PROTOCOL SOLUTIONS GROUP
3385 SCOTT BLVD
SANTA CLARA, CA 95054

Verification Script Engine for LeCroy UWB*Tracer*™ Reference Manual

Manual Version 2.1
For UWB*Tracer* Software Version 2.10

May 2006

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, UWBTracer, UWBTracer MPI, and *UWBTracer Automation* are trademarks of LeCroy.

Microsoft is a registered trademark of Microsoft Inc.

All other trademarks are property of their respective companies.

Copyright

Contents

1 Introduction

This document contains a description of the LeCroy PSG's Verification Script Engine (VSE), a new feature introduced by LeCroy in its UWB*Tracer*™ software that allows users to perform very complicated custom analyses of Ultra-Wideband (UWB) traffic recorded using the new generation of LeCroy Ultra-Wideband protocol analyzers.

The VSE allows users to ask the UWB*Tracer* application to send desired "events" (currently only frame level events are available) that occur in the recorded UWB trace to a special verification script written using LeCroy script language. This script then evaluates the sequence of events (timing, data, or both) in accordance with user-defined conditions and performs post-processing tasks; such as exporting key information to external files (in text or binary format) or sending special Automation™/COM notifications to client applications.

The VSE fully utilizes the high performance and intelligence of LeCroy PSG's decoding capabilities, making processing of information easy and fast. VSE can:

- Retrieve information about any field in UWB frame headers, contents of frame payloads, serial data, and bus states.
- Make complex timing calculations between different events in pre-recorded traces.
- Filter-in or filter-out data with dynamically changing filtering conditions
- Port information to a special output window
- Save data to text or binary files
- Send data to COM clients connected to the UWB*Tracer* application

2 Verification Script Structure

Writing verification scripts is easy, if you understand how the UWB*Tracer*™ application interacts with running scripts and if you follow some rules imposed by the verification script syntax.

The main file with the text of the verification script must have extension **.vse**. It must be located in the subfolder **..\Scripts\VFScripts** of the main UWB*Tracer* folder. Some other files must be included in the main script file using the directive **%include**. (See the *CATC Script Language (CSL) Manual* for details.)

The following schema presents a common structure of a verification script. It is similar to the script template **VSTemplate.vs_**, which is included with VSE.

```
#
#   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                               #
  # are passed to the script.                                                          #
  #                                                                                    #
  #  For details about how to specify the kind of events to pass 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 shows 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 UWB*Tracer* and Verification Script

The following steps describe the interaction between UWB*Tracer*™ and a verification script run over a recorded trace:

1. Before sending any information to the script main processing function **ProcessEvent()** (which must be present in any verification script),VSE looks for function **OnStartScript()** and calls it if it is found. In this function, some setup routines can be made, like specifying the channels, specifying the trace events to pass to the script, setting up initial values of the global script-specific global variables, etc.

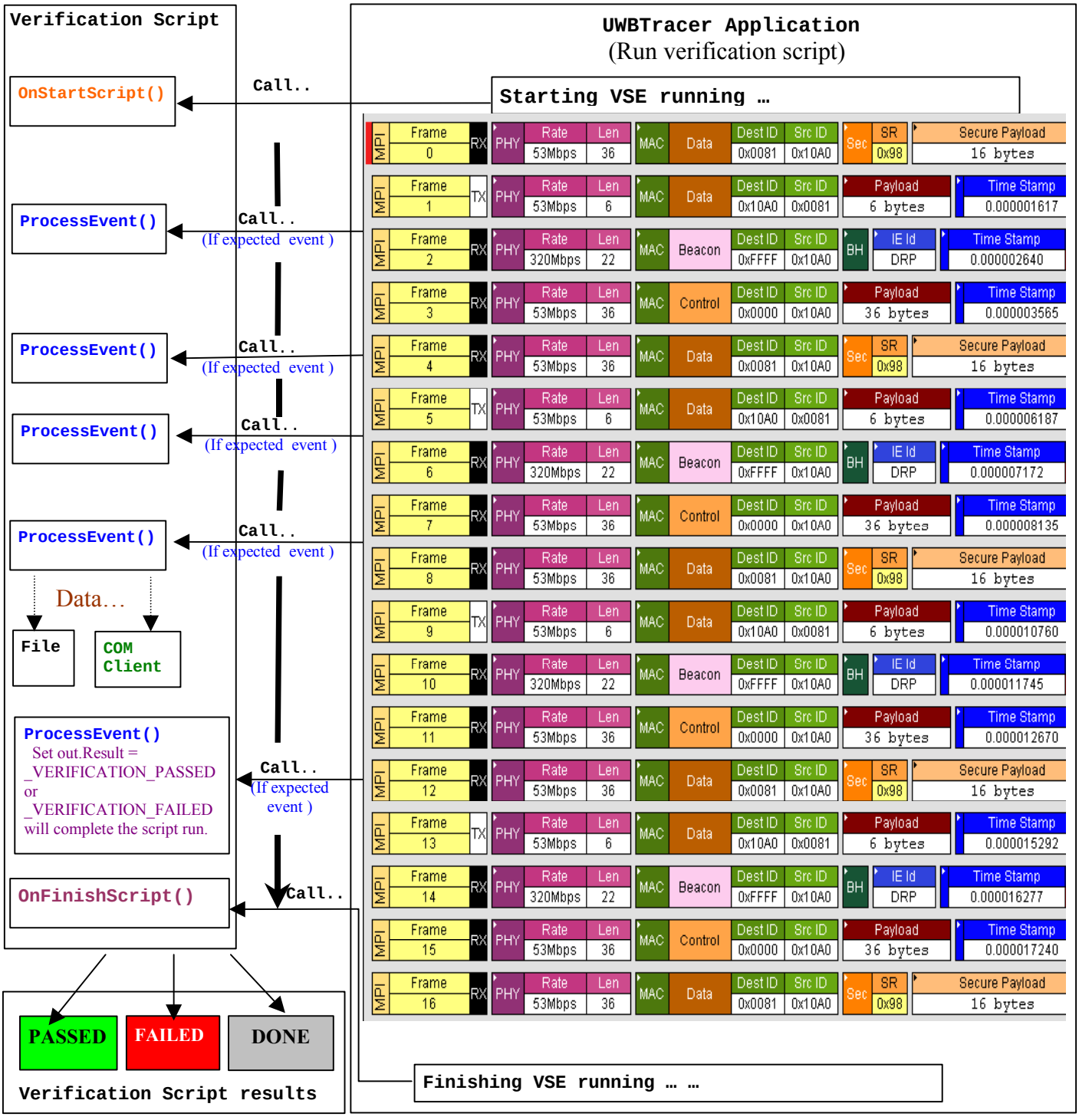
2. Then VSE goes through the recorded trace and checks if the current frame in the trace meets specified sending criteria. If it does, VSE calls the script main processing function **ProcessEvent(),** providing some information about the current event in the script input context variables.
   (See the topic “Input context variables” below in this document for a full description of verification script input context variables.)

3. **ProcessEvent()** is the main verification routine, in which all processing of incoming trace events is done. Basically, when the whole verification program consists of a few stages, this 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. If the whole result is clear on the current stage, it tells VSE to complete execution of the script.
   The completion of the test before the whole trace has been evaluated is usually done by setting the output context variable:
   out.Result = _VERIFICATION_PASSED or **_VERIFICATION_FAILED**.
   (See the topic “Output context variables” below in this document for a full description of verification script output context variables.)

   Note: Not only does a verification script verify recorded traces by some criteria, but it is also possible to extract some information of interest and post-process it later by third-party applications. (There is a set of script functions to save extracted data in text or binary files or send it to other applications via COM/Automation™ interfaces.)

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

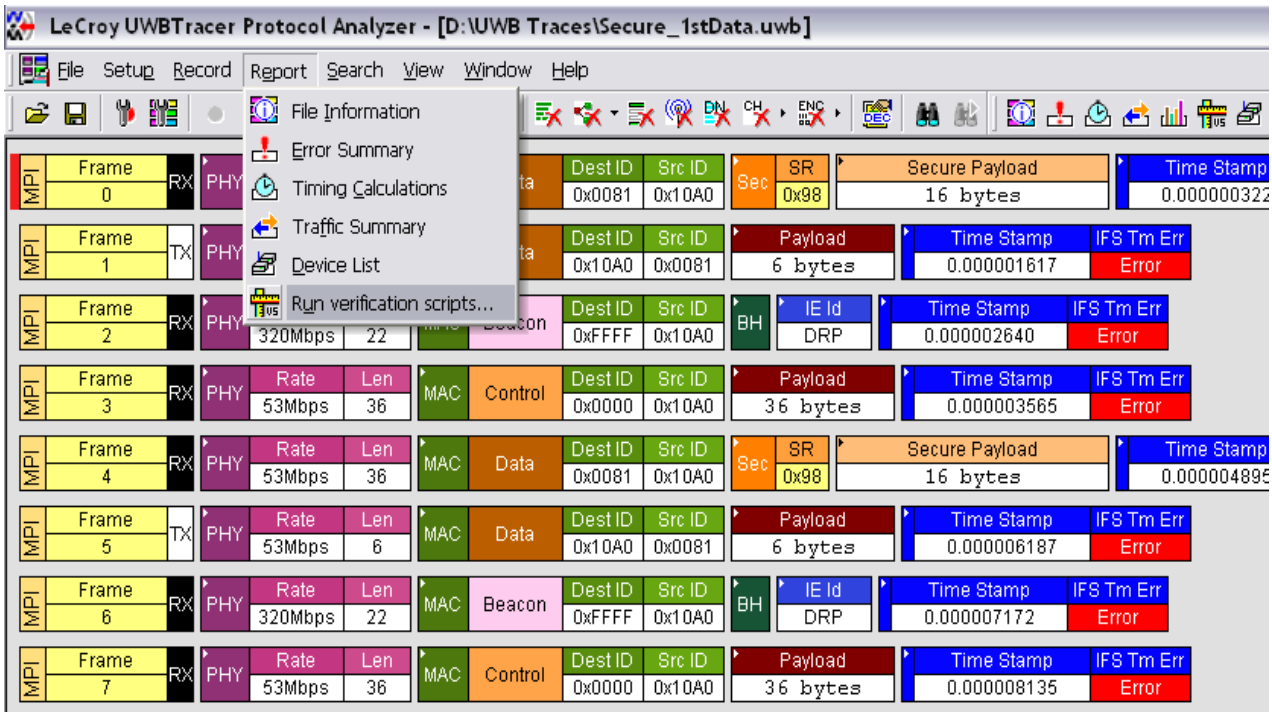
The following picture presents the interaction between the UWB*Tracer* application and a running verification script:



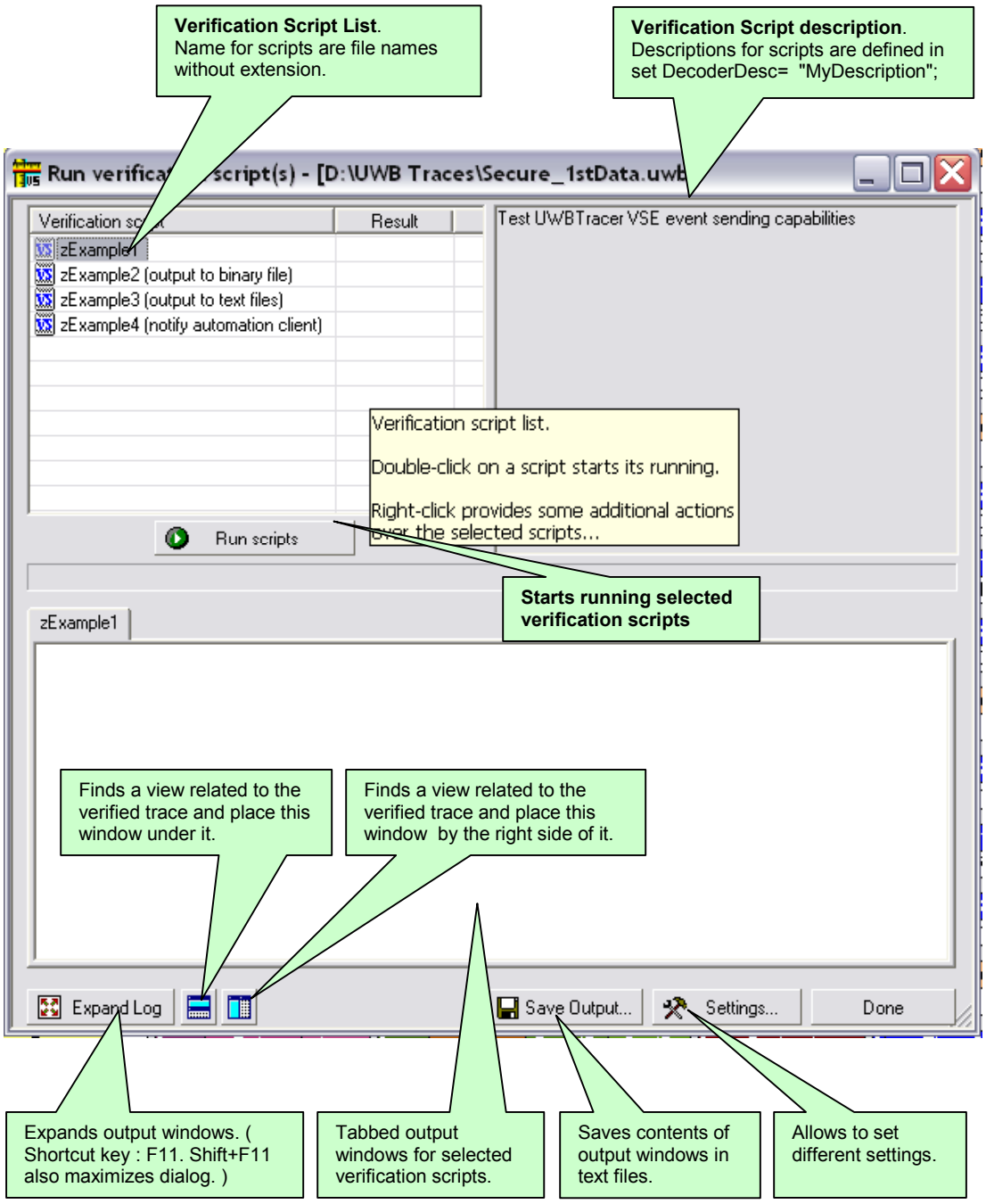
Note: Verification script result **<DONE>** means that the script is intended for extracting and displaying some information from recorded traces only and you do not care about results. To specify that your script is for displaying information only, call somewhere in your script (in **OnStartScript()**, for instance) the function **ScriptForDisplayOnly()**.

4 Running Verification Scripts from UWB*Tracer*

To run a verification script over a trace, you select the UWB*Tracer*™ main menu item **Report > Run verification scripts** or click the **Run verification scripts** button on the main tool bar (if it is not hidden):

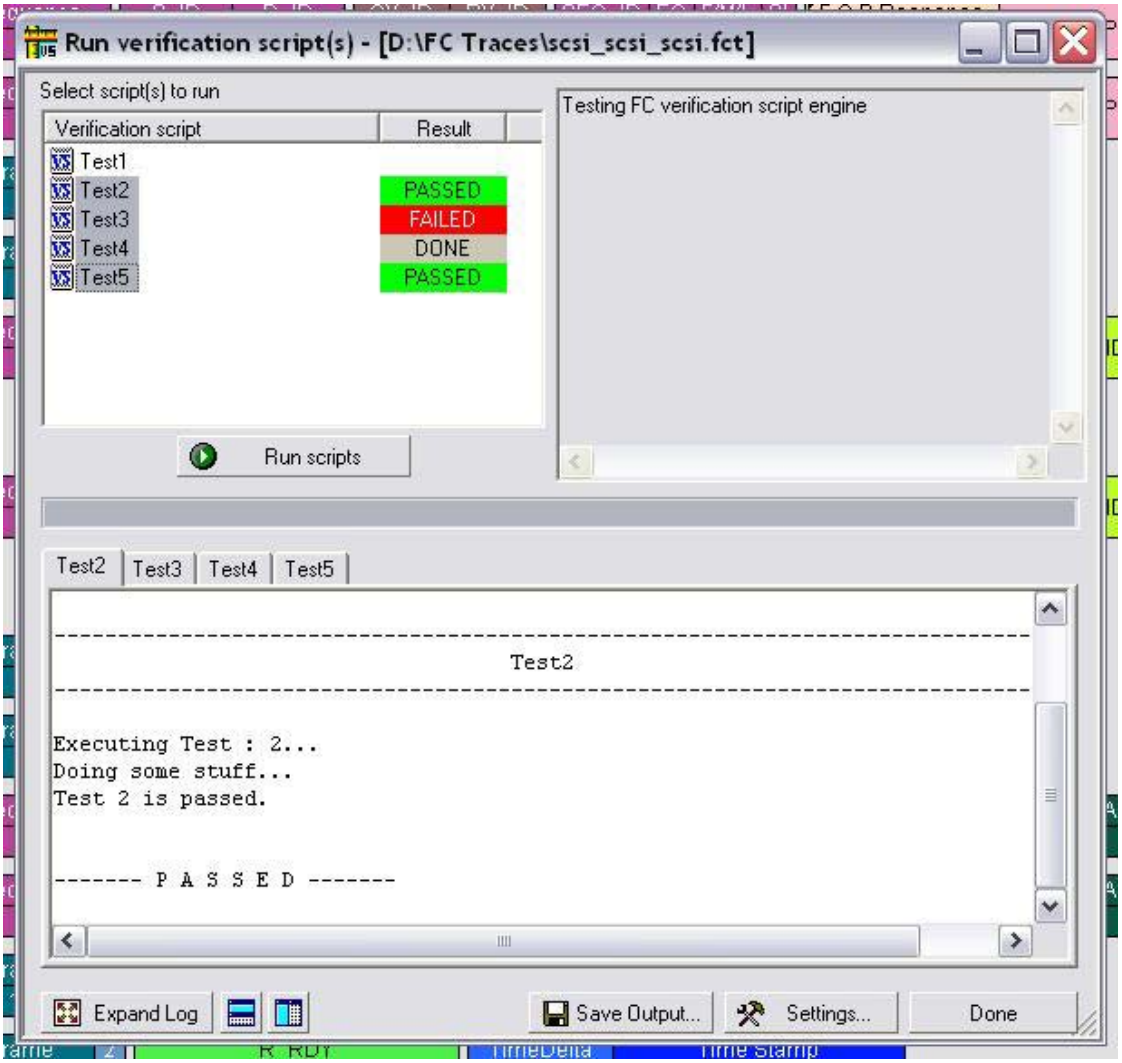


The **Run verification scripts** dialog opens where you choose then run one or several verification scripts:

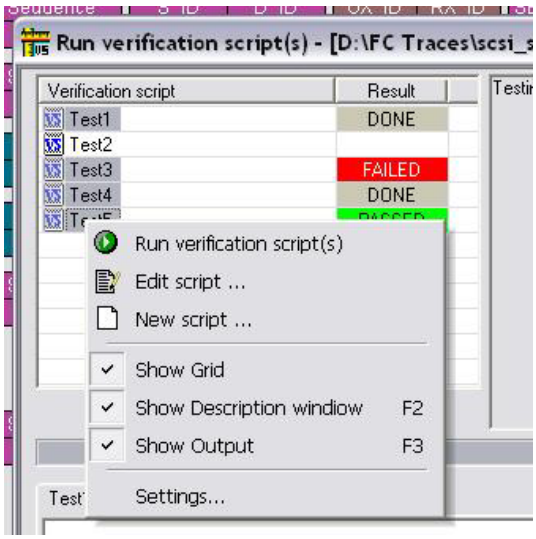


4.1 Running Verification Scripts

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



Right-clicking in the script list displays some additional operations over selected scripts:



`Run verification script(s):` Start running selected script(s).

`Edit script`: Edit selected scripts in the editor application specified in **Editor settings**.

`New script`: Create a new script file using the template specified in **Editor settings**.

`Show Grid:` Show/hide a grid in the verification script list**.**

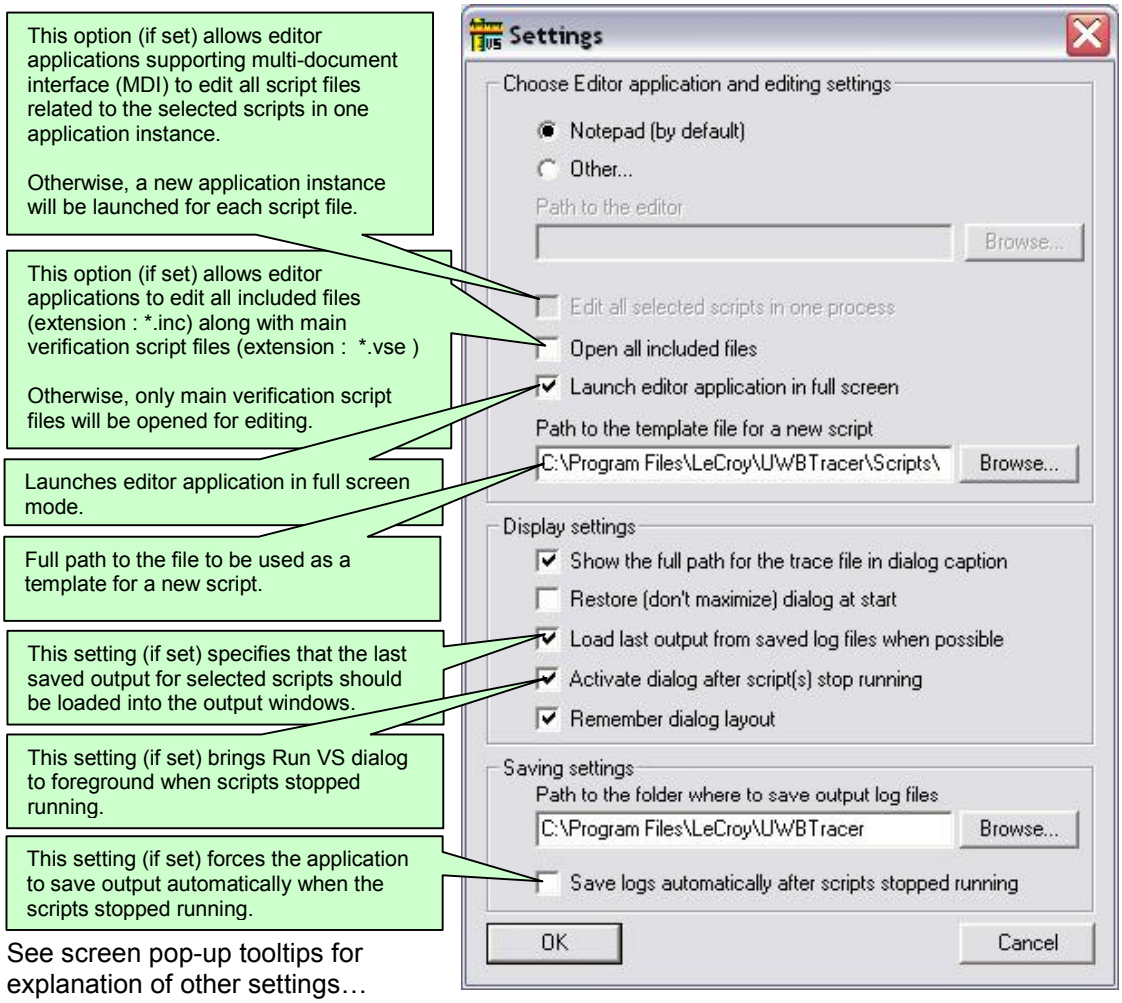
`Show Description window:` Show/hide the script description window **(Shortcut key F2).**

`Show Output:` Show/hide the script output windows **(Shortcut key F3).**

`Settings`: Open a special **Setting** dialog to specify different settings for VSE.

4.2 VSE GUI Settings

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



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

5 Verification Script Engine Input Context Members

All verification scripts have input contexts, special structures that can be used inside of the scripts. The application fills their members. (For more information about input contexts, please refer to the *LeCroy PSG 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 a script:

`in.Level`: Transaction level of the trace event (0 is frames, no other levels are currently supported).

`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]. For more information, see section 10.1 VSE Time Object.

`in.Channel`: Indicates on which channel the event occured. It can be one of the following constants:
_MPI: MPI channel, not decripted traffic (value = 1)
_RF: RF channel, not decrypted traffic (value = 2)
_MPI_DEC: MPI channel, decrypted traffic (value =: 3)
_RF_DEC: RF channel, decrypted traffic (value = 4) from **_CHANNEL_1 (= 1)** to **_CHANNEL_4 (= 4)**

`in.WirelessChannel`: Indicates wireless channel (valid only for RF channels) where the event occured.

`in.Receive`: Indicates PHY direction (0 = TX, 1 = RX).

`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 event or after calling functions that provide values of variables:

5.2.1 BusState-specific Set of Members

Members of this set are valid for Bus State trace events only:

`in.BusState`: Type of **Bus State** events

The table below describes the current list of possible Bus State events and values of **in.BusState**:

Bus State	in.BusState
PHY RESET	_RESET
PHY SLEEP	_SLEEP
PHY STANDBY	_ STANDBY
PHY READY	_ READY

5.2.2 Serial Data-specific Set of Members

(valid for Serial Data trace events only, undefined for other events)

`in.PhyDirection`: Serial Data direction (can be either **_READ** or **_WRITE**)

`in.PhyRegister`: Serial Data PHY register

`in.PhyValue`: Serial Data PHY value

5.2.3 Data frame-specific set of members

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

Names of frame-specific input context members basically follow a rule that simplifies writing script code accessing the frame fields:

- For fields less than or equal to DWORD (32 bits), input context member names are exactly the same as frame field names in the UWB*Tracer*™ trace view.
- For fields greater than DWORD (but less than 64 bits), two input context members for low and high DWORDS are provided:
  `in.Field_Lo:` Contains the lower DWORD (32 bits) part of the field
  `in.Field_Hi:` Contains the upper part of the field.
- Spaces in field names (as they appear in trace view cells) are replaced with '**_**'
  for example, for field **TF Code** there is a input context member **TF_Code**
- '**#**' in field names (as they appear in trace view cells) is replaced with **_Num**
  for example, for field **Frag#** there is a input context member **Frag_Num**

5.2.4 PHY Header Members

in.Rate: Frame data rate

in.Len: Length of frame payload

in.Scr: Scrambler Init

in.BM: Burst Mode

in.PreType: PLCP Preamble type (short or long)

in.TF_Code: TF Code

in.BG: Least significant bit of the Band Group member

in.PHY_Resvd_Oct_0: PHY Header reserved octet 0 (3 bits)

in.PHY_Resvd_Oct_2: PHY Header reserved octet 2 (2 bits)

in.PHY_Resvd_Oct_3: PHY Header reserved octet 3 (2 bits)

in.PHY_Resvd_Oct_4: PHY Header reserved octet 4 (8 bits)

in.Hdr_Err: PHY Header errors

in.Hdr_Err_Unused: Unused bits in the PHY Header error report

in.RSSI: RSSI reported by PHY

in.LQI: LQI reported by PHY

in.Rx_Err: Rx error report bits

in.Rx_Err_Unused: unused bits in Rx Error report

5.2.5 MAC Header Members

in.Dest_ID: Destination ID

in.Src_ID: Source ID

in.Policy: ACK Policy

in.Rtry: Retry

in.Type: Frame Type

in.SubType: Frame Subtype

in.Delivery_ID: Delivery ID

in.Sec: Secure

in.Vers: Protocol Version

in.MAC_FC_Reserved: Reserved

in.Frag_Num: Fragment Number

in.DU_Num: MSDU or MCDU Number

in.M_Frg: More Fragments

in.SC_Reserved: Reserved

in.Duration: Duration

in.M_Dat: More Frames

in.Acc: Access Method

5.2.6 Secure Header Members

(valid only for secure frames, when Secure field in MAC header is set to 1)

`in.TKID`: Temporal Key Identifier

`in.SR`: Reserved field in Secure header

`in.EO`: Encryption offset in secure payload

`in.SFN_Lo`: Low DWORD(32 bits) of Secure Frame Number (SFN)

`in.SFN_Hi`: High WORD (16 bits) of Secure Frame Number (SFN)

`in.MIC_Lo`: Low DWORD of Message Integrity Code (MIC)

`in.MIC_Hi`: High DWORD of Message Integrity Code (MIC)

5.2.7 Miscellaneous Members

`in.FCS`: Frame Checksum

`in.Frm_Duration`: Frame duration (type = list, having format = 2 sec 125 ns -> [2 , 125].
(See section 10.1 VSE Time Object for details.)

`in.PHY_ACT`: PHY Active signal duration (type = list, having format = 2 sec 125 ns -> [2 , 125].
(See section 10.1 VSE Time Object for details.)

5.2.8 Payload-related Members

`in.Payload`: Bit source of the frame payload. (You can extract any necessary information using **GetNBits(), NextNBits()**, or **PeekNBits()** function. (See the *CSL Manual* for details about these functions.)

`in.PayloadLength`: Length(in bytes) of the retrieved frame payload.

Note:
For not-decrypted channels (**_MPI, _RF**), the payload is not decrypted.
For decrypted channels (**MPI_DEC, RF_DEC**), the payload is decrypted if decryption was successful.

6 Verification Script Engine Output Context Members

All verification scripts have output contexts, special structures that can be used inside of the application. The scripts fill their members. (For more information about output contexts, see the *LeCroy PSG 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 can have three values:

- **_VERIFICATION_PROGRESS**: Set by default when a script starts running.
- **_VERIFICATION_PASSED**
- **_VERIFICATION_FAILED**

The last two values should be set if you decide that a 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 to **VERIFICATION_FAILED** at exit.

Note: If you don’t care about the result of script running, call the function ScriptForDisplayOnly() once before stopping the script. The result is **DONE**.

7 Verification Script Engine Events

VSE defines a large group of trace “events” – currently on the frame level only – that can be passed to a verification script for evaluating, retrieving, and displaying some contained information. The information about the type of event can be seen in the in.TraceEvent input context member.

See the topic “Sending Functions” in this manual for details about how to specify the events to send to verification scripts.

8 Frame Level Events

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

Types of frames	**in.TraceEvent**
PHY Bus State change	_BUS_STATE
Serial Data	_SERIAL_DATA
Frames	_FRAMES
Beacon frames	_FRM_BEACON
Control frames	_FRM_CONTROL
Command frames	_FRM_COMMAND
Data frames	_FRM_DATA
Aggregated Data frames	_FRM_AGG_DATA
Other frames	_FRM_OTHER

9 Sending Functions

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

9.1 SendLevel()

This function specifies that events of specified transaction level should be sent to the script. Currently only frame level events can be sent to verification scripts.

Format: `SendLevel(level)`

Parameters:

level This parameter can be one of following values:
_FRM:(value 0): Send Frame level events.

Remark:

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

Example:

```
...
SendLevel(_FRM); # Send frame level events.
```

9.2 SendLevelOnly()

This function specifies that ONLY events of a specified transaction level should be sent to the script. Currently only frame level events can be sent to verification scripts.

Format: `SendLevelOnly(level)`

Parameters:

level This parameter can be one of following values:
_FRM (value 0): Send Frame level events

Example:

```
…
SendLevelOnly(_FRM); # Send ONLY frame level events.
```

9.3 DontSendLevel()

This function specifies that events of a specified transaction level should NOT be sent to the script. Currently only frame level events can be sent to verification scripts.

Format: `DontSendLevel(level)`

Parameters:

level	This parameter can be one of following values: **_FRM**: DO NOT send Frame level events.

Example:

```
…
DontSendLevel(_FRM); # DO NOT send frame level events.
```

9.4 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:
_MPI (= 1): Send events from MPI channel
_RF (= 2): Send events from RF channel
_MPI_DEC (= 3): Send events from MPI decrypted channel
_RF_DEC (= 4): Send events from RF decrypted channel
_CHANNEL_1 (= 1): Send events from channel 1 (MPI)
_CHANNEL_2 (= 2): Send events from channel 2 (RF)
_CHANNEL_3 (= 3): Send events from channel 3 (MPI decrypted)
_CHANNEL_4 (= 4): Send events from channel 4 (RF decrypted)

Example:

```
SendChannel(_RF);          # Send events from RF channel.
SendChannel(_MPI_DEC); # Send events from MPI decrypted channel.
…
SendChannel(_CHANNEL_2); # Send events from channel 2 (RF).
SendChannel(_CHANNEL_3); # Send events from channel 3 (MPI_DEC).
…
SendChannel(2); # Send events from channel 2 (RF).
SendChannel(3); # Send events from channel 3 (MPI_DEC).
…
```

9.5 SendChannelOnly()

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

Format: `SendChannelOnly(channel)`

Parameters:

channel This parameter can be one of following values:
_MPI (= 1): Send ONLY events from MPI channel
_RF (= 2): Send ONLY events from RF channel
_MPI_DEC (= 3): Send ONLY events from MPI decrypted channel
_RF_DEC (= 4): DSend ONLY events from RF decrypted channel
_CHANNEL_1 (= 1): Send ONLY events from channel 1 (MPI)
_CHANNEL_2 (= 2): Send ONLY events from channel 2 (RF)
_CHANNEL_3 (= 3): Send ONLY events from channel 3 (MPI decrypted)
_CHANNEL_4 (= 4): Send ONLY events from channel 4 (RF decrypted)

Example:

```
SendChannelOnly (_RF);          # Send ONLY events from RF channel.
SendChannelOnly (_MPI_DEC); # Send ONLY events from MPI decrypted channel.
…
SendChannelOnly (_CHANNEL_2); # Send ONLY events from channel 2(RF).
SendChannelOnly (_CHANNEL_3); # Send ONLY events from channel 3(MPI_DEC).
…
SendChannelOnly  (2); # Send ONLY events from channel 2 (RF).
SendChannel Only (3); # Send ONLY events from channel 3 (MPI_DEC).
…
```

9.6 DontSendChannel()

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

Format: `DontSendChannel (channel)`

Parameters:

channel This parameter can be one of following values:
_MPI (= 1): DO NOT send events from MPI channel
_RF (= 2): DO NOT send events from RF channel
_MPI_DEC (= 3): DO NOT send events from MPI decrypted channel
_RF_DEC (= 4): DO NOT send events from RF decrypted channel
_CHANNEL_1 (= 1): DO NOT send events from channel 1 (MPI)
_CHANNEL_2 (= 2): DO NOT send events from channel 2 (RF)
_CHANNEL_3 (= 3): DO NOT send events from channel 3 (MPI decrypted)
_CHANNEL_4 (= 4): DO NOT send events from channel 4 (RF decrypted)

Example:

```
DontSendChannel(_RF);          # DO NOT send events from RF channel.
DontSendChannel(_MPI_DEC); # DO NOT send events from MPI decrypted channel.
…
DontSendChannel(_CHANNEL_2); # DO NOT send events from channel 2 (RF).
DontSendChannel(_CHANNEL_3); # DO NOT send events from channel 3.
                                          # (MPI_DEC)
…
DontSendChannel(2); # DO NOT send events from channel 2 (RF).
DontSendChannel(3); # DO NOT send events from channel 3 (MPI_DEC).
…
```

9.7 SendAllChannels()

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

Format: `SendAllChannels ()`

Example:

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

9.8 SendWirelessChannel()

This function specifies that events that occur ONLY on some specific wireless channel should be sent to the script. This function works only for RF (**_RF** and **_RF_DEC**) channels and displays trace events only for a specific wireless channel.

Format: `SendWirelessChannel(wireless_channel)`

Example:

```
...
SendChannel (_RF_DEC);  # Send events from RF decrypted channel.
SendWirelessChannel (10); # Send events only from wireless channel 10.
```

9.9 SendTraceEvent()

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

Format: `SendTraceEvent(event)`

Parameters:

event This parameter may have one of the following values:

Frame level events

Event Value	Description
_BUS_STATE	PHY Bus State change
_SERIAL_DATA	Serial Data
_FRAMES	All frames
_FRM_BEACON	Beacon frames
_FRM_CONTROL	Control frames
_FRM_COMMAND	Command frames
_FRM_DATA	Data frames
_FRM_AGG_DATA	Aggregated Data frames
_FRM_OTHER	Other frames

Example:

```
…
SendTraceEvent(_BUS_STATE);     # Send Bus State trace events.
SendTraceEvent(_SERIAL_DATA); # Send Serial Data trace events.
SendTraceEvent(_FRAMES);          # Send all frames.
…
SendTraceEvent(_FRM_BEACON)  # Send Beacon frames.
```

9.10 DontSendTraceEvent()

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

Format: `DontSendTraceEvent (event)`

Parameters:

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

Example:

```
…
SendTraceEvent(_FRAMES); # Send all frames.

…
if(SomeCondition)
{
        DontSendTraceEvent (_FRM_DATA);         # Don't send Data frames.
        DontSendTraceEvent (_FRM_CONTROL);   # Don't send Control frames.
        DontSendTraceEvent (_FRM_COMMAND); # Don't send Command frames.
        DontSendTraceEvent (_FRM_AGG_DATA); # Don't send Agg Data frames.
        DontSendTraceEvent (_FRM_OTHER);        # Don't send Other frames.
        # Only Beacon frames are sent.
}
```

9.11 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 there are many events to be sent, to send only one kind of event and not send the other ones.

Example:

```
…
SendTraceEvent(_FRAMES); # Send all frames.
…
if(SomeCondition)
{
        SendTraceEventOnly (_FRM_BEACON);
        # Only Beacon frames are sent.
}
```

9.12 SendAllTraceEvents()

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

Format: `SendAllTraceEvents ()`

Example:

```
...
SendAllTraceEvents ();  # Send all frame level trace events.
```

9.13 SendBusStates()

This function specifies more precise tuning for sending Bus State events. Only specified Bus States are sent.

Format: `SendBusStates (bus_state)`

Parameters:

bus_state Bus state parameter. This parameter may have one of the following values:

Primitive Values

Constant	Bus State
_RESET	PHY Reset
_SLEEP	PHY Sleep
_STANDBY	PHY StandBy
_READY	PHY Ready
_ANY	Any Bus State

If this parameter parameter is omitted, all Bus State trace events are sent, which is the same as:
`SendTraceEvent(_BUS_STATE);`

Example:

```
SendBusState(_RESET); # Send 'RESET' bus state.
SendBusState();       # Send all Bus State events.
SendBusState(_ANY);   # Send all Bus State events.
```

9.14 SendSerialData()

This function specifies more precise tuning for Serial Data events. Only selected Serial Data are sent.

Format: `SendSerialData (direction, register, value)`

Parameters:

direction This parameter specifies Serial Data direction. This parameter can have one of the following values:

Direction	Meaning
_READ	Serial Data Read
_WRITE	Serial Data Write
_ANY	Any direction

register This list parameter specifies PHY register.

value This list parameter specifies PHY value.

Note: You can use constant **_ANY** as a parameter 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 there is no parameter at all, all Serial Data events are sent, which is the same as **SendTraceEvent(_SERIAL_DATA))**

Example:

```
# Send only READ Serial Data events (with any register and value).
SendSerialData(_READ);
…
# Send only WRITE Serial Data events, with register = 0 and any value.
SendSerialData(_WRITE, 0);
…
# Send only WRITE Serial Data events, with register = 0 and value = 0x15.
SendSerialData(_WRITE, 0, 0x15);
…
# Send Serial Data events, with any register and value = 0x15.
SendSerialData(_ANY, _ANY, 0x15);
…
# Send all Serial Data events.
SendSerialData();
```

9.15 SendBeaconFrames()

This function specifies more precise tuning for Beacon frames.

Format: `SendBeaconFrames (dest_id, src_id, subtype)`

Parameters:

dest_id This parameter specifies that only Beacon frames containing Destination Id with the same value are sent. (The value**_ANY** means any Destination Id is accepted.)

src_id This parameter specifies that only Beacon frames containing Source Id with the same value are sent. (The value**_ANY** means any Source Id is accepted.)

sub_type This parameter specifies that only Beacon frames having SubType field with the same value are sent. (The value**_ANY** means any SubType is accepted.)

Example:

```
# Send any Beacon frames.
SendBeaconFrames();

# Send Beacon frames having Destination Id 0xBEEF.
SendBeaconFrames(0xBEEF);

# Send Beacon frames having Destination Id =  0xBEEF and Source Id = 0xAABB.
SendBeaconFrames(0xBEEF, 0xAABB);

# Send Beacon frames having Frame SubType = 14.
SendBeaconFrames(_ANY, _ANY, 14);
```

9.16 SendControlFrames()

This function specifies more precise tuning for Control frames.

Format: `SendControlFrames(dest_id, src_id, subtype)`

Parameters:

dest_id This parameter specifies that only Control frames containing Destination Id with the same value are sent. (The value**_ANY** means any Destination Id is accepted.)

src_id This parameter specifies that only Control frames containing Source Id with the same value are sent. (The value**_ANY** means any Source Id is accepted.)

sub_type This parameter specifies that only Control frames having SubType field with the same value are sent. (The value**_ANY** means any SubType is accepted.)

Example:

```
# Send any Control frames.
SendControlFrames();

# Send Control frames having Destination Id 0xBEEF.
SendControlFrames (0xBEEF);

# Send Control frames having Destination Id =  0xBEEF and Source Id = 0xAABB.
SendControlFrames (0xBEEF, 0xAABB);

# Send Control frames having Frame SubType = CF_APPLICATION (14).
# CF_APPLICATION and other Control Frame subtype constants are defined
# in "VS_constants.inc" file.
SendControlFrames (_ANY, _ANY, CF_APPLICATION);
```

9.17 SendCommandFrames()

This function specifies more precise tuning for Command frames.

Format: `SendCommandFrames(dest_id, src_id, subtype)`

Parameters:

dest_id This parameter specifies that only Command frames containing Destination Id with the same value are sent. (The value**_ANY** means any Destination Id is accepted.)

src_id This parameter specifies that only Command frames containing Source Id with the same value are sent. (The value**_ANY** means any Source Id is accepted.)

sub_type This parameter specifies that only Command frames having SubType field with the same value are sent. (The value**_ANY** means any SubType is accepted.)

Example:

```
# Send any Command frames.
SendCommandFrames();

# Send Command frames having Destination Id 0xBEEF.
SendCommandFrames (0xBEEF);

# Send Command frames having Destination Id =  0xBEEF and Source Id = 0xAABB.
SendCommandFrames (0xBEEF, 0xAABB);

# Send Command frames having Frame SubType = 14.
SendCommandFrames (_ANY, _ANY, 14);
```

9.18 SendDataFrames()

This function specifies more precise tuning for Data frames.

Format: `SendDataFrames(dest_id, src_id, subtype)`

Parameters:

dest_id This parameter specifies that only Data frames containing Destination Id with the same value are sent. (The value**_ANY** means any Destination Id is accepted.)

src_id This parameter specifies that only Data frames containing Source Id with the same value are sent. (The value**_ANY** means any Source Id is accepted.)

sub_type This parameter specifies that only Data frames having SubType field with the same value are sent. (The value**_ANY** means any SubType is accepted.)

Example:

```
# Send any Data frames.
SendDataFrames();

# Send Data frames having Destination Id 0xBEEF.
SendDataFrames (0xBEEF);

# Send Data frames having Destination Id =  0xBEEF and Source Id = 0xAABB.
SendDataFrames (0xBEEF, 0xAABB);

# Send Data frames having Frame SubType = 14.
SendDataFrames (_ANY, _ANY, 14);
```

9.19 SendAggDataFrames()

This function specifies more precise tuning for Aggregated Data frames.

Format: `SendAggDataFrames(dest_id, src_id, subtype)`

Parameters:

dest_id This parameter specifies that only Aggregated Data frames containing Destination Id with the same value are sent. (The value**_ANY** means any Destination Id is accepted.)

src_id This parameter specifies that only Aggregated Data frames containing Source Id with the same value are sent. (The value**_ANY** means any Source Id is accepted.)

sub_type This parameter specifies that only Aggregated Data frames having SubType field with the same value are sent. (The value**_ANY** means any SubType is accepted.)

Example:

```
# Send any Aggregated Data frames.
SendAggDataFrames();

# Send Aggregated Data frames having Destination Id 0xBEEF.
SendAggDataFrames (0xBEEF);

# Send Aggregated Data frames having Destination Id =  0xBEEF and Source Id =
# 0xAABB.
SendAggDataFrames (0xBEEF, 0xAABB);

# Send Data Aggregated frames having Frame SubType = 14.
SendAggDataFrames (_ANY, _ANY, 14);
```

9.20 SendOtherFrames()

This function specifies more precise tuning for "other" frames (having a reserved frame type).

Format: `SendOtherFrames(dest_id, src_id, subtype)`

Parameters:

dest_id This parameter specifies that only "other" frames containing Destination Id with the same value are sent. (The value**_ANY** means any Destination Id is accepted.)

src_id This parameter specifies that only "other" frames containing Source Id with the same value are sent. (The value**_ANY** means any Source Id is accepted.)

sub_type This parameter specifies that only "other" frames having SubType field with the same value are sent. (The value**_ANY** means any SubType is accepted.)

Example:

```
# Send any "other" frames.
SendOtherFrames();

# Send "other" frames having Destination Id 0xBEEF.
SendOtherFrames (0xBEEF);

# Send "other" frames having Destination Id =  0xBEEF and Source Id = 0xAABB.
SendOtherFrames (0xBEEF, 0xAABB);

# Send "other" frames having Frame SubType = 14.
SendOtherFrames (_ANY, _ANY, 14);
```

10 Timer Functions

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

10.1 VSE Time Object

A VSE time object presents time intervals in verification scripts. From the point of view of CSL, the verification script time object is a “list” object of two elements.

```
[seconds, nanoseconds]
```

(See the *CSL Manual* for more details about CSL types.)

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

10.2 SetTimer()

Starts a timing calculation from the event at which 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 the calculation again from the point where it was called.

10.3 KillTimer()

Stops a 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.
```

10.4 GetTimerTime()

Retrieves a timing interval from the specific timer.

Format: `GetTimerTime (timer_id = 0)`

Parameters:

timer_id Unique timer identifier.

Return values:

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

Example:

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

Remark:

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

11 Time Construction Functions

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

11.1 Time()

Constructs a 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
neconds	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**.

12 Time Calculation Functions

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

12.1 AddTime()

Adds two VSE time objects.

Format: `AddTime(time1, time2)`

Return values:

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

Parameters:

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

Example:

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

12.2 SubtractTime()

Subtracts two VSE time objects.

Format: `SubtractTime (time1, time2)`

Return values:

Returns a VSE time object representing a 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.
```

12.3 MulTimeByInt()

Multiplies a VSE time object by an integer value.

Format: `MulTimeByInt (time, mult)`

Return values:

Returns a VSE time object representing a time interval equal to **time * mul**t

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.
```

12.4 DivTimeByInt()

Divides a VSE time object by an integer value.

Format: `DivTimeByInt (time, div)`

Return values:

Returns a VSE time object representing 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.
```

13 Time Logical Functions

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

13.1 IsEqualTime()

Verifies that one VSE time object is equal to another VSE time object.

Format: `IsEqualTime (time1, time2)`

Return values:

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

Parameters:

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

Example:

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

13.2 IsLessTime()

Verifies that one VSE time object is less than another VSE time object.

Format: `IsLessTime (time1, time2)`

Return values:

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

Parameters:

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

Example:

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

13.3 IsGreaterTime()

Verifies that one VSE time object is greater than another VSE time object.

Format: `IsGreaterTime (time1, time2)`

Return values:

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

Parameters:

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

Example:

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

13.4 IsTimeInInterval()

Verifies that a VSE time object is greater than a minimum VSE time object and less than a maximum VSE time object.

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

Return values:

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

Parameters:

min_time	VSE time object representing a minimum interval
time	VSE time object representing a time interval
max_time	VSE time object representing a maximum interval

Example:

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

14 Time Text Functions

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

14.1 TimeToText()

Converts a VSE time object into text.

Format: `TimeToText (time)`

Return values:

Returns a text representation of a VSE time object.

Parameters:

time VSE time object

Example:

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

15 Output Functions

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

15.1 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);
…
```

15.2 EnableOutput()

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

Format: `EnableOutput ()`

Example:

```
EnableOutput ();
```

15.3 DisableOutput()

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

Format: `DisableOutput ()`

Example:

```
DisableOutput ();
```

16 Information Functions

16.1 GetTraceName()

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

Format: `GetTraceName(filepath_compatible)`

Parameters:

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

Example:

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

# For trace file with path "D:\Some UWB Traces\Data.uwb"
# GetTraceName(1) returns "D_Some UWB Traces_Data.uwb"
```

16.2 GetScriptName()

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

Format: `GetScriptName()`

Example:

```
ReportText(“Current script = ” + GetScriptName());
```

16.3 GetApplicationFolder()

This function returns the full path to the folder where the UWB*Tracer*™ application was started.

Format: `GetApplicationFolder()`

Example:

```
ReportText("UWBTracer folder = " + GetApplicationFolder ());
```

16.4 GetCurrentTime()

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

Format: `GetCurrentTime()`

Example:

```
# Yields current time in format “May 18, 2006, 5:49 PM”
ReportText(GetCurrentTime());
```

17 Navigation Functions

17.1 GotoEvent()

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

Format: `GotoEvent(level, index)`
`GotoEvent()`

Parameters:

level Transaction level of the event to jump (possible value is **_FRM**)

index Transaction index of the event to jump

Remarks:

If no parameters were specified, the application jumps to the current event being processed by VSE.
If wrong parameters were specified (for example, index exceeding the maximal 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_level  = in.Level;
  interesting_index = in.Index;
}
…
OnFinishScript()
{
  …
  # go to the interesting event…
  GotoEvent(interesting_level, interesting_index);
}
```

17.2 SetMarker()

This function sets a marker for some trace event.

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

Parameters:

marker_text	Text of the marker
level	Transaction level of the event to jump (possible values **_FRM**)
index	Transaction index of the event to jump

Remarks:

If no parameters were specified, other than **marker_text,** the application sets the marker to the current event being processed by VSE.

If wrong parameters were specified (for example, index exceeding the maximal index for the 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_level = in.Level;
  interesting_index = in.Index;
}
…
OnFinishScript()
{
…
# set marker to the interesting event…
SetMarker("  !!! Cool Marker !!! ", interesting_level, interesting_index);

# go to the interesting event…
GotoEvent(interesting_level, interesting_index);
}
```

18 File Functions

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

18.1 OpenFile()

This function opens a file for writing.

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

Parameters:

file_path Full path to the file to open. (For **'\'** use **'\\'**)

append This parameter (if present and not equal to 0) specifies that VSE should append the following write operations to the contents of the file; otherwise, the contents of the file are cleared.

mode This parameter specifies the open mode (text or binary) for the file.
If omitted, text mode is used. It can be one of the following values:
_FO_BINARY (= 0): Opens file in binary mode
_FO_TEXT (= 1): Opens file in text mode (by default)

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

# Opens file in binary mode, the previous contents are erased.
file_handle = OpenFile("D:\\data.bin", 0, _FO_BINARY);
…
Write(file_handle, 0xAABBCCDD); # Writes integer value into the binary file.
CloseFile(file_handle);         # Closes the binary file.
```

18.2 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.
…
```

18.3 WriteString()

This function writes a text string to the file.

Format: `WriteString(file_handle, text_string)`

Parameters:

file_handle	File “handle”
text_string	Text string

Remarks:

If the **file_handle** parameter refers to a text file, then a trailing **End-Of-Line** symbol is added to the text. In case of binary files, the **End-Of-Line** symbol is not added.

Example:

```
…
set file_handle = 0;
…
file_handle = OpenFile(“D:\\Log.txt”); # Opens text 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 the file.
…
```

18.4 Write()

This function writes data to the file.

Format: `Write(file_handle, value, num_of_bytes)`

Parameters:

file_handle "handle" to the file previously opened by **OpenFile()**

value Data to write

num_of_bytes Optional parameter specifying how many bytes to take from the **value** parameter (if omitted, length is calculated automatically based on the value type)

Remarks:

This function is primarily for binary files. It can be used for text files only if the **value** parameter is a string of text. In that case, it is equivalent to the **WriteString()** function and the **num_of_bytes** parameter is ignored.

Example:

```
…
set BinFile = 0;
…
BinFile = OpenFile("C:\\data.bin", 0, _FO_BINARY);
…
Write(BinFile, "All we need is love!!!");     # Write a string to the binary file.
Write(BinFile, "All we need is love!!!", 3); # Write a substring ("All") to the binary file.

val = 0xBEEF;
Write(BinFile, val);     # Writes integer = (EF BE 00 00) to the binary file.
Write(BinFile, val, 2); # Writes WORD = (EF BE) to the binary file.

Write(BinFile, 'AABBCCDDEEFF12345678'); # Write a byte chain to the binary file.

Write(BinFile, [ 0xAA, "UWB", 12, 0xBEEF ]); # Write a list of values to the binary file.
…
CloseFile(BinFile); # Closes the binary file.
…
```

18.5 ShowInBrowser()

This function opens 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 extension **.htm** and the file handle passed to **ShowInBrowser()** function belongs to a file with such an extension, then this file is opened in the Internet Explorer.

Format: `ShowInBrowser (file_handle)`

Parameters:

file_handle	File “handle”

Example:

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

19 COM/Automation Communication Functions

This group of functions covers VSE capabilities to communicate with COM/Automation™ clients connected to the UWB*Tracer*™ application. (See the *UWBTracer Automation Manual* for details about how to connect to the UWB*Tracer* application and VSE)

19.1 NotifyClient()

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

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

Parameters:

event_id Integer parameter representing event identifier

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

Example:

```
…
if(SomeCondition())
{
        NotifyClient(2, [ in.Index, "RF CHANNEL", “Beacon Frame", in.SubType,
                        in. PayloadLength, TimeToText(in.Time)]);
}
…
# Here we sent 2 parameters to clients applications:
# 2 (integer),
# [ in.Index, "RF CHANNEL", “Beacon Frame", in.SubType,
#   in. PayloadLength, TimeToText(in.Time)] (list)
```

Remark:

See an example of handling this notification by client applications and parsing code in the *UWBTracer Automation™ Manual*.

20 User Input Functions

20.1 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 title is omited, 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 was 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 a GUI. For VSEs controlled by COM/Automation™ clients, it does nothing.

This function "locks" the UWB*Tracer*™ application, which means that there is no access to other application features until the dialog box is closed. To prevent too many MsgBox calls (in case a script is not written correctly), VSE keeps track of all function calls demanding user interaction and doesn't show dialog boxes if some customizable limit is 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…
 }
```

20.2 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 or string containing the contents of the text box. (See the *CSL Manual* for details about list objects.)

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 title is omitted, 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 default_text is omited, 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 is 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 a CSL list object or the text typed in the text box as it is.

If **return_type = _IB_LIST** (by default), the text in the text box is considered to be a set of list items divided by **','** (only hexadecimal, decimal and string items are currently supported).

For example, the text:

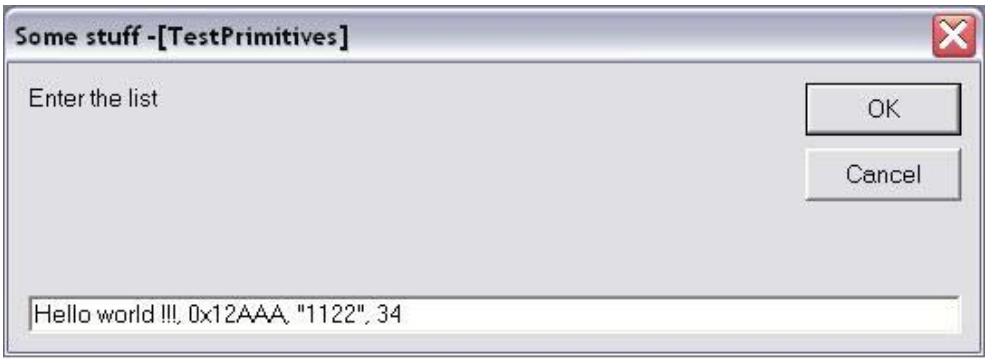
```
Hello world !!!, 12, Something, 0xAA, 10, "1221"
```

produces 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 to be a string.



Remark:

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

This function "locks" the UWB*Tracer* application, which means that there is no access to other application features until the dialog box is closed. To prevent too many InputBox calls (in case a script is not written correctly), VSE keeps track of all function calls demanding user interaction and doesn't show dialog boxes if some customizable limit is exceeded (returns a null object in that case).

Example:

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

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

20.3 GetUserDlgLimit()

This function returns the current limit of user dialogs 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
…
```

20.4 SetUserDlgLimit()

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

21 String Manipulation/Formating Functions

21.1 FormatEx()

Writes formatted data to a string. **Format** 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:

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

Remarks:

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"
```

22 Miscellaneous Functions

22.1 ScriptForDisplayOnly()

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

Format: `ScriptForDisplayOnly ()`

Example:

```
ScriptForDisplayOnly();
```

22.2 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 a sleeping time.

Format: `Sleep(time)`

Parameters:

time	VSE time object specifying sleep time

Example:

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

22.3 ConvertToHTML()

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

Format: `ConvertToHTML(text_string)`

Parameters:

text_string	Text string

Example:

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

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

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

22.4 Pause()

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

Format: `Pause()`

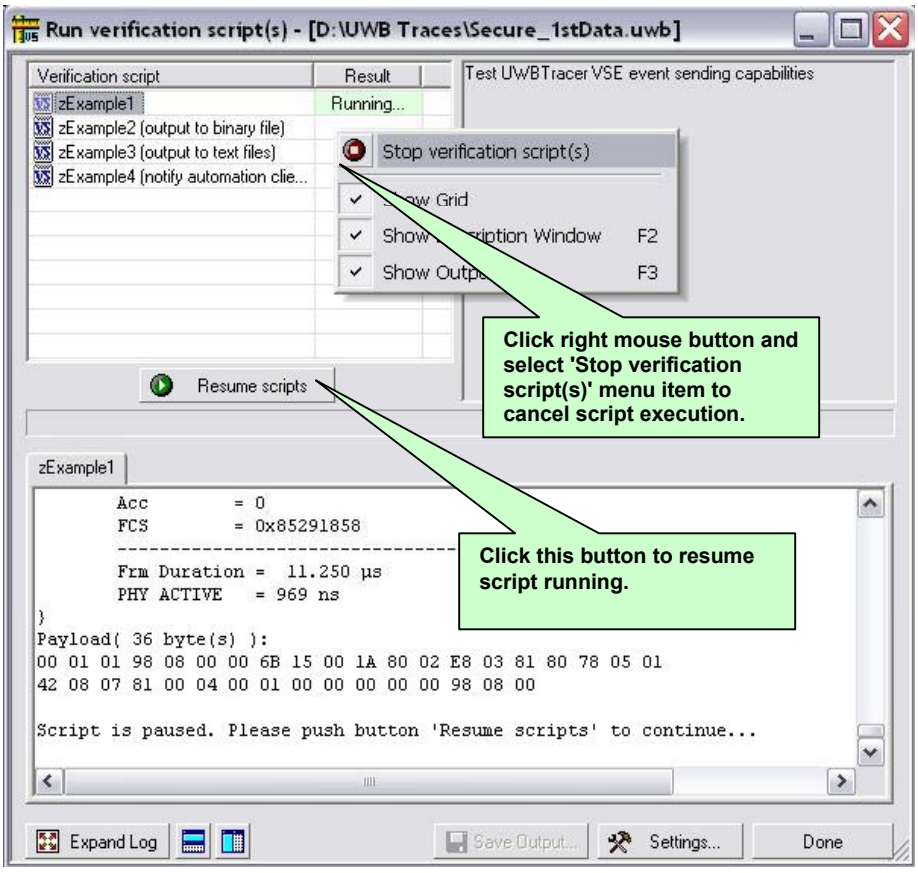
Example:

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

Remark:

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

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



23 The Important VSE Script Files

The VSE working files are located in the ..**Scripts\VFScripts** subfolder of the main UWB*Tracer*™ 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 LeCroy PSG (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 (put the definitions of constants, variables, and functions used in many scripts here)

How to Contact LeCroy

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 …	LeCroy Protocol Solutions Group Customer Support 3385 Scott Blvd. Santa Clara, CA 95054-3115
Send e-mail…	support@catc.com
Visit LeCroy's web site…	http://www.lecroy.com/