanding of how FCTracer application interacts with running scripts.

The main script file that contains the text of the varification script should have extension .vse, and be located in the subfolder ..\Scripts\VFScripts of the main FCTracer folder. Some other files might be included in the main script file using directive %include. (see CATC Script Language (CSL) manual for details).

The following schema can present the common structure of verification script:

```
#
#
#   VS1.vse
#
#   Verification script
#
#   Brief Description:
#   Verify something…
#


####################################################################################
Module info                                   #
####################################################################################      Filling of this
block is necessary for proper verification script operation...           #
####################################################################################

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 initial values for global variables         #
 # and what kinds of trace events should be passed to the script.                 #
 # ( By default, only Primitive events from all channels                    #
 #  will be 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.                                                  #
 ###########################################################################################




   # 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()
{
   #
   # The function below will show specified message only one time -
   # no matter how many times ProcessEvent is called.
   #
   ShowStartPrompt("ShowStartPrompt\n");


   # Write the body of this function depending on your needs …


   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 == "Blah" ) return 1;
            return 0;
}
```

3 Interaction between FCTracer and verification script

When a user runs a script over a 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, some setup routines can be made such as specifying what kind of trace events should be passed to the script, and setting up initial values of global script-specific global variables.

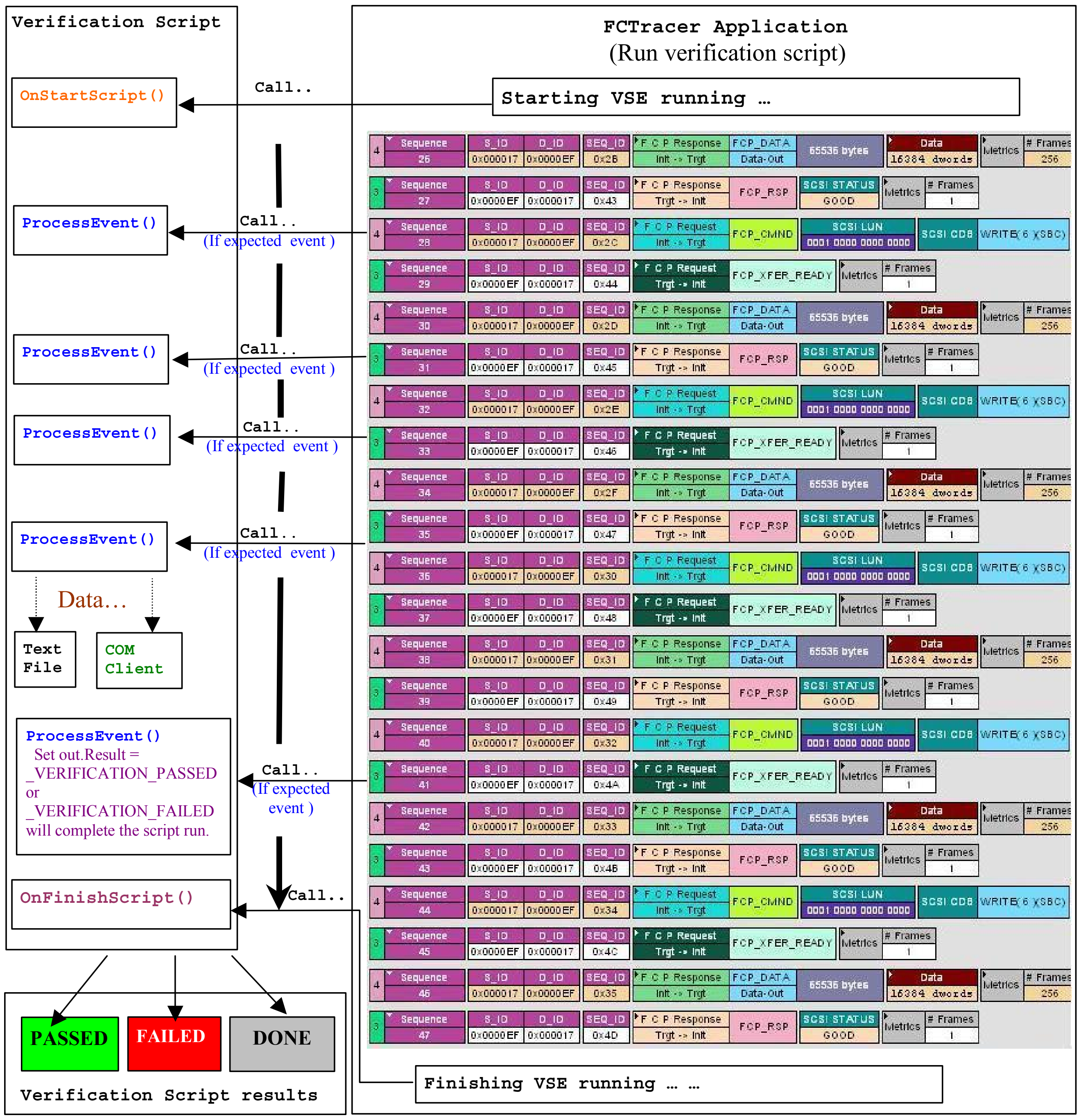
2. Next, the VSE goes through the recorded trace and checks if the current frame suits specified sending criteria – if it does, VSE calls the script main processing function ProcessEvent() and sends some information about the current event to the script input context variables.
(Please refer to the topic "Input context variables" below in this document for full description of verification script input context variables)

3. ProcessEvent() – is the main verification routine in which all processing of incoming trace event is done. 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 script running 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 output context variable : out.Result = _VERIFICATION_PASSED or _VERIFICATION_FAILED.
(Please refer to the topic "Output context variables" below in this document for full description of verification script output context variables)

   NOTE: Not only does a verification script evaluate recorded traces by 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 to save extracted data in text files or send it to other applications via COM/Automation interfaces)

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

The following figure illustrates the interaction between FCTracer application and running verification script:

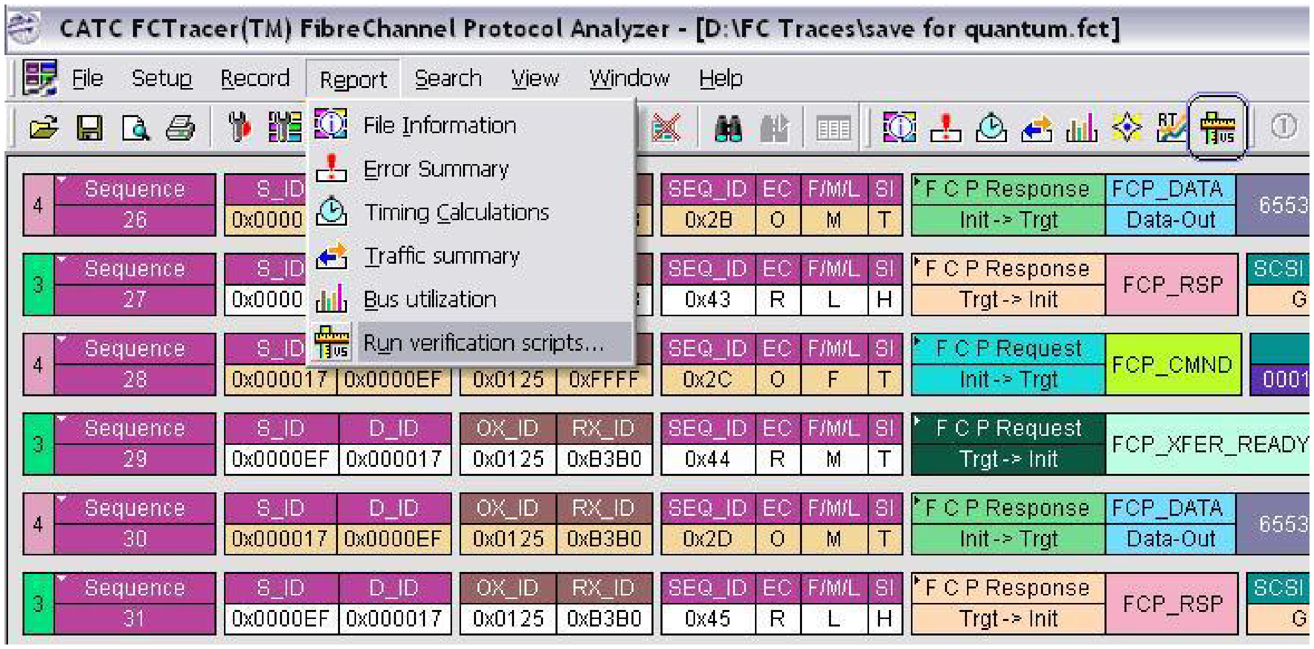

Verification Script
OnStartScript()
ProcessEvent()
ProcessEvent()
ProcessEvent()
ProcessEvent()
Data…
Text File
COM Client
ProcessEvent()
Set out.Result = _VERIFICATION_PASSED or _VERIFICATION_FAILED will complete the script run.
OnFinishScript()
Call..
Call.. (If expected event)
Call.. (If expected event)
Call.. (If expected event)
Call.. (If expected event)
Call.. (If expected event)
Call..
PASSED
FAILED
DONE
Verification Script results
FCTracer Application
(Run verification script)
Starting VSE running …
Finishing VSE running … …

```
4 Sequence 26 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x2B | F C P Response Init -> Trgt | FCP_DATA Data-Out | 65536 bytes | Data 16384 dwords | Metrics | # Frames 256
3 Sequence 27 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x43 | F C P Response Trgt -> Init | FCP_RSP | SCSI STATUS GOOD | Metrics | # Frames 1
4 Sequence 28 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x2C | F C P Request Init -> Trgt | FCP_CMND | SCSI LUN 0001 0000 0000 0000 | SCSI CDB | WRITE(6)(SBC)
3 Sequence 29 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x44 | F C P Request Trgt -> Init | FCP_XFER_READY | Metrics | # Frames 1
4 Sequence 30 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x2D | F C P Response Init -> Trgt | FCP_DATA Data-Out | 65536 bytes | Data 16384 dwords | Metrics | # Frames 256
3 Sequence 31 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x45 | F C P Response Trgt -> Init | FCP_RSP | SCSI STATUS GOOD | Metrics | # Frames 1
4 Sequence 32 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x2E | F C P Request Init -> Trgt | FCP_CMND | SCSI LUN 0001 0000 0000 0000 | SCSI CDB | WRITE(6)(SBC)
3 Sequence 33 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x46 | F C P Request Trgt -> Init | FCP_XFER_READY | Metrics | # Frames 1
4 Sequence 34 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x2F | F C P Response Init -> Trgt | FCP_DATA Data-Out | 65536 bytes | Data 16384 dwords | Metrics | # Frames 256
3 Sequence 35 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x47 | F C P Response Trgt -> Init | FCP_RSP | SCSI STATUS GOOD | Metrics | # Frames 1
4 Sequence 36 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x30 | F C P Request Init -> Trgt | FCP_CMND | SCSI LUN 0001 0000 0000 0000 | SCSI CDB | WRITE(6)(SBC)
3 Sequence 37 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x48 | F C P Request Trgt -> Init | FCP_XFER_READY | Metrics | # Frames 1
4 Sequence 38 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x31 | F C P Response Init -> Trgt | FCP_DATA Data-Out | 65536 bytes | Data 16384 dwords | Metrics | # Frames 256
3 Sequence 39 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x49 | F C P Response Trgt -> Init | FCP_RSP | SCSI STATUS GOOD | Metrics | # Frames 1
4 Sequence 40 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x32 | F C P Request Init -> Trgt | FCP_CMND | SCSI LUN 0001 0000 0000 0000 | SCSI CDB | WRITE(6)(SBC)
3 Sequence 41 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x4A | F C P Request Trgt -> Init | FCP_XFER_READY | Metrics | # Frames 1
4 Sequence 42 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x33 | F C P Response Init -> Trgt | FCP_DATA Data-Out | 65536 bytes | Data 16384 dwords | Metrics | # Frames 256
3 Sequence 43 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x4B | F C P Response Trgt -> Init | FCP_RSP | SCSI STATUS GOOD | Metrics | # Frames 1
4 Sequence 44 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x34 | F C P Request Init -> Trgt | FCP_CMND | SCSI LUN 0001 0000 0000 0000 | SCSI CDB | WRITE(6)(SBC)
3 Sequence 45 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x4C | F C P Request Trgt -> Init | FCP_XFER_READY | Metrics | # Frames 1
4 Sequence 46 | S_ID 0x000017 | D_ID 0x0000EF | SEQ_ID 0x35 | F C P Response Init -> Trgt | FCP_DATA Data-Out | 65536 bytes | Data 16384 dwords | Metrics | # Frames 256
3 Sequence 47 | S_ID 0x0000EF | D_ID 0x000017 | SEQ_ID 0x4D | F C P Response Trgt -> Init | FCP_RSP | SCSI STATUS GOOD | Metrics | # Frames 1
```

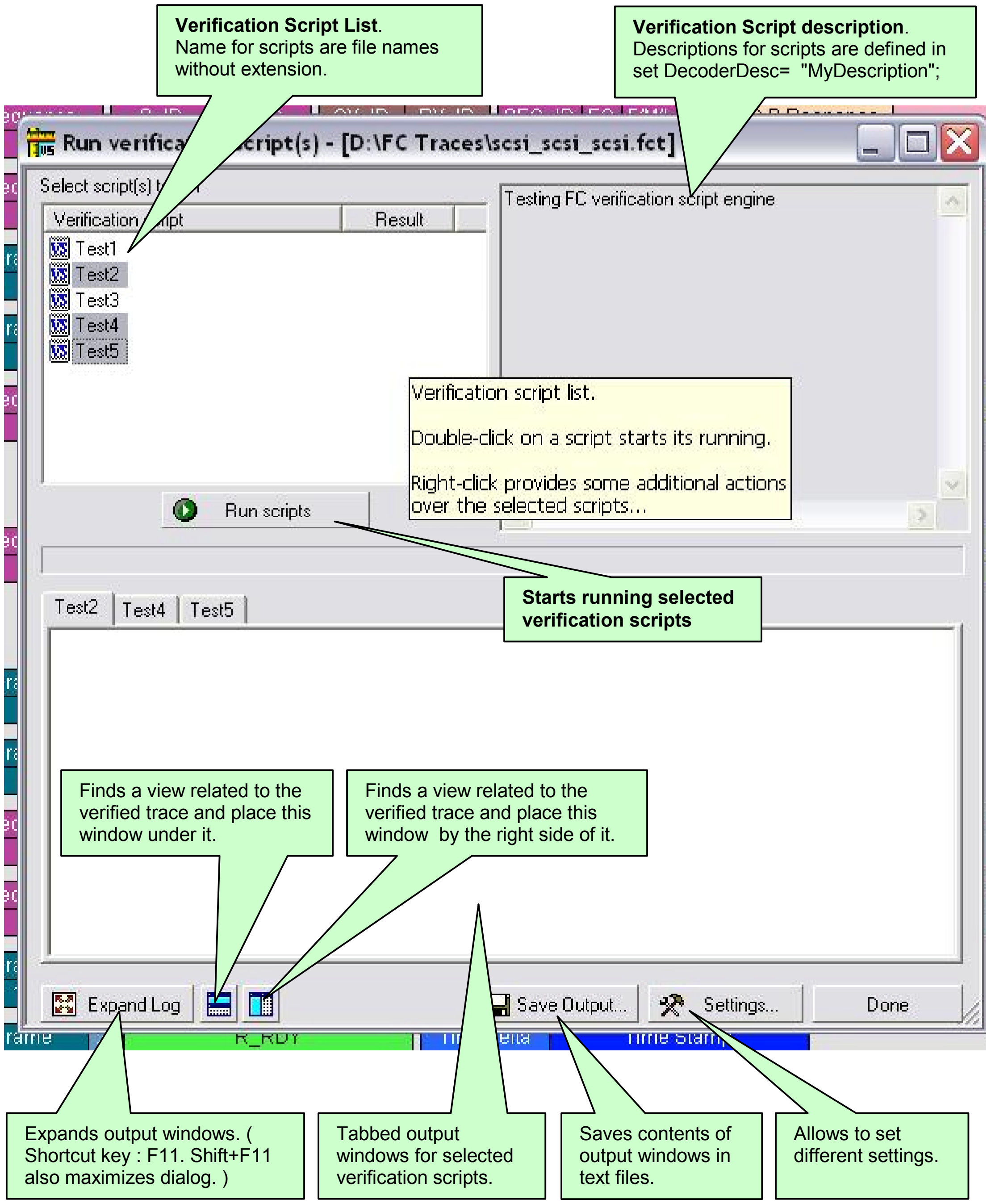
The Verification script result "DONE" occurs when the script has been configured to extract and display some information about the trace but to not display PASSED/FAILED results. To configure a script so that it only displays information – place a call somewhere in your script for the function ScriptForDisplayOnly())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

4 Running verification scripts from FCTracer

In order to run a verification script over a trace - you need to open the FCTracer main menu item: Report\Run verification scripts… or push the button on the main toolbar (if it is not hidden):


CATC FCTracer(TM) FibreChannel Protocol Analyzer - [D:\FC Traces\save for quantum.fct]
File Setup Record Report Search View Window Help
File Information
Error Summary
Timing Calculations
Traffic summary
Bus utilization
Run verification scripts...

The special dialog will open displaying a list of verifications scripts. You can select one or several scripts from the list to run:



Verification Script List. Name for scripts are file names without extension.

Verification Script description. Descriptions for scripts are defined in set DecoderDesc= "MyDescription";

Verification script list.

Double-click on a script starts its running.

Right-click provides some additional actions over the selected scripts...

Starts running selected verification scripts

Finds a view related to the verified trace and place this window under it.

Finds a view related to the verified trace and place this window by the right side of it.

Expands output windows. (Shortcut key : F11. Shift+F11 also maximizes dialog.)

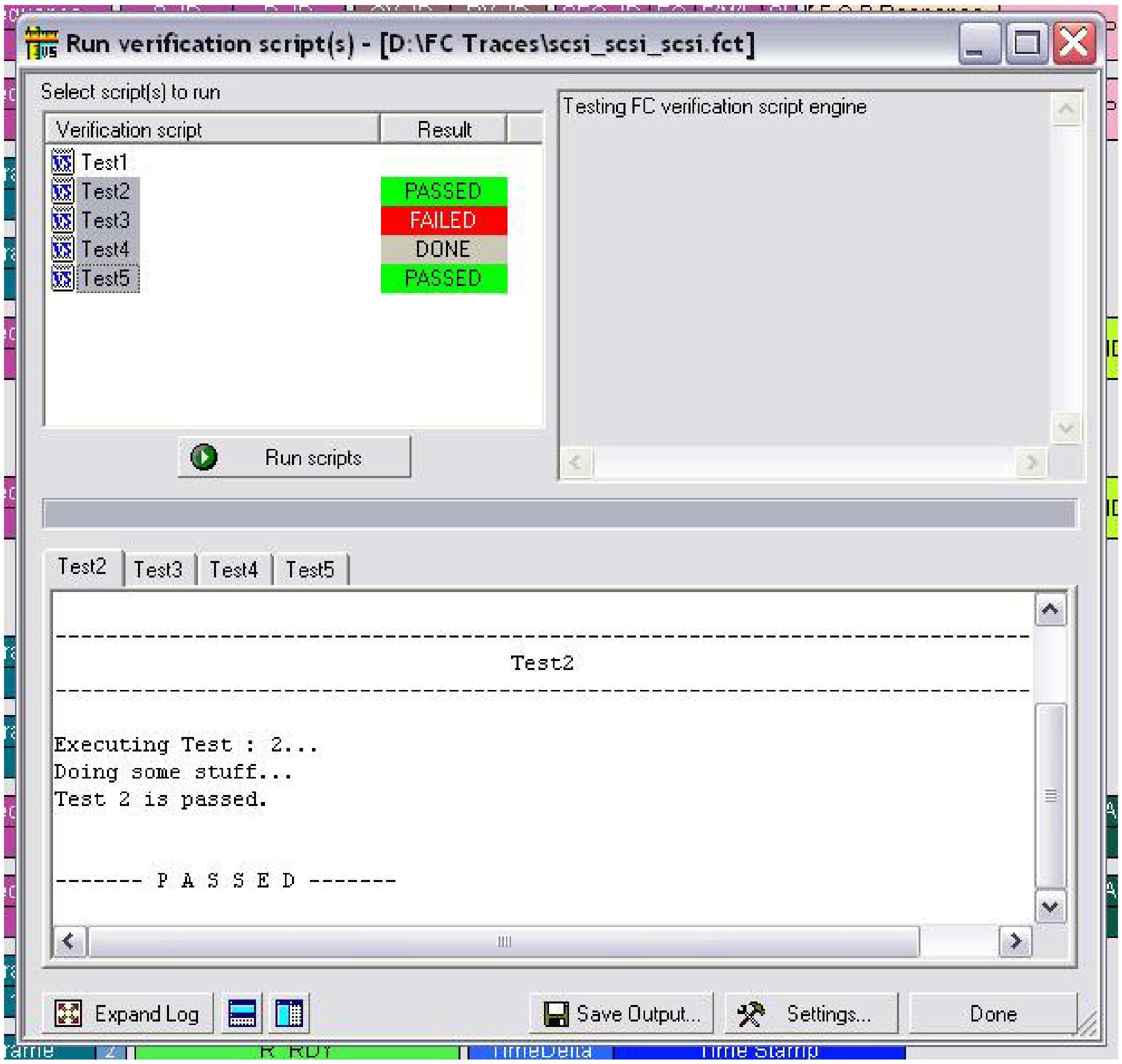
Tabbed output windows for selected verification scripts.

Saves contents of output windows in text files.

Allows to set different settings.

4.1 Running verification scripts

Push the button ‘Run scripts’ after you selected desired scripts to run. VSE will start running selected verification scripts, show script report information in the output windows and present results of verifications in the script list:



Verification script	Result
Test1	
Test2	PASSED
Test3	FAILED
Test4	DONE
Test5	PASSED

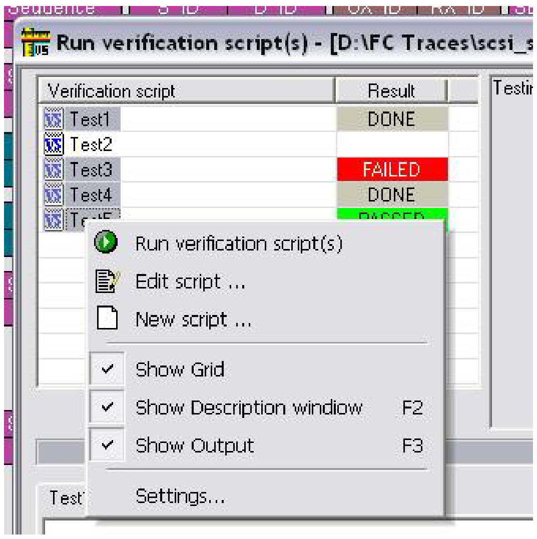
```
Testing FC verification script engine
```

```
----------------------------------------------------------------------------
                                   Test2
----------------------------------------------------------------------------

Executing Test : 2...
Doing some stuff...
Test 2 is passed.


------- P A S S E D -------
```

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 the editing of selected scripts 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**.**

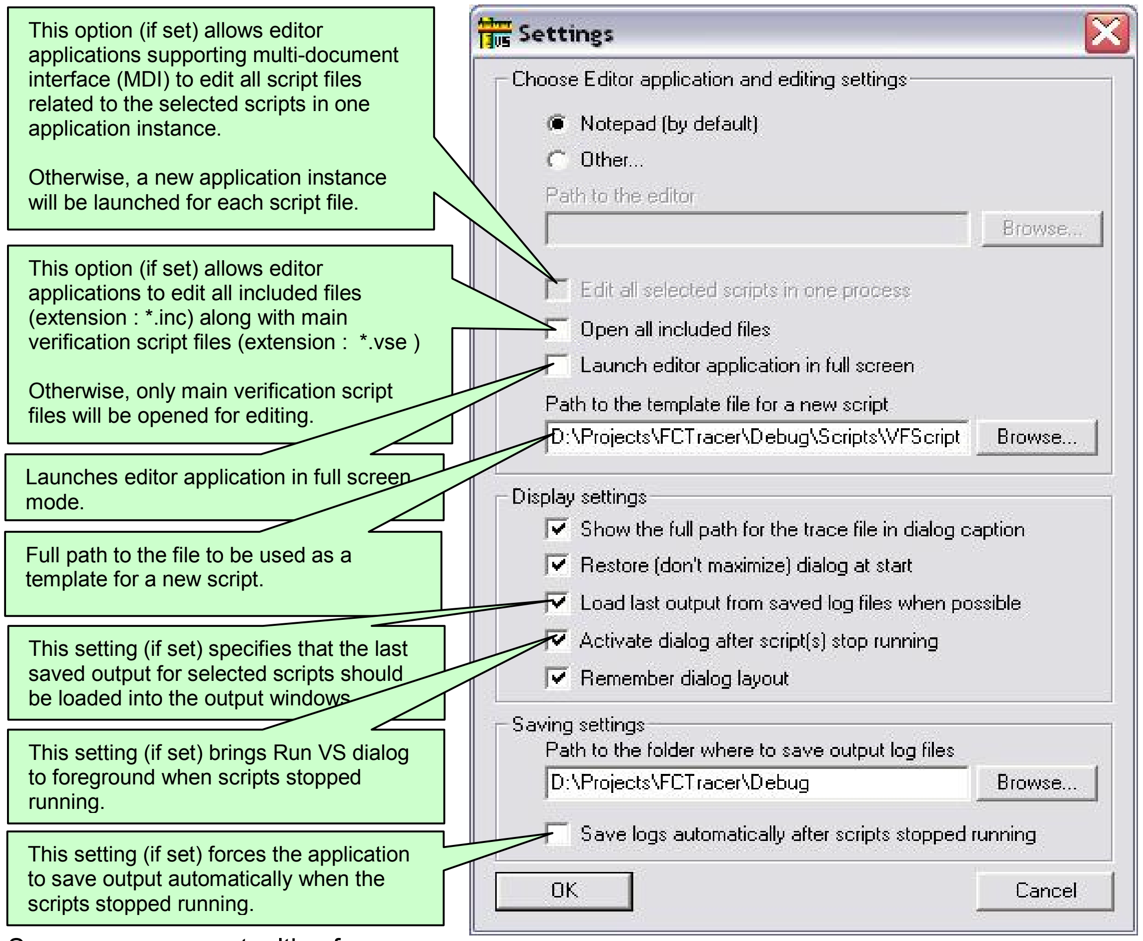
- **Show Description window –** shows/hides the script description window.**(Shortcut key : F2)**

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

- **Settings** – opens a special 'Setting' dialog which allows to specify different settings for VSE.

4.2 VSE GUI settings.

After choosing ‘Settings’ the following dialog will appear:



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.

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.

Launches editor application in full screen mode.

Full path to the file to be used as a template for a new script.

This setting (if set) specifies that the last saved output for selected scripts should be loaded into the output windows

This setting (if set) brings Run VS dialog to foreground when scripts stopped running.

This setting (if set) forces the application to save output automatically when the scripts stopped running.

Settings

Choose Editor application and editing settings

● Notepad (by default)
○ Other...

Path to the editor:

☐ 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\FCTracer\Debug\Scripts\VFScript

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\FCTracer\Debug

☐ Save logs automatically after scripts 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 which 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 format : 2 sec 125 ns -> [2 , 125]. See 11.1 VSE time object for details)

in.Channel - channel where the event occured. (may be from _CHANNEL_1 (1) to _CHANNEL_8 (8)) indicating on which channel the event occurred)

in.TraceEvent - type of trace events(application predefined constants are used. See list of possible events below)

in.Notification - type of notifications (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 variables:

5.2.1 All frame/sequence-specific set of members

Members of this set are valid for any frames.

in.Payload - bit source of the frame/sequence payload (you can extract any necessary information using GetNBits(), NextNBits() or PeekNBits() functions – please refer to CSL Manual for details about these functions)

in.PayloadLength - the length(in bytes of the retrieved frame payload)

NOTE: Because a payload may be very large – the VSE sets up the in.Payload member up to special limit (by default 16Mb). All attempts to get data beyond this limit will produce null values. This limit can be changed via SetTraPayloadLimit () function.

5.2.2 Primitive frame-specific set of members

(valid for primitive frames only, undefined for other events)

in.Byte1	– first byte of primitive
in.Byte2	– second byte of primitive
in.Byte3	– third byte of primitive
in.Byte4	– fourth byte of primitive
in.Primitive	– DWORD value of primitive
in.NumOfPackedPrims	– Contains the number of packed primitives (0 – for regular primitives)

5.2.3 Data frame-specific set of members

(valid for data frames only, undefined for other events)

in.R_CTL	– Routing Control
in.D_ID	– Destination ID
in.CS_CTL	– Class Specific Control
in.S_ID	– Source ID
in.TYPE	– Data structure type
in.F_CTL	– Frame Control
in.SEQ_ID	– Sequence ID
in.D_ID	– Destination ID
in.DF_CTL	– Data Field Control
in.SEQ_CNT	– Sequence Count
in.OX_ID	– Originator Exchange ID
in.RX_ID	– Responder Exchange ID
in.Param	– Parameter
in.CRC	– CRC

5.2.4 Sequence-specific set of members

(valid for sequences only, undefined for other events)

in.S_ID	– Source ID
in.D_ID	– Destination ID
in.OX_ID	– Originator Exchange ID
in.RX_ID	– Responder Exchange ID
in.SEQ_ID	– Sequence ID
in.EC	– Exchange context (0 – Originator, 1 - Responder)
in.FML	– integer value which is either 0 for middle sequence of exchange or combination of the following flags : _SEQ_FIRST, _SEQ_LAST. (Hint : use '&' operator to check if this value contains some flag. Like: if(in.FML & _SEQ_FIRST) DoSomething();)
in.SI	– Sequence Initiative (0 – hold, 1 – transfer)
in.TYPE	– Data structure type of the first frame of sequence
in.R_CTL	– Routing Control of the first frame of sequence
in.NumOfFrames	– Number of frames in the sequence
in.Pld_Bytes	– Number of payload bytes the sequence transferred
in.RespTime	– Time from the beginning of the first frame to the end of the last frame (it has the same format as in.Time. See 11.1 VSE time object for details)
in.Thrpt	– The sequence payload divided by response time expressed in megabytes per second

6 Verification Script Engine Output Context members

All verification scripts have output contexts – some special structures which 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 - the result of the whole verification program defined in the verification script.

This member is supposed to have 3 values:
_VERIFICATION_PROGRESS,(is set by default when script starts running)
_VERIFICATION_PASSED,
_VERIFICATION_FAILED

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

If you don't specify any of those values - the result of script execution will be set as VERIFICATION_FAILED at exit.

NOTE: If you don't care about result of script running please call function ScriptForDisplayOnly() **one time before stopping script – then the result will be DONE.**

7 Verification Script Engine events

VSE defines a large group of trace "events" – both on frame and sequence 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.

8 Frame level events

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

Types of frames	**in.TraceEvent**
Disconnect (FC link was disconnected)	_DISCONNECT
Connect (FC link was connected)	_CONNECT
Primitive frames	_FRM_PRIMITIVE
FCP frames	_FRM_FCP
FC-GS frames	_FRM_GS
FC-SW frames	_FRM_SW
Data frames	_FRM_DATA

9 Sequence level events

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

Types of sequences	**in.TraceEvent**
FCP sequences	_SEQ_FCP
FC-GS sequences	_SEQ_GS
FC-SW sequences	_SEQ_SW
Other kinds of FC sequences	_SEQ_OTHER

10 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.

10.1 SendLevel()

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

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

Parameters: *level* – This parameter can be one of following values:

_FRM	– (value 0) send Frame level events
_SEQ	– (value 1) send Sequence level events

Example:

```
…
SendLevel( _FRM ); # - send frame level events
SendLevel( _SEQ  ); # - send sequence level events
```

Remark:

If no level was specified – events of frame level will be sent to the script by default.

10.2 SendLevelOnly()

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

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

Parameters: *level* – This parameter can be one of following values:

_FRM	– (value 0) send Frame level events
_SEQ	– (value 1) send Sequence level events

Example:

```
…
SendLevelOnly( _SEQ  ); # - send ONLY sequence level events
```

10.3 DontSendLevel()

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

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

Parameters: *level* – This parameter can be one of following values:

_FRM	– DO NOT send Frame level events
_SEQ	– DO NOT send Sequence level events

Example:

```
…
DontSendLevel( _FRM ); # - DO NOT send frame level events
DontSendLevel( _SEQ  ); # - DO NOT send sequence level events
```

10.4 SendChannel()

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

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

Parameters: *channel* – This parameter can be one of following values:

_CHANNEL_1 (= 1)	– send events from channel 1
_CHANNEL_2 (= 2)	– send events from channel 2
_CHANNEL_3 (= 3)	– send events from channel 3
_CHANNEL_4 (= 4)	– send events from channel 4
_CHANNEL_5 (= 5)	– send events from channel 5
_CHANNEL_6 (= 6)	– send events from channel 6
_CHANNEL_7 (= 7)	– send events from channel 7
_CHANNEL_8 (= 8)	– send events from channel 8

Example:

```
…
SendChannel(_CHANNEL_1); # - send events from channel 1
SendChannel(_CHANNEL_5); # - send events from channel 5
…
SendChannel( 6 ); # - send events from channel 6
SendChannel( 8 ); # - send events from channel 8
…
```

10.5 SendChannelOnly()

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

Format : **SendChannelOnly(*channel*)**

Parameters: *channel* – This parameter can be one of following values:

_CHANNEL_1 (= 1)	– send ONLY events from channel 1
_CHANNEL_2 (= 2)	– send ONLY events from channel 2
_CHANNEL_3 (= 3)	– send ONLY events from channel 3
_CHANNEL_4 (= 4)	– send ONLY events from channel 4
_CHANNEL_5 (= 5)	– send ONLY events from channel 5
_CHANNEL_6 (= 6)	– send ONLY events from channel 6
_CHANNEL_7 (= 7)	– send ONLY events from channel 7
_CHANNEL_8 (= 8)	– send ONLY events from channel 8

Example:

```
…
SendChannelOnly( _CHANNEL_1 );  # - send ONLY events from channel 1
SendChannelOnly( _CHANNEL_5 );  # - send ONLY events from channel 5
…
SendChannelOnly( 6 );  # - send ONLY events from channel 6
SendChannelOnly( 8 );  # - send ONLY events from channel 8
…
```

10.6 DontSendChannel ()

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

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

Parameters: *channel* – This parameter can be one of following values:

_CHANNEL_1 (= 1)	– DO NOT send events from channel 1
_CHANNEL_2 (= 2)	– DO NOT send events from channel 2
_CHANNEL_3 (= 3)	– DO NOT send events from channel 3
_CHANNEL_4 (= 4)	– DO NOT send events from channel 4
_CHANNEL_5 (= 5)	– DO NOT send events from channel 5
_CHANNEL_6 (= 6)	– DO NOT send events from channel 6
_CHANNEL_7 (= 7)	– DO NOT send events from channel 7
_CHANNEL_8 (= 8)	– DO NOT send events from channel 8

Example:

```
…
DontSendChannel ( _CHANNEL_1 );  # - DO NOT send events from channel 1
DontSendChannel ( _CHANNEL_5 );  # - DO NOT send events from channel 5
```

```
…
DontSendChannel ( 6 );  # - DO NOT send events from channel 6
DontSendChannel ( 8 );  # - DO NOT send events from channel 8
…
```

10.7 SendAllChannels()

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

Format : **SendAllChannels ()**

Example:

```
…
SendAllChannels ();  # - send events from ALL channels
```

10.8 SendTraceEvent ()

This function specifies the events to be sent to script.

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

Parameters: *event* – This parameter may be one of the following values:

Frame level events:

event value	**Description**
_DISCONNECT	Disconnect (FC link was disconnected)
_CONNECT	Connect (FC link was connected)
_FRM_PRIMITIVE	Primitive frames
_FRM_FCP	FCP frames
_FRM_GS	FC-GS frames
_FRM_SW	FC-SW frames
_FRM_DATA	Data frames

Sequence level events:

event value	**Description**
_SEQ_FCP	FCP sequences
_SEQ_GS	FC-GS sequences
_SEQ_SW	FC-SW sequences
_SEQ_OTHER	Other kinds of FC sequences

Example:

```
…
SendTraceEvent( _FRM_FCP  );
```

```
…

SendLevel( _SEQ );
SendTraceEvent ( _SEQ_FCP ); # - send FCP sequences
```

10.9 DontSendTraceEvent()

This function specifies that the event specified in this function shouldn't be sent to script.

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

Parameters: *event* – See SendTraceEvent () for all possible values.

Example:

```
        …
SendLevel( _SEQ );                  # Send sequence level events
SendTraceEvent ( _SEQ_FCP );    # Send FCP sequences
SendTraceEvent ( _SEQ_GS   );       # Send GS sequences
SendTraceEvent ( _SEQ_SW   );       # Send SW sequences
…
if( SomeCondition )
{
        DontSendTraceEvent ( _SEQ_GS );   # Don't send FC-GS sequences
        DontSendTraceEvent ( _SEQ_SW );  # Don't send FC-SW sequences

        # Only FC-FCP sequences will be sent.
}
```

10.10 SendTraceEventOnly()

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

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

Parameters: *event* – See SendTraceEvent () for all possible values.

Remark: This function may be useful when there are many events to be sent and it is needed to send only one kind of events and turn off the other ones.

Example:

```
        …
SendLevel( _SEQ );                    # Send sequence level events
SendTraceEvent ( _SEQ_FCP );    # Send FCP sequences
SendTraceEvent ( _SEQ_GS   );       # Send GS sequences
SendTraceEvent ( _SEQ_SW   );       # Send SW sequences
…
if( SomeCondition )
{


        SendTraceEventOnly ( _SEQ_FCP );
        # Only FC-FCP sequences will be sent.
}
```

10.11 SendAllTraceEvents()

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

Format : **SendAllTraceEvents ()**

Example:

```
        …
SendLevel( _SEQ );          # Send sequence level events
SendAllTraceEvents ( );    # Send FCP, SW, GS and other sequences
```

10.12 SendPrimitive()

This function specifies more precise tuning for sending primitive frames. (Currently not implemented)

Format : **SendPrimitive(*primitive*)**

Parameters:

primitive – dword primitive value. This parameter may be one of the following values:

Primitive values:

Constant	Primitive
_IDLE	Idle Primitive
_R_RDY	Receiver Ready
_VC_RDY	Virtual Circuit Ready
_ARB_yx	Arbitrate
_ARB_val	Arbitrate
_OPN_yx	Open full-duplex
_OPN_yy	Open full-duplex
_OPN_yr	Open selective replicate
_OPN_fr	Open broadcast replicate
_CLS	Close
_MRK_tx	Mark
_DHD	Dynamic Half-Duplex
_OLS	Offline
_NOS	Not Operational
_LR	Link Reset
_LRR	Link Reset Response
_LIP_F7_F7	Loop Initialization--F7,F7
_LIP_F8_F7	Loop Initialization--F8,F7
_LIP_F7_x	Loop Initialization--F7,x
_LIP_F8_x	Loop Initialization--F8,x
_LIP_yx	Loop Initialization--reset
_LIP_fx	Loop Initialization--reset all
_LIP_ba	Loop Initialization--reserved
_LPE_yx	Loop Port Enable
_LPE_fx	Loop Port Enable all
_LPB_yx	Loop Port Bypass
_LPE_fx	Loop Port Bypass all
_SYN_x	Clock Synchronization
_SYN_y	Clock Synchronization
_SYN_z	Clock Synchronization

Example:

```
SendPrimitive( _R_RDY );    # - send ‘Receiver Ready’ primitive
```

10.13 SendFCPSeq ()

This function specifies more precise tuning for FCP-SCSI sequences.

Format : **SendFCPSeq(*type*, *seq_info* = *null*)**

Parameters:

type – This parameter s
pecifies that only FCP sequences with this type will be sent. This
parameter may be one of the following values:

Type	Meaning
_FCP_CMND	FCP CMND Information Unit
_FCP_XFER_RDY	FCP XFER_RDY Information Unit
_FCP_DATA	FCP DATA Information Unit
_FCP_RSP	FCP RSP Information Unit

seq_info – This list parameter specifies some sequence attributes and has the following structure:

[*s_id* , *d_id*, *ox_id*, *rx_id*]

where

s_id	– Source ID,
d_id	– Destination ID,
ox_id	– Originator Exchange ID,
rx_id	– Responder Exchange ID

NOTE: It is allowed to use constant _ANY as a paramter value to specify that any value is acceptable.

If some of the parameters are missing – it is assumed that they are equal to _ANY.

If the whole seq_info parameter is missing – it is assumed that any combination of the s_id, d_id, ox_id, rx_id is acceptable.

Example:

```
 # - send FCP CMND sequence.
 SendFCPSeq(  _FCP_CMND  ) ;

 …
 seq_info = [ 0x700100, 0x683456, _ANY, _ANY ];
SendFCPSeq(  _FCP_CMND, seq_info  );
 seq_info = [ 0x700100, 0x683456 ];
SendFCPSeq(  _FCP_CMND, seq_info  ); # the same as the previous case.
 …

 seq_info = [ 0x700100, 0x683456, 0x022D];
 SendFCPSeq( _FCP_RSP, seq_info );
```

10.14 SendFCPCmndSeq ()

This function specifies more precise tuning for FCP-SCSI CMND sequences.

Format : **SendFCPCmndSeq(*opcode* = *_ANY*,
field_name = "" ,
field_value = 0,
seq_info = *null*)**

Parameters:

opcode – This parameter specifies that only FCP CMND sequences containing SCSI CDB with this opcode will be sent. (The value *_ANY* means any FCP CMND sequences)

field_name – This parameter specifies that only FCP CMND sequences having a field with *field_name* (how it is shown in CATC trace) and value equal to *field_value* will be sent.
(This field is valid only if the parameter 'opcode' is not equal to *_ANY*)

field_value – This parameter specifies that only FCP CMND sequences having a field with *field_name*(how it is shown in CATC trace) and value equal to *field_value* will be sent.
(This field is valid only if parameter 'opcode' is not equal to *_ANY* and parameter 'field_name' is not empty)

NOTE: For fields having size more than 32 bits use raw binary values (like : '0011223344556677FF') For more information about raw binary values please refer to CATC CSL Manual.

seq_info – This list parameter specifies some sequence attributes.
See SendFCPSeq() function for details.

Example:

```
# - send any FCP CMND sequence.
SendFCPCmndSeq( );

# - send FCP CMND sequence where the field 'Opcode' has value 0x28 (READ(10))
# and the field 'Logical Block Addr' has value 0x00020249.
SendFCPCmndSeq(  0x28,  null, "Logical Block Addr",  0x00020249 );

const READ_10 = 0x28;
…
seq_info = [ 0x700100, 0x683456, 0x022D];
SendFCPCmndSeq ( READ_10, "Logical Block Addr",  0x00020249, seq_info  );
```

10.15 0SendFCPRspSeq ()

This function specifies more precise tuning for FCP-SCSI RSP sequences.

Format : **SendFCPRspSeq(*status* = *_ANY*,**
_field_name_ = “” ,
_field_value_ = 0,
***seq_info* = *null*)**

Parameters:

status – This parameter specifies that only FCP RSP sequences containing SCSI Status equal to the parameter value will be sent. (The value *_ANY* means FCP RSP sequences with any status will be sent)

field_name – This parameter specifies that only FCP CMND sequences having a field with *field_name* (how it is shown in CATC trace) and value equal to *field_value* will be sent.
(This field is valid only if the parameter ‘opcode’ is not equal to *_ANY*)

field_value – This parameter specifies that only FCP CMND sequences having a field with *field_name*(how it is shown in CATC trace) and value equal to *field_value* will be sent.
(This field is valid only if parameter ‘opcode’ is not equal to *_ANY* and parameter ‘field_name’ is not empty)

NOTE: For fields having size more than 32 bits use raw binary values (like : ‘0011223344556677FF’) For more information about raw binary values please refer to CATC CSL Manual.

seq_info – This list parameter specifies some sequence attributes.
See SendFCPSeq() function for details.

NOTE: For fields having size more than 32 bits use raw binary values (like : ‘0011223344556677FF’) For more information about raw binary values please refer to CATC CSL Manual.

Example:

```
# - send any FCP RSP sequence occurred in the recorded trace.
SendFCPRspSeq( );

const GOOD = 0;
…
SendFCPRspSeq(  GOOD );
…
seq_info = [ 0x700100, 0x683456, 0x022D];
SendFCPRspSeq( GOOD, "", 0, seq_info );
```

11 Timer functions

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

11.1 VSE time object

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

[seconds, nanoseconds]

NOTE: The best way to construct VSE time object is to use Time() function (see below).

11.2 SetTimer()

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

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

Parameters:

timer_id – a 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 second time for the same timer id – it resets timer and starts timing calculation again from the point where it was called.

11.3 KillTimer()

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

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

Parameters:

timer_id – a unique timer identifier.

Example:

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

11.4 GetTimerTime()

Retrieve timing interval from the specific timer

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

Parameters:

timer_id – a 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, doesn’t reset timer.

12 Time construction functions

This group of functions is used to construct VSE time objects.

12.1 Time()

Constructs verification script time object.

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

Return values:

First function returns *[0, nanoseconds]*, second one 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".

13 Time calculation functions

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

13.1 AddTime()

Adds two VSE time objects

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

Return values:

Returns VSE time object presenting time interval equal to sum of time_1 and time_2

Parameters:

time_1	- VSE time object presenting first time interval
time_2	- VSE time object presenting second time interval

Example:

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

13.2 SubtractTime()

Subtract two VSE time objects

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

Return values:

Returns VSE time object presenting time interval equal to subtraction of time_1 and time_2

Parameters:

time_1	- VSE time object presenting first time interval
time_2	- VSE time object presenting second time interval

Example:

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

13.3 MulTimeByInt()

Multiplies VSE time object by integer value

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

Return values:

Returns VSE time object presenting time interval equal to time * mult

Parameters:

time - VSE time object
mult - multiplier, integer value

Example:

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

13.4 DivTimeByInt()

Divides VSE time object by integer value

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

Return values:

Returns VSE time object presenting time interval equal to time / div

Parameters:

time - VSE time object
div - divider, integer value

Example:

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

14 Time logical functions

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

14.1 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:

time_1	- VSE time object presenting first time interval
time_2	- VSE time object presenting second time interval

Example:

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

14.2 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:

time_1	- VSE time object presenting first time interval
time_2	- VSE time object presenting second time interval

Example:

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

14.3 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:

time_1	- VSE time object presenting first time interval
time_2	- VSE time object presenting second time interval

Example:

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

14.4 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:

time_1	- VSE time object presenting first time interval
time_2	- VSE time object presenting second time interval

Example:

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

15 Time text functions

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

15.1 TimeToText()

Converts a VSE time object into text.

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

Return values:

Returns text representation of VSE time object

Parameters:

time - VSE time object

Example:

```
t = Time(100);
ReportText( TimeToText( t ) ); # see below details for ReportText() function
```

16 Output functions

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

16.1 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 = 0xBBBB;
text = FormatEx( “x = 0x%04X,  y = 0x%04X”, x, y );
ReportText( “Text : ” + text );
…
```

16.2 EnableOutput()

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

Format : **EnableOutput ()**

Example:

```
EnableOutput ( );
```

16.3 DisableOutput()

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

Format : **DisableOutput ()**

Example:

```
DisableOutput ();
```

17 Common Retrieving functions

This group of functions covers VSE capability to retrieve information from the recorded trace.

17.1 GetTraPayloadLimit ()

Returns the maximum value of the transaction payload in bytes stored in in.Payload input context member.

Format : **GetTraPayloadLimit ()**

Example:

```
If( GetTraPayloadLimit () < SomeValue )
        SetTraPayloadLimit(  SomeValue ); # ensure that we will get a big enough
                                          # payload.

val = GetNBits( in.Payload, 128, 8 ); # retrieve one byte from frame/sequence payload
                                      # starting from offset 16 bytes
```

17.2 SetTraPayloadLimit ()

Sets up the maximum value of the transaction payload in bytes stored in in.Payload input context member.

Format : **SetTraPayloadLimit (tra_payload_limit)**

Parameters:

tra_payload_limit - New value of transaction payload limit.

Example:

```
If( GetTraPayloadLimit () < SomeValue )
        SetTraPayloadLimit(  SomeValue ); # ensure that we will get a big enough
                                          # payload.

val = GetNBits( in.Payload, 128, 8 ); # retrieve one byte from frame/sequence payload
                                      # starting from offset 16 bytes
```

17.3 IsFcp()

Verifies that current event is FCP SCSI frame or sequence

Format : **IsFcp()**

Example:

```
if( IsFcp() )  DoSomething();
```

18 FC data frame retrieving functions

This group of functions covers VSE capability to extract information about FC data frame headers other than Frame Header. Frame Header fields are provided as members of input context for data frame trace event. (See Data frame-specific set of members for details)

If either header or field is not present in the frame – all of those functions will return null-value (see CSL Manual –for details about null-value).

18.1 GetNetHField()

Extracts information about Network header field. (Currently not implemented.)

Format : **GetNetHField (** ***net_fld*** **)**

Parameters:

net_fld - Network header field identifier that can be one of the following values:

identifier	**Meaning**
_D_NAA_H	Network Destination Address (high order bits)
_D_NAA_L	Network Destination Address (low order bits)
_S_NAA_H	Network Source Address (high order bits)
_S_NAA_L	Network Source Address (low order bits)

Example:

```
val  = GetNetHField ( _S_NAA_L  );
```

NOTE : If there are some reserved fields in headers – they can be retrieved by using some special keywords in header retrieving functions :

Keyword	**Length**
_RES_1	1 bit field
_RES_2	2 bit field
_RES_5	5 bit field
_RES_7	7 bit field
_RES_8	8 bit field
_RES_16	16 bit field

19 Script decoded fields retrieving functions

This group of functions covers VSE generic capability to extract information about data payload fields decoded with CATC Script Language (CSL).
(Currently only FCP-SCSI, GS and SW sequence payloads are decoded with CSL)

19.1 GetDecodedScriptField()

Extracts information about script decoded field how it is shown in FCTracer trace view or "View … Fields" dialog.

Format : **GetDecodedScriptField (*fld_name*)**

Parameters:

fld_name - name of the field supposedly existing in the FCP-SCSI, GS or SW sequence being processed:

Return Values:

The text value of the decoded field how it is seen in the trace if the field name asked is present in the current sequence, empty string otherwise.

Example:

```
str  = GetDecodedScriptField ("STATUS");   # extract the decoded value of SCSI
                                            # Status  field.
```

Remark:

The name of field should be exactly the same as it seen in the trace (case sensitive)

19.2 GetHexScriptField()

Extracts raw hexadecimal information about script decoded field.

Format : **GetHexScriptField (*fld_name*)**

Parameters:

fld_name - name of the MAD field supposedly existing in the FCP-SCSI, GS or SW sequence being processed:

Return Values:

If the field with the specified name is present in the current sequence - this function returns the hex value of the decoded field (integer value- if the length of field is less than 32 bits or raw binary value (list of bytes, see CSL manual for further details about raw binary values) - if the length of field is greater than 32 bits), null-value if the field was not found.

Example:

```
val  = GetHexScriptField ( “Logical Block Addr” ); # extract the hex value of LBA field.

# extract the hex value of SomeBig field.
if( GetHexScriptField ( "Some Big" ) ==  'FE80000000000000' )
        ReportText( "Some Big field = FE80-0000-0000-0000");
```

Remark:

The name of field should be exactly the same as it seen in the trace (case sensitive)

20 Information functions

20.1 GetTraceName()

This function returns the filename of the trace file being processed by VSE.

If the script is being run over multi-segmented trace this function will return the path to the 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 filename.

Example:

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

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

20.2 GetScriptName()

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

Format : **GetScriptName()**

Example:

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

20.3 GetApplicationFolder()

This function returns the full path to the folder where the FCTracer application was started.

Format : **GetApplicationFolder()**

Example:

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

20.4 GetCurrentTime()

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

Format : **GetCurrentTime()**

Example:

ReportText(GetCurrentTime()); # will yield "February 10, 2004, 5:49 PM"

20.5 GetEventSegNumber()

In case if a multi-segmented trace is being processed - this function returns the index of the segment for current event.

NOTE: When multi-segmented trace file (extension *.mlt) 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()));

21 Navigation functions

21.1 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 - the transaction level of the event to jump (possible values : _FRM, _SEQ)
index - the transaction index of the event to jump
segment - the segment index of the event to jump. If omitted current segment index will be 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 verification script is running over multi-segmented trace (extension : *.mlt). For regular traces it is ignored.

If wrong parameters were specified (like index exceeding the maximal index for specified transaction level) – the function will do nothing and 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 );
}
```

21.2 SetMarker()

This function sets a marker for some trace event.

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

Parameters:

marker_text - the text of the marker

level	- the transaction level of the event to jump (possible values : _FRM, _SEQ)
index	- the transaction index of the event to jump
segment	- the segment index of the event to jump. If omitted current segment index will be used.

Remarks:

If no parameters were specified other than '*marker_text*' the application will set marker to the current event being processed by VSE. The '*segment*' parameter is used only when verification script is running over multi-segmented trace (extension : *.mlt). For regular traces it is ignored.

If wrong parameters were specified (like index exceeding the maximal index for specified transaction level) – the function will do nothing and 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 );
}
```

22 File functions

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

22.1 OpenFile()

This function opens file for writing.

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

Parameters:

file_path - the full path to the file to open. (For '\' use '\\')

append - this parameter (if present and not equal to 0) specifies that VSE should append following write operations to the contents of the file – otherwise, the contents of the file will be cleared.

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

22.2 CloseFile()

This function closes opened file.

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

Parameters:

file_handle - the file "handle".

Example:

```
…
set file_handle = 0;
…
file_handle  = OpenFile( "D:\\Log.txt" );  # opens file, the previous contents will be
                                            # 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
…
```

22.3 WriteString()

This function writes text string to the file.

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

Parameters:

file_handle - the file "handle".
text_string - the text string".

Example:

```
…
set file_handle = 0;
…
file_handle  = OpenFile( "D:\\Log.txt" );  # opens file, the previous contents will be
                                              # 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
…
```

22.4 ShowInBrowser()

This function allows to open file in the Windows Explorer. If the extension of the file has the application registered to open files with such extensions – it will be 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 – then this file will be opened in the Internet Explorer.

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

Parameters:

file_handle - the 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 );
…
```

23 COM/Automation communication functions

This group of functions covers VSE capabilities to communicate with COM/Automation clients connected to FCTracer application. (Please refer to FCTracer Automation manual for the details how to connect to FCTracer application and VSE)

23.1 NotifyClient()

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

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

Parameters:

param_list - the 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 of the list itself may contain integers, strings or other lists – it is possible to send a complicated messages.
(lists should be treated as arrays of VARIANTs)

Example:

```
…
if( SomeCondition() )
{
        NotifyClient( 2, [ in.Index, in.Level, “CHANNEL 2”, “FCP sequence”,
                    TimeToText( in.Time )] );
}
…
# Here we sent 2 parameters to  clients applications :
#   2 ( integer ),
#  [ in.Index, in.Level, “CHANNEL 2”, “FCP sequence”, TimeToText( in.Time )] ( list )
```

Remark:

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

24 User input functions

24.1 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:

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)

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

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.

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 GUI. For VSEs controlled by COM/Automation clients it does nothing.

This function "locks" the FCTracer 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 case if a script written not correctly – VSE keeps track of all function calls demanding user interaction and doesn't show dialog boxes in case if some customizable limit was exceeded (returns _MB_OK in this case). (See)

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…
  }
```

24.2 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 CSL manual for details about list objects) or 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 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, the text box is displayed empty.

return_type – Optional. It 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 on the *return_type* argument this function returns either 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 divided by ',' (only hexadecimal, decimal and string items are currently supported).
i.e. text :

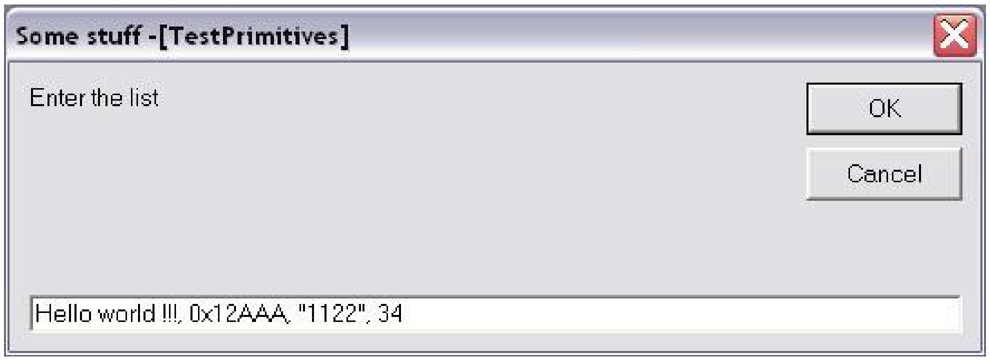
Hello world !!!, 12, Something, 0xAA, 10, "1221"

Will produce a CSL list object (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.



Enter the list

Hello world !!!, 0x12AAA, "1122", 34

Remark:

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

This function "locks" the FCTracer 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 case if a script written not correctly – VSE keeps track of all function calls demanding user interaction and doesn't show dialog boxes in case if some customizable limit was exceeded (returns null object in that case). (See)

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 );
   }
```

24.3 GetUserDlgLimit()

This function returns the current limit of user dialogs allowed in the verification script. If the script reaches this limit no user dialogs will be shown and 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
    …
```

24.4 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 be shown and script will 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
    …
```

25 String manipulation/formating functions

25.1 FormatEx()

Write formatted data to a string. **Format** is used to control the way that arguments will 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:

format_string - Format-control string

argument_list- 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 (-) will cause 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 will insert a plus sign (+) before a positive signed integer. This only works with the conversion characters d and i.

- A space will insert 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 will be ignored.
- A hash mark (#) will prepend a 0 to an octal number when used with the conversion character o. If # is used with x or X, it will prepend 0x or 0X to a hexadecimal number.
- A zero (0) will pad 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"
```

26 Miscellaneous functions

26.1 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 will have a result <DONE> after execution.

Format : **ScriptForDisplayOnly ()**

Example:

```
ScriptForDisplayOnly();
```

26.2 Sleep()

Asks VSE not to send any events to a script until the timestamp of the next event is greater than timestamp of 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
```

26.3 ConvertToHTML()

This function replaces spaces with "&nbsp" and carriage return symbols with "<br>" 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&nbsp!!!<br>How&nbspare&nbspyou&nbsptoday?"
```

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

26.4 Pause()

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

Format : **Pause()**

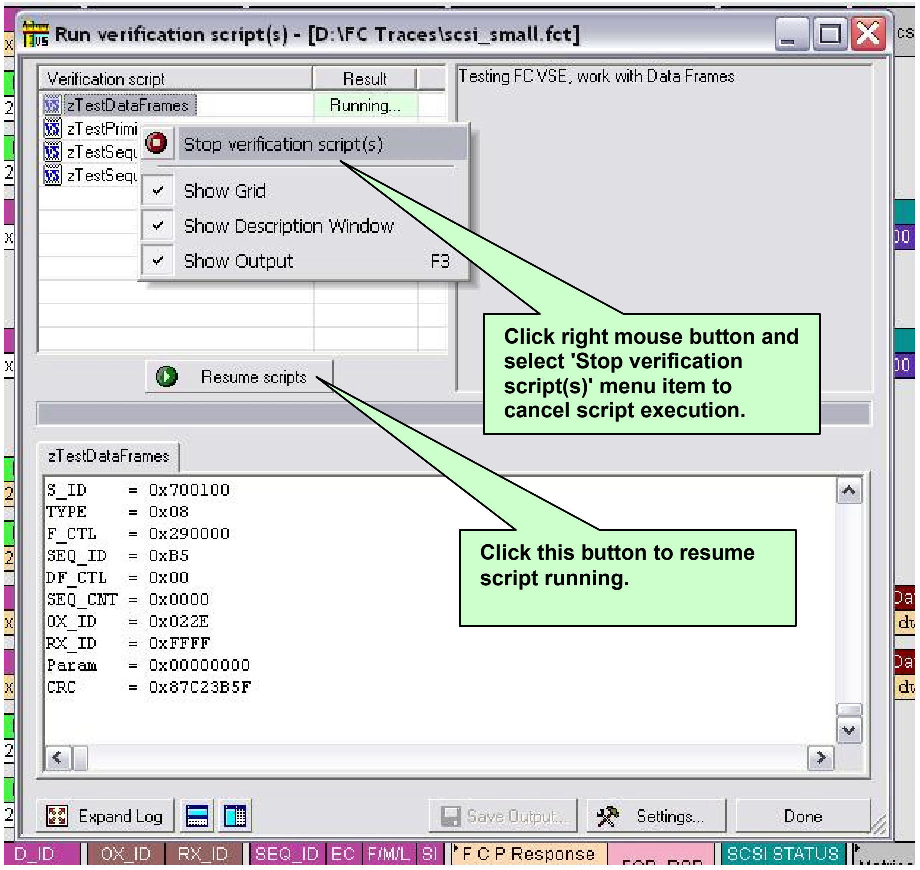
Example:

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

Remark:

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

When script execution is paused – the Run Verification Script – window will look like:



Testing FC VSE, work with Data Frames

```
S_ID    = 0x700100
TYPE    = 0x08
F_CTL   = 0x290000
SEQ_ID  = 0xB5
DF_CTL  = 0x00
SEQ_CNT = 0x0000
OX_ID   = 0x022E
RX_ID   = 0xFFFF
Param   = 0x00000000
CRC     = 0x87C23B5F
```

Click right mouse button and select 'Stop verification script(s)' menu item to cancel script execution.

Click this button to resume script running.

27 The VSE important script files

The VSE working files are located in ..\Scripts\VFScripts subfolder of the main FCTracer folder. The current version of VSE includes following files:

File	Description
VSTools.inc	main VSE file containing definitions of some useful VSE script functions provided by CATC (must be included in every VS)
VS_constants.inc	file containing definitions of some important VSE global constants
VSTemplate.vs_	template file for new verification scripts.
VSUser_globals.inc	file of user global variables and constants definitions (It is useful to put here definitions of constants, variables and functions used in many scripts)

How to Contact CATC

Type of Service	Contract
Call for technical support…	US and Canada: 1 (800) 909-2282 Worldwide: 1 (408) 727-6600
Fax your questions…	Worldwide: 1 (408) 727-6622
Write a letter …	Computer Access Technology Corporation Customer Support 3385 Scott Blvd. Santa Clara, CA 95054
Send e-mail…	support@CATC.com
Visit CATC's web site…	http://www.CATC.com/