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

Automation API

for

LeCroy SAS/SATA Tracer/Trainer

Reference Manual

Manual Version 1.0
For SAS/SATA Tracer Software Version 2.60/4.60

January 13, 2006

Document Disclaimer

Trademarks and Servicemarks

Copyright

Version

This is version 1.0 the SASTracer/SATracer Automation API Reference Manual. This manual applies to SASTracer/SATracer software version 2.60/4.60 and higher.

Table of Contents

1 Introduction 1
1.1 System Requirements 1
1.2 Support Resources 1
1.3 Setting Up Automation for Local Use 1
1.4 Setting Up Automation for Remote Use 1
2 SASTracer/SATracer Object Model 2
3 SASAnalyzer Object 4
3.1 IAnalyzer Interface 5
3.1.1 IAnalyzer::GetVersion 6
3.1.2 IAnalyzer::OpenFile 7
3.1.3 IAnalyzer::StartGeneration 8
3.1.4 IAnalyzer::StopGeneration 9
3.1.5 IAnalyzer::StartRecording 10
3.1.6 IAnalyzer::StopRecording 12
3.1.7 IAnalyzer::MakeRecording 13
3.1.8 IAnalyzer::LoadDisplayOptions 14
3.1.9 IAnalyzer::GetRecordingOptions 15
3.2 ISASAnalyzer Interface 16
3.2.1 ISASAnalyzer::ResumeGeneration 17
4 SASTrace Object 18
4.1 ITrace Interface 19
4.1.1 ITrace::GetName 20
4.1.2 ITrace::ApplyDisplayOptions 21
4.1.3 ITrace::Save 22
4.1.4 ITrace::ExportToText 23
4.1.5 ITrace::Close 25
4.1.6 ITrace::ReportFileInfo 26
4.1.7 ITrace::ReportErrorSummary 27
4.1.8 ITrace::GetPacket 29
4.1.9 ITrace::GetPacketsCount 31
4.1.10 ITrace::GetTriggerPacketNum 32
4.1.11 ITrace::AnalyzerErrors 33
4.2 ISASTrace interface 34
4.2.1 ISASTrace::GetBusPacket 34
4.3 ISASVerificationScript Interface 35
4.3.1 ISASVerificationScript:: RunVerificationScript 36
4.3.2 ISASVerificationScript:: GetVScriptEngine 38

5 SASRecOptions Object 40
5.1 IRecOptions Interface 41
5.1.1 IRecOptions::Load 41
5.1.2 IRecOptions::Save 42
5.1.3 IRecOptions::SetRecMode 43
5.1.4 IRecOptions::SetBufferSize 44
5.1.5 IRecOptions::SetPostTriggerPercentage 45
5.1.6 IRecOptions::SetTriggerBeep 46
5.1.7 IRecOptions::SetSaveExternalSignals 47
5.1.8 IRecOptions::SetTraceFileName 48
5.1.9 IRecOptions::Reset 49
5.2 ISASRecOptions Interface 50
6 SASPacket Object 51
6.1 IPacket Interface 52
6.1.1 IPacket::GetTimestamp 52
6.2 ISASPacket Interface 53
6.2.1 ISASPacket::GetPacketData 53
6.2.2 ISASPacket::GetDirection 56
6.2.3 ISASPacket::GetErrors 57
7 SASTraceErrors Object 58
7.1 IAnalyzerErrors Dispinterface 58
7.1.1 IAnalyzerErrors::get_Item 59
7.1.2 IAnalyzerErrors::get_Count 60
8 SASVScriptEngine Object 62
8.1 IVScriptEngine Interface 63
8.1.1 IVScriptEngine::VScriptName 64
8.1.2 IVScriptEngine::Tag 65
8.1.3 IVScriptEngine::RunVScript 66
8.1.4 IVScriptEngine:: RunVScriptEx 67
8.1.5 IVScriptEngine:: LaunchVScript 69
8.1.6 IVScriptEngine:: Stop 70
8.1.7 IVScriptEngine:: GetScriptVar 71
8.1.8 IVScriptEngine:: SetScriptVar 73
9 SASVScriptEngine Object Events 75
9.1 _IVScriptEngineEvents Interface 75
9.1.1 _ IVScriptEngineEvents::OnVScriptReportUpdated 78
9.1.2 _ IVScriptEngineEvents:: OnVScriptFinished 79
9.1.3 _ IVScriptEngineEvents:: OnNotifyClient 80
10 SASAnalyzer Object Events 81
10.1 _IAnalyzerEvents Dispinterface 81
10.1.1 _IAnalyzerEvents::OnTraceCreated 82
10.1.2 _IAnalyzerEvents::OnStatusReport 83
11 CATCAnalyzerAdapter 86
11.1 IAnalyzerAdapter Interface 87
11.1.1 IAnalyzerAdapter::CreateObject 87
11.1.2 IAnalyzerAdapter:: Attach 89
11.1.3 IAnalyzerAdapter:: Detach 91
11.1.4 IAnalyzerAdapter::IsValidObject 93

1 Introduction

LeCroy's SASTracer/SATracer software provides a rich, functional COM/Automation API to the most important functionalities of the LeCroy SASTracer/SATracer Protocol Analyzer and LeCroy SASTrainer/SATrainer Exerciser. This makes it a great tool for implementation of automated programs for complicated testing, development and debugging. The "dual" nature of the interfaces provided makes it easy to use the SASTracer/SATracer COM API in different IDEs (Integrated Development Environments) supporting the COM architecture.

A special support for typeless script languages (such as VB and JavaScript), while overriding some restrictions imposed by script engines (remote access, dynamic object creation and handling events), gives the opportunity to write client applications very quickly and easily. One does not require significant programming skills nor is installation of expensive and powerful programming language systems required. All these features, along with the ability to set up all necessary DCOM permissions during the installation process, make the LeCroy SASTracer/SATracer an attractive tool in automating and speeding up many engineering processes.

1.1 System Requirements

The Automation API was introduced with the following release: SASTracer/SATracer software 2.60/4.60. This document covers the functionality available in SASTracer/SATracer 2.60/4.60.

1.2 Support Resources

As new functionalities are added to the API, not all of them are supported by older versions of the SASTracer/SATracer software. For newer releases of SASTracer/SATracer software, please refer to the LeCroy web site: www.lecroy.com

1.3 Setting Up Automation for Local Use

If you intend to run Automation on the SASTracer/SATracer's Host Controller (i.e., the PC attached to the SASTracer/SATracer), you do not need to perform any special configuration. You can simply execute the scripts or programs you have created and they will run the analyzer. In order to use the SASTracer/SATracer COM API, during the installation process the application should be registered as a COM server in a system registry.

1.4 Setting Up Automation for Remote Use

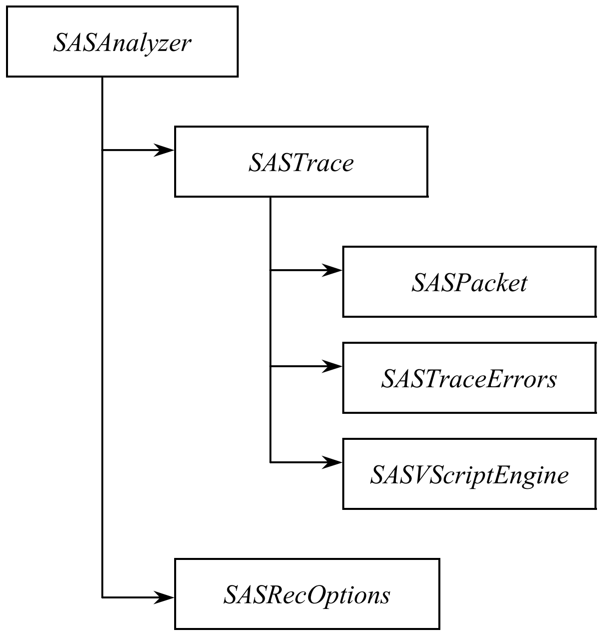
If you would like to access SASTracer/SATracer remotely over a network, you should install the SASTracer/SATracer application on both server and client systems and accept the enabling remote access option during the installation. You can also perform a manual DCOM configuration.

2 SASTracer/SATracer Object Model

LeCroy's SASTracer/SATracer API programmatically exposes its functionality through objects. You work with an object by using its properties and methods. Objects are named according to the portion of an application they represent, and they are ordered in a hierarchy.

A single object occupies the topmost tier of LeCroy SASTracer/SATracer object hierarchy: *SASAnalyzer*.

The following object model diagram shows how the objects in an object model fit together:



Only the *SASAnalyzer* object is creatable at the top level (for instance, via the *CoCreateInstance* call from a C/C++ client). Instantiation of an object of other classes requires API calls.

The Class ID and App ID for the *SASAnalyzer* object are the following.

Class ID (SASTracer):	`0B179BB7-DC61-11d4-9B71-000102566088`
Class ID (SATracer):	`486E3D65-C8F4-4f5f-A417-CD292C034DB3`
App ID (both SASTracer & SATracer):	CATC.SASTracer

All interfaces are dual interfaces that allow simple use from typeless languages (like VBScript), as well as from C/C++.

All objects implement *ISupportErrorInfo* interface for easy error handling from the client.

Objects	Interfaces	Description
SASAnalyzer	*Ianalyzer* *ISASAnalyzer* *_IAnalyzerEvents*	Represents the SASTracer application
SASTrace	*ITrace* *ISASTrace** *ISASVerificationScript**	Represents recorded trace
SASRecOptions	*IRecOptions* *ISASRecOptions*	Represents recording options
SASPacket	*IPacket* *ISASPacket**	Represents single packet of the recorded trace
SASTraceErrors	*IAnalyzerErrors**	Represents the collection of errors occurred in the recorded trace

* Primary interfaces

The examples of C++ code given in this document assume using the “import” technique of creating COM clients; that means the corresponding include is used:

```
#import "SASAutomation.tlb" no_namespace named_guids
```

Appropriate wrapper classes are created in .tli and .tlh files by the compiler.

Samples of WSH, VBScript and C++ client applications are provided.

3 SASAnalyzer Object

The *SASAnalyzer* object is a top-level object of SASTracer API. The *SASAnalyzer* object allows user to control the recording and traffic generation, open trace files and access the recording and generation options. The *SASAnalyzer* object supports the following interfaces:

Interfaces	Description
IAnalyzer	Facilitates recording and traffic generation, opens trace files, and retrieves recording options.
ISASAnalyzer	Extends the *IAnalyzer* interface: Adds advanced generator functionality, retrieves generation options.
_IAnalyzerEvents	Events from *SASAnalyzer* object.

The *ISASAnalyzer* interface is a primary interface for the *SASAnalyzer* object.

The Class ID and App ID for the *SASAnalyzer* object are the following.

Class ID (SASTracer):	`0B179BB7-DC61-11d4-9B71-000102566088`
Class ID (SATracer):	`486E3D65-C8F4-4f5f-A417-CD292C034DB3`
App ID (both SASTracer & SATracer):	CATC.SASTracer

Example

```
WSH:

       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )

C++:

       ISASAnalyzer*   poSASAnalyzer;

       // create SASAnalyzer object
       if ( FAILED( CoCreateInstance(
                CLSID_SASAnalyzer,
                NULL, CLSCTX_SERVER,
                IID_ISASAnalyzer,
                (LPVOID *)&poSASAnalyzer ) )
               return;
```

3.1 IAnalyzer Interface

The *IAnalyzer* interface is a dual interface for the *SASAnalyzer* object.

IAnalyzer implements the following methods:

- *GetVersion*
- *OpenFile*
- *StartGeneration*
- *StopGeneration*
- *StartRecording*
- *StopRecording*
- *MakeRecording*
- *LoadDisplayOptions*

Note: All methods of the *IAnalyzer* interface are also available in the *ISASAnalyzer* interface (see Section 3.2 *ISASAnalyzer Interface* on page 16).

3.1.1 IAnalyzer::GetVersion

```
HRESULT GetVersion (
       [in] EAnalyzerVersionType version_type,
       [out, retval] WORD* analyzer_version )
```

Retrieves the current version of a specified subsystem.

Parameters

`version_type`	– the subsystem being queried for version; *EAnalyzerVersionType* enumerator has the following values: `ANALYZERVERSION_SOFTWARE ( 0 )` – software
`analyzer_version`	– version of the subsystem queried

Return values

`ANALYZERCOMERROR_INVALIDVERSIONTYPE` – specified version type is invalid

Remarks

Example

```
WSH:
       Set Analyzer = WScript.CreateObject( “CATC.SASTracer” )
       SwVersion = Analyzer.GetVersion( 0 )
       MsgBox "Software " & SwVersion

C++:

       HRESULT      hr;
       ISASAnalyzer*   poSASAnalyzer;

       // create SASAnalyzer object
       if ( FAILED( CoCreateInstance(
                CLSID_SASAnalyzer,
                NULL, CLSCTX_SERVER,
                IID_ISASAnalyzer,
                (LPVOID *)&poSASAnalyzer ) )
              return;

       WORD sw_version;
       try
       {
              sw_version = poAnalyzer->GetVersion( ANALYZERVERSION_SOFTWARE );
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
              return 1;
       }

       TCHAR buffer[20];
       _stprintf( buffer, _T("Software version:%X.%X"), HIBYTE(sw_version), LOBYTE(sw_version)
);
```

3.1.2 IAnalyzer::OpenFile

```
HRESULT OpenFile (
       [in] BSTR file_name,
       [out, retval] IDispatch** trace )
```

Opens a trace file, and creates the *SASTrace* object.

Parameters

`file_name`	– string providing the full pathname to the trace file
`trace`	– address of a pointer to the *SASTrace* object interface

Return values

`ANALYZERCOMERROR_UNABLEOPENFILE` – unable to open file

Remarks

SASTrace object is created via this method call, if the call was successful.

Example

WSH:

```
CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set Trace = Analyzer.OpenFile( CurrentDir & "Input\errors.sas" )
```

C++:

```
HRESULT        hr;
ISASAnalyzer*   poSASAnalyzer;

// create SASAnalyzer object
if ( FAILED( CoCreateInstance(
        CLSID_SASAnalyzer,
        NULL, CLSCTX_SERVER,
        IID_ISASAnalyzer,
        (LPVOID *)&poSASAnalyzer ) )
       return;

// open trace file
IDispatch* trace;
try
{
       trace = poSASAnalyzer->OpenFile( m_szRecFileName ).Detach();
}
catch ( _com_error& er)
{
       if (er.Description().length() > 0)
              ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
       else
              ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
       return 1;
}

// query for VTBL interface
ISASTrace* SAS_trace;
hr = trace->QueryInterface( IID_ISASTrace, (LPVOID *)&SAS_trace );
trace->Release();

if( FAILED(hr) )
       return;
```

3.1.3 IAnalyzer::StartGeneration

```
HRESULT StartGeneration (
      [in] BSTR gen_file_name,
      [in] long reserved1,
      [in] long reserved2 )
```

Starts traffic generation from the file.

Parameters

`gen_file_name`	- string providing the full pathname to the generation file.
`reserved1`	- reserved for future use
`reserved2`	- reserved for future use

Return values

`ANALYZERCOMERROR_UNABLEOPENFILE`	- unable to open file
`ANALYZERCOMERROR_UNABLESTARTGENERATION`	- unable to start generation (invalid state, etc.)

Remarks

Example

WSH:

```
CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
ret = Analyzer.StartGeneration( CurrentDir & "Input\connect.ssg", 0, 0 )
```

C++:

```
HRESULT        hr;
ISASAnalyzer*   poSASAnalyzer;
TCHAR          m_szGenFileName   [_MAX_PATH];

// create SASAnalyzer object
if ( FAILED( CoCreateInstance(
        CLSID_SASAnalyzer,
        NULL, CLSCTX_SERVER,
        IID_ISASAnalyzer,
        (LPVOID *)&poSASAnalyzer ) )
       return;

. . .

try
{
       poAnalyzer->StartGeneration( m_szGenFileName, 0, 0 );
}
catch ( _com_error& er)
{
       if (er.Description().length() > 0)
              ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
       else
              ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
       return 1;
}
```

3.1.4 IAnalyzer::StopGeneration

```
HRESULT StopGeneration ( )
```

Stops any current generation in progress.

Return values

`ANALYZERCOMERROR_UNABLESTARTGENERATION` – unable to stop generation (invalid state, etc.)

Remarks

Example

```
C++:
        ISASAnalyzer* poAnalyzer;

        . . .

         try
         {
             poAnalyzer->StopGeneration();
         }
         catch ( _com_error& er)
         {
                if (er.Description().length() > 0)
                       ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
                else
                       ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
                return 1;
         }
```

3.1.5 IAnalyzer::StartRecording

```
HRESULT StartRecording (
       [in] BSTR ro_file_name )
```

Starts recording with the specified recording options.

Parameters

`ro_file_name`	– string providing the full pathname to the recording options file; if the parameter is omitted, then recording starts with default recording options

Return values

`ANALYZERCOMERROR_UNABLESTARTRECORDING`	– unable to start recording

Remarks

After recording starts, this function will return. The analyzer continues recording until it is finished or until the *StopRecording* method call is performed. During the recording, the events are sent to the event sink (see the *_IAnalyzerEvents Dispinterface* on page 81).

The recording options file is the file with extension *.rec* created by the SASTracer application. You can create this file when you select "*Setup -> Recording Options…*" from the SASTracer application menu, change the settings in the "*Recording Options*" dialog box and then select the "*Save…*" button.

Example

```
VBScript:

        <OBJECT
               RUNAT=Server
               ID = Analyzer
               CLASSID = "clsid: 297CD804-08F5-4A4F-B3BA-779B2654B27C "
        >
        </OBJECT>

        <INPUT TYPE=TEXT   VALUE="" NAME="TextRecOptions">

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnStartRecording_OnClick
               On Error Resume Next
               Analyzer.StartRecording TextRecOptions.value
               If Err.Number <> 0 Then
                      MsgBox Err.Number & ":" & Err.Description
               End If
        End Sub
        -->
        </SCRIPT>


C++:
        ISASAnalyzer* sas_analyzer;
        BSTR         ro_file_name;

        . . .

        try
        {
               sas_analyzer->StartRecording( ro_file_name )
        }
        catch ( _com_error& er)
```

```
{
        if (er.Description().length() > 0)
                ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
        else
                ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
        return 1;
}
```

3.1.6 IAnalyzer::StopRecording

```
HRESULT StopRecording (
       [in] BOOL abort_upload )
```

Stops recording started by the *IAnalyzer::StartRecording* (on page 10) method.

Parameters

`abort_upload`	– TRUE if the caller wants to abort the upload, no trace file will be created; FALSE if the caller wants to upload the recorded trace

Return values

`ANALYZERCOMERROR_UNABLESTOPRECORDING`	– error stopping recording

Remarks

Stops recording that was started by the *StartRecording* method. The event will be issued when recording is actually stopped (via the *_IAnalizerEvents* interface) if the parameter of this method call was FALSE.

Example (for SASTracer)

```
VBScript:

        <OBJECT
               RUNAT=Server
               ID = Analyzer
               CLASSID = "clsid: 0B179BB7-DC61-11d4-9B71-000102566088 "
        >
        </OBJECT>

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnStopRecording_OnClick
               On Error Resume Next
               Analyzer.StopRecording True
               If Err.Number <> 0 Then
                      MsgBox Err.Number & ":" & Err.Description
               End If
        End Sub
        -->
        </SCRIPT>


C++:
        ISASAnalyzer* sas_analyzer;

        . . .

        try
        {
               sas_analyzer->StopRecording( FALSE )
        }
        catch ( _com_error& er)
        {
               if (er.Description().length() > 0)
                      ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
               else
                      ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
               return 1;
        }
```

3.1.7 IAnalyzer::MakeRecording

```
HRESULT MakeRecording (
       [in] BSTR ro_file_name,
       [out, retval] IDispatch** trace )
```

Makes recording with the specified recording options file.

Parameters

`ro_file_name`	– string providing the full pathname to a recording options file; if the parameter is omitted, then recording starts with default recording options
`Trace`	– address of a pointer to the *SASTrace* object interface

Return values

`ANALYZERCOMERROR_UNABLESTARTRECORDING`	– unable to start recording

Remarks

This method acts like the *StartRecording* method but will not return until recording is completed. The *SASTrace* object is created via this method call if the call was successful.

The recording options file is the file with extension *.rec* created by the SASTracer application. You can create this file when you select "*Setup -> Recording Options…*" from the SASTracer application menu, change the settings in the "*Recording Options*" dialog box, and then select the "*Save…*" button.

Example

WSH:

```
CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set Trace = Analyzer.MakeRecording( CurrentDir & "Input\test_ro.rec" )
```

C++:

```
IDispatch* trace;
ISASAnalyzer* sas_analyzer;
BSTR       ro_file_name;
HRESULT    hr;

. . .

try
{
       trace = sas_analyzer->MakeRecording( ro_file_name ).Detach();
}
catch ( _com_error& er)
{
       if (er.Description().length() > 0)
              ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
       else
              ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
       return 1;
}

// query for VTBL interface
ISASTrace* sas_trace;
hr = trace->QueryInterface( IID_ISASTrace, (LPVOID *)&sas_trace );
trace->Release();
```

3.1.8 IAnalyzer::LoadDisplayOptions

```
HRESULT LoadDisplayOptions (
      [in] BSTR do_file_name )
```

Loads display options that will apply to a trace opened or recorded later.

Parameters

`do_file_name`	– string providing the full pathname to a display options file

Return values

`ANALYZERCOMERROR_UNABLELOADDO`	– unable to load the display options file

Remarks

Use this method if you want to filter traffic of some type. The display options loaded by this method call will apply only on trace file opened or recorded after this call.

Display options file is the file with extension *.opt* created by the SASTracer application. You can create this file when you select "*Setup -> Display Options…*" from the SASTracer application menu, change the settings in the "Display Options" dialog box, and then select the "*Save…*" button.

Example

See Section 4.1.2 *ITrace::ApplyDisplayOptions* on page 21.

3.1.9 IAnalyzer::GetRecordingOptions

```
HRESULT GetRecordingOptions (
       [out, retval] IDispatch** recording_options )
```

Retrieves the interface for access to the recording options.

Parameters

`recording_options` – address of a pointer to the *SASRecOptions* object interface

Return values

Remarks

SASRecOptions object is created via this method call, if the call was successful.

Example

```
WSH:
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       Set RecOptions = Analyzer.GetRecordingOptions( )

C++:

       HRESULT        hr;
       ISASAnalyzer*   poSASAnalyzer;

       // create SASAnalyzer object
       if ( FAILED( CoCreateInstance(
                CLSID_SASAnalyzer,
                NULL, CLSCTX_SERVER,
                IID_ISASAnalyzer,
                (LPVOID *)&poSASAnalyzer ) )
              return;

       // open trace file
       IDispatch* rec_opt;
       try
       {
              rec_opt = poSASAnalyzer->GetRecordingOptions().Detach();
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
              return 1;
       }

       // query for VTBL interface
       ISASRecOptions* ib_rec_opt;
       hr = rec_opt->QueryInterface( IID_ISASRecOptions, (LPVOID *)&ib_rec_opt );
       rec_opt->Release();

       if( FAILED(hr) )
              return;
```

3.2 ISASAnalyzer Interface

The *ISASAnalyzer* interface is a dual interface for the *SASAnalyzer* object.

This interface is derived from the *IAnalyzer* interface.

The *ISASAnalyzer* interface implements all methods from *IAnalyzer* interface plus the following:

GetGenerationOptions
ResumeGeneration

3.2.1 ISASAnalyzer::ResumeGeneration

```
HRESULT ResumeGeneration ( )
```

Resumes generation if it was previously paused.

Return values

Remarks

Example

```
C++:
        ISASAnalyzer* poAnalyzer;

        . . .

         try
         {
             poAnalyzer->ResumeGeneration();
         }
         catch ( _com_error& er)
         {
                if (er.Description().length() > 0)
                       ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
                else
                       ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
                return 1;
         }
```

4 SASTrace Object

SASTrace object represents the recorded trace file. The *SASTrace* object allows users to:

- get trace information
- access trace packets
- access trace errors
- save/export the trace or a portion of the trace

The *SASTrace* object can be created by:

- using *IAnalyzer::OpenFile* method (on page 7)
- using *IAnalyzer::MakeRecording* method (on page 13)
- handling *_IAnalyzerEvents::OnTraceCreated* event (on page 82)

The *SASTrace* object supports the following interfaces:

Interfaces	Description
Itrace	Implements trace packets and trace errors access, different report types, export, and saving.
ISASTrace	Extends *ITrace* interface: Adds the functionality for accessing the *SASTracePacket* object.
ISASVerificationScript	Exposes the functionality for running verification scripts

The *ISASTrace* interface is a primary interface for the *SASTrace* object.

4.1 ITrace Interface

The *ITrace* interface is a dual interface for the *SASTrace* object.

It implements the following methods:

GetName
ApplyDisplayOptions
Save
ExportToText
Close
ReportFileInfo
ReportErrorSummary
GetPacket
GetPacketsCount
GetTriggerPacketNum
AnalyzerErrors

Note: All methods of *ITrace* interface are also available in Section 4.2 ISASTrace interface (on page 34).

4.1.1 ITrace::GetName

```
HRESULT GetName (
       [out, retval] BSTR* trace_name )
```

Retrieves the trace name.

Parameters

`trace_name`	– the name of the trace

Return values

Remarks

This name can be used for presentation purposes.
Do not forget to free the string returned by this method call.

Example

```
WSH:
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Trace = Analyzer.MakeRecording( CurrentDir & "Input\test_ro.rec" )
       MsgBox "Trace name " & Trace.GetName

C++:
       ISASTrace* sas_trace;

       . . .

       _bstr_t bstr_trace_name;
       try
       {
              bstr_trace_name = sas_trace->GetName();
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
              return 1;
       }

        TCHAR str_trace_name[256];
        _tcscpy( str_trace_name, (TCHAR*)( bstr_trace_name) );
        SysFreeString( bstr_trace_name );

       ::MessageBox( NULL, str_trace_name, _T("Trace name"), MB_OK );
```

4.1.2 ITrace::ApplyDisplayOptions

```
HRESULT ApplyDisplayOptions (
       [in] BSTR do_file_name )
```

Applies the specified display options to the trace.

Parameters

`do_file_name`	– string providing the full pathname to the display options file

Return values

`ANALYZERCOMERROR_UNABLELOADDO`	- unable to load the display options file

Remarks

Use this method if you want to filter traffic of some type in the recorded or opened trace.
The display options file is the file with extension *.opt* created by the SASTracer application. You can create this file when you select "*Setup -> Display Options…*" from the SASTracer application menu, change the settings in the "Display Options" dialog box, and then select the "*Save…*" button.

Note: This does not work on Multisegment traces

Example

```
WSH:
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Trace = Analyzer.MakeRecording( CurrentDir & "Input\test_ro.rec" )
       Trace.ApplyDisplayOptions    CurrentDir & "Input\test_do.opt"
       Trace.Save                   CurrentDir & "Output\saved_file.sas"

C++:

       ISASTrace* sas_trace;
       TCHAR file_name[_MAX_PATH];

       . . .

       try
       {
              sas_trace->ApplyDisplayOptions( file_name );
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
              return 1;
       }
```

4.1.3 ITrace::Save

```
HRESULT Save (
       [in] BSTR file_name,
       [in, defaultvalue(-1)] long packet_from,
       [in, defaultvalue(-1)] long packet_to )
```

Saves trace into a file while allowing you to specify a range of packets.

Parameters

`file_name`	– string providing the full pathname to file where the trace is saved
`packet_from`	– *beginning packet number* when you are saving a range of packets; value –1 means that the first packet of the saved trace is the first packet of this trace
`packet_to`	– *ending packet number* when you are saving a range of packets; value –1 means that the last packet of the saved trace is the last packet of this trace

Return values

`ANALYZERCOMERROR_UNABLESAVE`	- unable to save the trace file
`ANALYZERCOMERROR_INVALIDPACKETNUMBER`	- bad packet range

Remarks

Use this method if you want to save a recorded or an opened trace into a file. If the display options are applied to this trace (see *ITrace::ApplyDisplayOptions* on page 21 or *IAnalyzer::LoadDisplayOptions* on page 14), then hidden packets would not be saved.

If the packet range specified is invalid (for example, *packet_to* is more than the last packet number in the trace, or *packet_from* is less than the first packet number in the trace, or *packet_from* is more than *packet_to*), then the packet range will be adjusted automatically.

Example

```
WSH:
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Trace = Analyzer.MakeRecording (CurrentDir & "Input\test_ro.rec")
       Trace.ApplyDisplayOptions    CurrentDir & "Input\test_do.opt"
       Trace.Save                   CurrentDir & "Output\saved_file.sas"

C++:

       ISASTrace* sas_trace;
       TCHAR file_name[_MAX_PATH];
       LONG packet_from;
       LONG packet_to;
       . . .

       try
       {
              sas_trace->Save( file_name, packet_from, packet_to );
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
              return 1;
       }
```

4.1.4 ITrace::ExportToText

```
HRESULT ExportToText (
      [in] BSTR file_name,
      [in, defaultvalue(-1)] long packet_from,
      [in, defaultvalue(-1)] long packet_to );
```

Exports the trace into a text file while allowing you to specify a range of packets.

Parameters

`file_name`	- string providing the full file pathname for the exported trace
`packet_from`	- *beginning packet number* when you are exporting a range of packets; value –1 means that the first packet of the exported trace is the first packet of this trace
`packet_to`	- *ending packet number* when you are exporting a range of packets, value –1 means that the last packet of the exported trace is the last packet of this trace

Return values

`ANALYZERCOMERROR_UNABLESAVE` – unable to export trace file

Remarks

Use this method if you want to export a recorded or an opened trace into a text file. If the display options applied to this trace (see *ITrace::ApplyDisplayOptions* on page 21 or *IAnalyzer::LoadDisplayOptions* on page 14), then hidden packets would not be exported.

If the packet range is specified and it is invalid (for example, *packet_to* is more than the last packet number in the trace, or *packet_from* is less than the first packet number in the trace, or *packet_from* is more than *packet_to*), then packet range will be adjusted automatically.

Here is a snippet of an exported text file:

```
  File c:\analyzersw\traces\sas\allsata.sas.
  From Frame #1 to Frame #20.

Frame#
_______|_______________________________________________________________________T2
Frame(1) 1.5(G) SATA RCV Time Stamp(29.196 501 432)
_______|_______________________________________________________________________T2
Frame(2) 1.5(G) SATA XMT SATA_SOF FIS Type(DMA Activate) Port(0x0)
_______| Data(4 bytes) CRC(0x8FA86FC5) SATA_EOF Time Stamp(29.196 513 752)
_______|_______________________________________________________________________I2
Frame(3) 1.5(G) SATA RCV Time Stamp(29.196 514 177)
_______|_______________________________________________________________________I2
Frame(4) 1.5(G) SATA XMT SATA_SOF FIS Type(Data) Port(0x0)
_______| Data(8196 bytes) CRC(0x7BFAA709) SATA_EOF Time Stamp(29.196 518 682)
_______|_______________________________________________________________________T2
Frame(5) 1.5(G) SATA RCV Time Stamp(29.196 518 952)
_______|_______________________________________________________________________T2
Frame(6) 1.5(G) SATA XMT SATA_SOF FIS Type(DMA Activate) Port(0x0)
_______| Data(4 bytes) CRC(0x8FA86FC5) SATA_EOF Time Stamp(29.196 632 872)
_______|_______________________________________________________________________I2
Frame(7) 1.5(G) SATA RCV Time Stamp(29.196 633 167)
_______|_______________________________________________________________________I2
Frame(8) 1.5(G) SATA XMT SATA_SOF FIS Type(Data) Port(0x0)
_______| Data(8196 bytes) CRC(0x7919EFB6) SATA_EOF Time Stamp(29.196 634 687)
_______|_______________________________________________________________________T2
Frame(9) 1.5(G) SATA RCV Time Stamp(29.196 634 950)
```

```
_______|_______________________________________________________________________T2
Frame(10) 1.5(G) SATA XMT SATA_SOF FIS Type(DMA Activate) Port(0x0)
_______| Data(4 bytes) CRC(0x8FA86FC5) SATA_EOF Time Stamp(29.196 748 927)
_______|_______________________________________________________________________I2
Frame(11) 1.5(G) SATA RCV Time Stamp(29.196 749 220)
_______|_______________________________________________________________________I2
Frame(12) 1.5(G) SATA XMT SATA_SOF FIS Type(Data) Port(0x0)
_______| Data(8196 bytes) CRC(0x38CA16DA) SATA_EOF Time Stamp(29.196 750 740)
_______|_______________________________________________________________________T2
Frame(14) 1.5(G) SATA XMT SATA_SOF FIS Type(DMA Activate) Port(0x0)
_______| Data(4 bytes) CRC(0x8FA86FC5) SATA_EOF Time Stamp(29.196 864 980)
_______|_______________________________________________________________________I2
Frame(15) 1.5(G) SATA RCV Time Stamp(29.196 865 272)
_______|_______________________________________________________________________I2
```

Example

```
WSH:
        Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
        CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
        Set Trace = Analyzer.MakeRecording (CurrentDir & "Input\test_ro.rec")
        Trace.ApplyDisplayOptions    CurrentDir & "Input\test_do.opt"
        Trace.ExportToText           CurrentDir & "Output\text_export.txt"

C++:

        ISASTrace* sas_trace;
        TCHAR file_name[_MAX_PATH];
        LONG packet_from;
        LONG packet_to;
        . . .
        try
        {
                sas_trace->ExportToText( file_name, packet_from, packet_to );
        }
        catch ( _com_error& er)
        {
                if (er.Description().length() > 0)
                        ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
                else
                        ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
                return 1;
        }
```

4.1.5 ITrace::Close

```
HRESULT Close ( )
```

Closes the trace.

Parameters

Return values

Remarks

Closes the current trace, but does not release the interface pointer. Call *IUnknown::Release* method right after this method call. No *ITrace* method call will succeed after calling *ITrace::Close* method.

(Currently, there is no need to call *ITrace::Close* directly since *IUnknown::Release* will close the trace.)

Example

4.1.6 ITrace::ReportFileInfo

```
HRESULT ReportFileInfo (
       [in] BSTR file_name )
```

Saves trace information into a specified HTML file.

Parameters

`file_name`	– string providing the full pathname to a file where the trace information report is stored

Return values

`ANALYZERCOMERROR_UNABLESAVE`	– unable to create the trace information report

Remarks

Creates a new trace information file if the file specified in the *file_name* parameter does not exist.

Example

```
WSH:
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Trace = Analyzer.MakeRecording (CurrentDir & "Input\test_ro.rec")
       Trace.ReportFileInfo         CurrentDir & "Output\file_info.html"

C++:
       ISASTrace* sas_trace;
       TCHAR file_name[_MAX_PATH];

       . . .

       try
       {
              sas_trace->ReportFileInfo( file_name );
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
              return 1;
       }
```

4.1.7 ITrace::ReportErrorSummary

```
HRESULT ReportErrorSummary (
      [in] BSTR file_name )
```

Saves trace error summary information into the specified text file.

Parameters

`file_name` – string providing the full pathname to a file where the error summary report is stored.

Return values

`ANALYZERCOMERROR_UNABLESAVE` – unable to create trace information report

Remarks

This method doesn't work on Multisegment traces.

Creates a new error summary file if the file specified in the *file_name* parameter does not exist. Stores error summary in the specified file.

Here is an example of data stored using this method call:

```
   Error report for allsata.sas recording file.

_______|_______________________________________________________________________
Packet Error: Idle Error on channel I1 (0):
_______|_______________________________________________________________________
Packet Error: Idle Error on channel T1 (0):
_______|_______________________________________________________________________
Packet Error: Bad CRC on channel I1 (0):
_______|_______________________________________________________________________
Packet Error: Bad CRC on channel T1 (0):
_______|_______________________________________________________________________
Packet Error: Disparity Error on channel I1 (0):
_______|_______________________________________________________________________
Packet Error: Disparity Error on channel T1 (0):
_______|_______________________________________________________________________
Packet Error: Bad Code on channel I1 (0):
_______|_______________________________________________________________________
Packet Error: Bad Code on channel T1 (0):
_______|_______________________________________________________________________
Packet Error: Alignment Error on channel I1 (0):
_______|_______________________________________________________________________
Packet Error: Alignment Error on channel T1 (0):
_______|_______________________________________________________________________
Packet Error: Delimiter Error on channel I1 (0):
_______|_______________________________________________________________________
Packet Error: Delimiter Error on channel T1 (0):
_______|_______________________________________________________________________
Packet Error: Invalid SSP Frame Type on channel I1 (0):
_______|_______________________________________________________________________
Packet Error: Invalid SSP Frame Type on channel T1 (0):
_______|_______________________________________________________________________
Packet Error: Invalid SMP Frame Type on channel I1 (0):
_______|_______________________________________________________________________
Packet Error: Invalid SMP Frame Type on channel T1 (0):
_______|_______________________________________________________________________
Transaction Error: Timed Out SSP on channel I1 (0):
```

```
_______|_______________________________________________________________________
Transaction Error: Timed Out SSP on channel T1 (0):
_______|_______________________________________________________________________
SCSI Error: Incomplete Command on channel I1 (0):
_______|_______________________________________________________________________
SCSI Error: Incomplete Command on channel T1 (0):
_______|_______________________________________________________________________
MGMT Error: Incomplete Command on channel I1 (0):
_______|_______________________________________________________________________
MGMT Error: Incomplete Command on channel T1 (0):
_______|_______________________________________________________________________
```

Example

```
WSH:
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Trace = Analyzer.MakeRecording (CurrentDir & "Input\test_ro.rec")
       Trace.ReportErrorSummary     CurrentDir & "Output\error_summary.txt"

C++:
       ISASTrace* sas_trace;
       TCHAR file_name[_MAX_PATH];

       . . .

       try
       {
              sas_trace->ReportErrorSummary( file_name );
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(),  _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(), _T("SASTracer client"), MB_OK );
              return 1;
       }
```

4.1.8 ITrace::GetPacket

```
HRESULT GetPacket (
      [in] long packet_number,
      [in, out] VARIANT* packet,
      [out, retval] long* number_of_bytes )
```

Retrieves a raw packet representation in the *PACKETFORMAT_BYTES* format (see *IPacket Interface* for details, on page 52).

Parameters

`packet_number`	- zero based number of packet to retrieve
`packet`	- raw packet representation
`number_of_bytes`	- number of bytes in the raw packet representation

Return values

`ANALYZERCOMERROR_INVALIDPACKETNUMBER` - specified packet number is invalid

Remarks

packet parameter has *VT_ARRAY | VT_VARIANT* actual automation type. Each element of this array has the *VT_UI1* automation type.

Example

```
VBScript:
        <OBJECT
                ID = Analyzer
                CLASSID = "clsid: 297CD804-08F5-4A4F-B3BA-779B2654B27C "
        >
        </OBJECT>
        <INPUT TYPE=TEXT NAME="TextPacketNumber">
        <P ALIGN=LEFT ID=StatusText></P>

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Function DecToBin(Param, NeedLen)
                While Param > 0
                        Param = Param/2
                        If Param - Int(Param) > 0 Then
                                Res = CStr(1) + Res
                        Else
                                Res = CStr(0) + Res
                        End If
                        Param = Int(Param)
                Wend
                DecToBin = Replace( Space(NeedLen - Len(Res)), " ", "0") & Res
        End Function

        Sub BtnGetPacket_OnClick
                On Error Resume Next
                Dim Packet
                NumberOfBytes = CurrentTrace.GetPacket (TextPacketNumber.value, Packet)
                If Err.Number <> 0 Then
                        MsgBox "GetPacket:" & Err.Number & ":" & Err.Description
                Else
                        For Each PacketByte In Packet
                                PacketStr = PacketStr & DecToBin(PacketByte, 8) & " "
                                NBytes = NBytes + 1
                        Next
                        PacketStr = Left( PacketStr, NumberOfBytes)
                        StatusText.innerText = "Packet ( " & NumberOfBytes & " bytes ): " &
                        PacketStr
                End If
        End Sub
        -->
        </SCRIPT>
```

C++:

```
        ISASTrace* sas_trace;
        LONG packet_number;

        . . .

        VARIANT packet;
        VariantInit( &packet );
        long number_of_bytes;
        try
        {
            number_of_bytes = sas_trace->GetPacket( packet_number, &packet );
        }
        catch ( _com_error& er)
        {
            if (er.Description().length() > 0)
               ::MessageBox( NULL, er.Description(), _T("SASTracer client"), MB_OK );
            else
               ::MessageBox( NULL, er.ErrorMessage(),_T("SASTracer client"), MB_OK );
            return 1;
        }

        if ( packet.vt == ( VT_ARRAY | VT_VARIANT) )
        {
            SAFEARRAY* packet_safearray = packet.parray;

            TCHAR packet_message[256];
            TCHAR elem[64];
             _stprintf( packet_message, _T("packet #%ld: "), packet_number );

            for ( long i=0; i<(long)packet_safearray->rgsabound[0].cElements; i++)
            {
                VARIANT var;
                HRESULT hr = SafeArrayGetElement(packet_safearray, &i, &var);
                if (FAILED(hr))
                {
                    ::MessageBox( NULL, _T("Error accessing array"), _T("SASTracer client"),
MB_OK );
                    return 1;
                }
                if ( var.vt != ( VT_UI1) )
                {
                    ::MessageBox( NULL, _T("Array of bytes expected"), _T("SASTracer client"),
MB_OK );
                    return 1;
                }

                _stprintf( elem, _T("%02X "), V_UI1(&var) );
                _tcscat( packet_message, elem );
            }
            _stprintf( elem, _T("%d bytes"), number_of_bytes );
            _tcscat( packet_message, elem );

            ::MessageBox( NULL, packet_message, _T("Raw packet bits"), MB_OK );
        }
        else
        {
            ::MessageBox( NULL, _T("Invalid argument"), _T("SASTracer client"), MB_OK );
        }
```

4.1.9 ITrace::GetPacketsCount

```
HRESULT GetPacketsCount (
       [out, retval] long* number_of_packets )
```

Retrieves the total number of packets in the trace.

Parameters

`number_of_packets`	– total number of packets in the trace

Return values

Remarks

Example

```
WSH:
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Trace = Analyzer.MakeRecording( CurrentDir & "Input\test_ro.rec" )
       MsgBox Trace.GetPacketsCount & " packets recorded"

C++:
       ISASTrace* sas_trace;

       . . .

       long number_of_packets;
       long trigg_packet_num;
       try
       {
              bstr_trace_name = sas_trace->GetName();
              number_of_packets = sas_trace->GetPacketsCount();
              trigg_packet_num = sas_trace->GetTriggerPacketNum();
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(), _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(),_T("SASTracer client"), MB_OK );
              return 1;
        }

        TCHAR str_trace_name[256];
        _tcscpy( str_trace_name, (TCHAR*)( bstr_trace_name) );
        SysFreeString( bstr_trace_name );

       TCHAR trace_info[256];
       _stprintf( trace_info, _T("Trace:'%s', total packets:%ld, trigger packet:%ld"),
              str_trace_name, number_of_packets, trigg_packet_num );

       ::SetWindowText( m_hwndStatus, trace_info );
```

4.1.10 ITrace::GetTriggerPacketNum

```
HRESULT GetTriggerPacketNum  (
      [out, retval] long* packet_number )
```

Retrieves the trigger packet number.

Parameters

`packet_number`	- zero based number of the packet where the trigger occured

Return values

Remarks

Example

```
WSH:
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       Set Trace = Analyzer.MakeRecording( CurrentDir & "Input\test_ro.rec" )
       TriggerPacket = Trace.GetTriggerPacketNum
       Trace.Save CurrentDir & "Output\trigger_portion.sas", CInt(ErrorPacket)-5,
CInt(ErrorPacket)+5
```

C++:

See an example for *ITrace::GetPacketsCount* on page 31.

4.1.11 ITrace::AnalyzerErrors

```
HRESULT AnalyzerErrors (
       [in] long error_type,
       [out, retval] IAnalyzerErrors** analyzer_errors )
```

Retrieves trace file errors. Returns an interface pointer to the *SASTraceErrors* object.

Parameters

`error_type` – type of error collection you want to retrieve; the following values are valid:

```
0x00000001 - Idle Error
0x00000002 - Bad CRC
0x00000004 - Disparity Error
0x00000008 - Bad Code
0x00000010 - Alignment Error
0x00000020 - Delimiter Error
0x00000400 - Invalid SSP Frame Type
0x00000800 - Invalid SMP Frame Type
```

`analyzer_errors` – address of a pointer to the *SASTraceErrors* object interface

Return values

`ANALYZERCOMERROR_INVALIDERROR` – invalid error type specified

Remarks

SASTraceErrors object is created via this method call if the call was successful.

Example

```
WSH:
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       Set Trace = Analyzer.MakeRecording( CurrentDir & "Input\test_ro.rec" )
       Set Errors = Trace.AnalyzerErrors( 8 ) ' Packet Length Error

C++:
       ISASTrace* sas_trace;

       . . .

       IAnalyzerErrors* trace_errors;
       try
       {
              trace_errors = sas_trace->AnalyzerErrors(error_type).Detach();
       }
       catch ( _com_error& er)
       {
              if (er.Description().length() > 0)
                     ::MessageBox( NULL, er.Description(), _T("SASTracer client"), MB_OK );
              else
                     ::MessageBox( NULL, er.ErrorMessage(),_T("SASTracer client"), MB_OK );
              return 1;
       }

       . . .

       analyser_errors->Release();
```

4.2 ISASTrace interface

The *ISASTrace* interface is a primary dual interface for the *SASTrace* object.

This interface is derived from the *ITrace* interface.

The *ISASTrace* interface implements all methods from the *ITrace* interface plus the following:
GetBusPacket

4.2.1 ISASTrace::GetBusPacket

```
HRESULT GetBusPacket   (
       [in] long packet_number,
       [out, retval] IDispatch** packet )
```

Retrieves the interface for a packet within a trace.

Parameters

`packet_number`	- zero based number of packet to retrieve
`packet`	- address of a pointer to the *SASPacket* object interface

Return values

Remarks

SASPacket object is created via this method call if the call was successful.

Example

```
WSH:

C++:
        ISASTrace* sas_trace;

        . . .

        IDispatch* packet;
        try
        {
           packet = sas_trace->GetBusPacket( GetDlgItemInt(IDC_PACKET_NUMBER) ).Detach();
        }
        catch ( _com_error& er)
        {
                if (er.Description().length() > 0)
                        ::MessageBox( NULL, er.Description(), _T("SASTracer client"), MB_OK );
                else
                        ::MessageBox( NULL, er.ErrorMessage(),_T("SASTracer client"), MB_OK );
                return 1;
        }

        ISASPacket* custom_packet;
        HRESULT hr = packet->QueryInterface( IID_ISASPacket, (void**)&custom_packet );
        packet->Release();
```

4.3 ISASVerificationScript Interface

The *ISASVerificationScript* interface is an interface for the *SASTrace* object. It exposes the trace functionality for running verification scripts. This interface is not dual—which means that scripting languages cannot use it directly, though all of its methods described below are exposed to script languages through the primary automation interface of the *SASTrace* object.

Remarks

Verification scripts are scripts written in a special manner using the *CATC Script Language (CSL)*. These scripts can be “run” over a recorded trace to “verify” the trace for some verification conditions or to extract more advanced information from the trace. Such scripts utilize a special feature of the SASTracer application, its *Verification Script Engine*.

Please refer to the *SASTracer Manual*, the *SASTracer Verification Script Engine Manual* and the *SASTracer File Based Decoding Manual* for more details.

Attention:

The functions of this interface may be legally called either for regular traces or multi-segmented traces. The VSE will open segments of the multi-segmented trace during script execution when it is needed.

4.3.1 ISASVerificationScript:: RunVerificationScript

```
HRESULT RunVerificationScript (
       [in] BSTR verification_script,
       [out, retval] VS_RESULT *result )
```

Runs a verification script over the recorded trace

Parameters

`verification_script` - the name of the verification script to run

`result` - address of a variable where to keep the result of verification; *VS_RESULT* is an enumeration type that can have 5 possible meanings:

`SCRIPT_RUNNING`	`(-2)`	– verification script is running
`SCRIPT_NOT_FOUND`	`(-1)`	– verification script with the specified name was not found
`FAILED`	`( 0)`	– verification failed
`PASSED`	`( 1)`	– verification passed
`DONE`	`( 2)`	– verification is done, don't care about result

Return values

S_OK – if the verification script executed successfully.

Remarks

The name of the verification script is the name of the verification script file (**.pevs*). If only the name of the script, without file extension, is specified, SASTracer's server is going to search for the named script among the scripts loaded from the *\Scripts\VFScripts* folder under SASTracer's installation folder. If the full path to the script is specified, then the server is going to attempt loading the script from the specified path prior to running it. Example: For a verification script file named "test.pevs" the test name would be "test". Please refer to the *SASTracer Verification Script Engine Manual* for more details.

Example

```
C++:
       // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* trace;
        . . .

        ISASVerificationScript* vscript = NULL;

        if ( SUCCEEDED ( trace->QueryInterface( IID_ISASVerificationScript, (void**)&vscript ) )
        {
                try
                {

                    VS_RESULT result = vscript ->RunVerificationScript("Test1");
                    if( result == PASSED )
                    {
                        ::MessageBox( NULL, "Test verification 1 is passed !!!", "SASTracer
                        client", MB_OK );
                }
            }
            catch ( _com_error& er)
            {
                if (er.Description().length() > 0)
                    ::MessageBox( NULL, er.Description(),  "SASTracer client", MB_OK );
                else
                    ::MessageBox( NULL, er.ErrorMessage(), "SASTracer client", MB_OK );
                return 1;
            }

        }
        else
        {
            ::MessageBox( NULL, "Unable to get ISASVerificationScript interface !!!",
          _T("SASTracer client"), MB_OK );
          return 1 ;
        }

        . . .
```

WSH:

```
        Set Analyzer = WScript.CreateObject("CATC.SASTracer")
        Set Trace    = Analyzer.OpenFile( "C:\Some trace files\some_trace.sas" )

        Dim Result
        Result = Trace.RunVerificationScript( "Test1" )


        If Result = 1 Then
            Msgbox "PASSED"
        Else
            Msgbox "FAILED"
        End If

        MsgBox( "Done" )
```

4.3.2 ISASVerificationScript:: GetVScriptEngine

```
HRESULT GetVScriptEngine(
       [in] BSTR script_name,
       [out, retval] IVScriptEngine** vs_engine )
```

Retrieves the verification script engine object

Parameters

`script_name`	– the name of the verification script to initialize the verification script engine
`vs_engine`	– address of a pointer to the *SASVScriptEngine* object interface

Return values

S_OK – if the verification script engine object was successfully retrieved.

Remarks

The name of the verification script is the name of the verification script file (*.*pevs*). See remark to *ISASVerificationScript:: RunVerificationScript* for details on page 36.

Example

```
C++:
       // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* sas_trace;

        . . .

        ISASVerificationScript* sas_vscript = NULL;

        sas_trace->QueryInterface( IID_ISASVerificationScript, (void**)&sas_vscript ) )
        assert( sas_vscript != NULL );

        IVScriptEngine* sas_vsengine = NULL;
        sas_vsengine = sas_vscript -> GetVScriptEngine("Test_1");
        assert( sas_vsengine != NULL );

        VS_RESULT result = sas_vsengine ->RunVScript();
        if( result == PASSED )
        {
               ::MessageBox( NULL, "Test verification 1 is passed !!!", "SASTracer client", MB_OK
               );
        }

        . . .

WSH:
        Set Analyzer = WScript.CreateObject("CATC.SASTracer")
        Set Trace    = Analyzer.OpenFile( "C:\Some trace files\some_trace.sas" )

        Dim Result

        Set VSEngine = Trace.GetVScriptEngine( "Test1" )
        Result = VSEngine.RunVScript

        If Result = 1 Then
               Msgbox "PASSED"
        Else
               Msgbox "FAILED"
        End If

        MsgBox( "Done" )
```

5 SASRecOptions Object

The *SASRecOptions* object represents the options for the SASTracer hardware and is used to specify the recording parameters.

The *SASRecOptions* object allows user to:

- load/save the recording options from/to the file
- set up recording mode and recording buffer size
- set up custom recording parameters such as ChannelSettings, DataTruncate, MultiSegment mode, SpoolMode, etc.

The *SASRecOptions* object can be created by using the IAnalyzer::GetRecordingOptions method (on page 15).

The *SASRecOptions* object supports the following interfaces:

Interfaces	Description
IRecOptions	Allows you to load/save recording options from/to the file, reset recording options, set up recording mode, recording buffer size, trigger position, and the trace file name
ISASRecOptions	Identical to *IRecOptions* interface

The *ISASRecOptions* interface is a primary interface for *SASRecOptions* object.

5.1 IRecOptions Interface

The *IRecOptions* interface is a dual interface for *SASRecOptions* object.

IRecOptions implements the following methods:

- *Load*
- *Save*
- *SetRecMode*
- *SetBufferSize*
- *SetPostTriggerPercentage*
- *SetTriggerBeep*
- *SetSaveExternalSignals*
- *SetTraceFileName*
- *Reset*

Note: All methods of the *IRecOptions* interface are also available in the *ISASRecOptions Interface* (on page 50).

5.1.1 IRecOptions::Load

```
HRESULT Load (
      [in] BSTR ro_file_name )
```

Loads recording options from the specified file.

Parameters

`ro_file_name`	– string that provides the full pathname to the recording options file

Return values

`ANALYZERCOMERROR_UNABLEOPENFILE` – unable to open file

Remarks

Example

WSH:

```
CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set RecOptions = Analyzer.GetRecordingOptions( )
RecOptions.Load( CurrentDir & "Input\rec_options.rec" )
```

C++:

5.1.2 IRecOptions::Save

```
HRESULT Save (
       [in] BSTR ro_file_name )
```

Saves recording options into the specified file.

Parameters

`ro_file_name`	- string that provides the full pathname to the recording options file

Return values

`ANALYZERCOMERROR_UNABLEOPENFILE`	- unable to open file

Remarks

If the specified file does not exist, it will be created; if it exists, it will be overwritten.

Example

WSH:

```
CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set RecOptions = Analyzer.GetRecordingOptions( )
' do the changes of recording options here
RecOptions.Save( CurrentDir & "Input\rec_options.rec" )
```

C++:

5.1.3 IRecOptions::SetRecMode

```
HRESULT SetRecMode (
       [in] ERecModes rec_mode )
```

Sets the recording mode.

Parameters

`rec_mode`	– enumerated value providing the mode to set; *ERecModes* enumerator has the following values:

`RMODE_SNAPSHOT`	`( 0 )`	– snapshot recording mode
`RMODE_MANUAL`	`( 1 )`	– manual trigger
`RMODE_USE_TRG`	`( 2 )`	– event trigger

Return values

`E_INVALIDARG` – invalid recording mode was specified

Remarks

The default setting of recording options is a “snapshot” recording mode.

Example

WSH:

```
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set RecOptions = Analyzer.GetRecordingOptions( )
RecOptions.SetRecMode 2       ' Event trigger
```

C++:

5.1.4 IRecOptions::SetBufferSize

```
HRESULT SetBufferSize (
       [in] long buffer_size )
```

Sets the size of buffer to record.

Parameters

`buffer_size`	– size of the recording buffer in bytes

Return values

`E_INVALIDARG` – invalid buffer size was specified

Remarks

The default setting is 16MB for Conventional Recording mode and 120GB or 12.5 hours for Spooled Recording Mode.

Example

WSH:

```
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       Set RecOptions = Analyzer.GetRecordingOptions( )
       RecOptions.SetBufferSize 2*1024*1024  ' 2Mb
```

C++:

5.1.5 IRecOptions::SetPostTriggerPercentage

```
HRESULT SetPostTriggerPercentage (
      [in] short posttrigger_percentage )
```

Sets the post trigger buffer size.

Parameters

`posttrigger_percentage` – the size of the post trigger buffer in percent of the whole recording buffer (see *IRecOptions::SetBufferSize* on page 44).

Return values

`E_INVALIDARG` – invalid percentage was specified

Remarks

This method call has no effect if recording mode was set to `RMODE_SNAPSHOT` (see *IRecOptions::SetRecMode* on page 43). The default setting is 50%.

Example

WSH:

```
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set RecOptions = Analyzer.GetRecordingOptions( )
RecOptions.SetPostTriggerPercentage 60       ' 60%
```

C++:

5.1.6 IRecOptions::SetTriggerBeep

```
HRESULT SetTriggerBeep (
       [in] BOOL beep )
```

Sets a flag to make a sound when a trigger occurs.

Parameters

`beep`	- TRUE – beep when a trigger occurs, FALSE – do not beep when a trigger occurs

Return values

Remarks

The default state of the beeper is FALSE.

Example

WSH:

```
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set RecOptions = Analyzer.GetRecordingOptions( )
RecOptions.SetTriggerBeep TRUE
```

C++:

5.1.7 IRecOptions::SetSaveExternalSignals

```
HRESULT SetSaveExternalSignals (
      [in] BOOL save )
```

Sets a flag to save external signals.

Parameters

`save`	- TRUE – save external signals, FALSE – do not save external signals

Return values

Remarks

By default, external signals are not saved.

Example

WSH:

```
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set RecOptions = Analyzer.GetRecordingOptions( )
RecOptions.SetSaveExternalSignals TRUE
```

C++:

5.1.8 IRecOptions::SetTraceFileName

```
HRESULT SetTraceFileName (
       [in] BSTR file_name )
```

Sets the file path to where the trace will be stored after recording.

Parameters

`file_name`	– string that provides the full file pathname to where the recording will be stored

Return values

Remarks

If the specified file does not exist, it will be created; if it exists, it will be overwritten.

Example

```
WSH:

       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       Set RecOptions = Analyzer.GetRecordingOptions( )
       ' do the changes of recording options here
       RecOptions.Save( CurrentDir & "Input\trace.sas" )

C++:
```

5.1.9 IRecOptions::Reset

```
HRESULT Reset ( )
```

Resets the recording options to the initial state.

Parameters

Return values

Remarks

For default values of recording options, see the remarks sections of all *IRecOptions and ISASRecOptions* methods.

Example

```
WSH:

       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       Set RecOptions = Analyzer.GetRecordingOptions
       RecOptions.SetRecMode 2              ' Event trigger
       RecOptions.SetBufferSize 1024*1024  ' 1Mb
       RecOptions.SetPostTriggerPercentage 60  ' 60%
       . . .
       RecOptions.Reset


C++:
```

5.2 ISASRecOptions Interface

This interface is identical to the *IRecOptions Interface* (on page 41).

6 SASPacket Object

The *SASPacket* object represents a single packet of the recorded trace file.

The *SASPacket* object allows user to retrieve packet content and packet properties such as timestamp, link width, packet start lane, packet direction, and packet errors.

The *SASPacket* object can be created by calling *ISASTrace::GetBusPacket* (on page 34).

The *SASPacket* object supports the following interfaces:

Interfaces	Description
IPacket	Allows retrieval of the packet's timestamp
ISASPacket	Extends the *IPacket* interface

The *ISASPacket* interface is a primary interface for the *SASPacket* object.

6.1 IPacket Interface

The *IPacket* interface is a dual interface for *SASPacket* object.

IPacket implements the following method:
GetTimestamp

Note: All methods of the *IPacket* interface are also available in the *ISASPacket Interface* (on page 53).

6.1.1 IPacket::GetTimestamp

```
HRESULT GetTimestamp (
       [out, retval] double* timestamp )
```

Returns the packet timestamp in nanoseconds.

Parameters

`timestamp`	– timestamp of the beginning symbol of the packet from the start of recording

Return values

Remarks

Example

WSH:

```
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set Trace = Analyzer.MakeRecording( CurrentDir & "Input\test_ro.rec" )
TriggerPacket = Trace. GetTriggerPacketNum
Set Packet = Trace.GetBusPacket(TriggerPacket)
MsgBox "Trigger packet at " & Packet.GetTimestamp & " ns"
```

C++:

6.2 ISASPacket Interface

The *ISASPacket* interface is a primary dual interface for the *SASPacket* object.

This interface is derived from the *IPacket* interface.

The *ISASPacket* interface implements all methods from the *IPacket* interface plus the following:

GetPacketData
GetLinkNumber
GetFrameType
GetDirection
GetErrors
GetTotalDwords

6.2.1 ISASPacket::GetPacketData

```
HRESULT GetPacketData (
       [in] EPacketFormat format,
       [out] VARIANT* packet,
       [out, retval] long* number_of_bytes )
```

Retrieves a raw packet representation.

Parameters

`format`	- data representation format; the *EPacketFormat* enumerator has the following values:
	`PACKETFORMAT_BYTES ( 0 ) - bytes`
	`PACKETFORMAT_SCRAMBLED_BYTES( 1 ) - scrambled bytes`
	`PACKETFORMAT_TEN_BIT ( 2 ) - 10bit codes`

`packet`	- raw packet data
`number_of_bytes`	- number of bytes in the packet

Return values

```
ANALYZERCOMERROR_WRONGCALL -  Unknown packet format specified
```

Remarks

packet parameter has *VT_ARRAY | VT_VARIANT* actual automation type. For *PACKETFORMAT_BYTES* and *PACKETFORMAT_SCRAMBLED_BYTES*, each element of this array has the *VT_UI1* automation type. For *PACKETFORMAT_TEN_BIT*, each element of this array has the *VT_UI2* automation type.

Example

```
VBScript:
        <OBJECT
                ID = Analyzer
                CLASSID = "clsid: 0B179BB7-DC61-11d4-9B71-000102566088"
        >
        </OBJECT>
        <INPUT TYPE=TEXT NAME="TextPacketNumber">
```

```
<P ALIGN=LEFT ID=StatusText></P>

<SCRIPT LANGUAGE="VBScript">
<!--
Function DecToBin(Param, NeedLen)
       While Param > 0
              Param = Param/2
              If Param - Int(Param) > 0 Then
                     Res = CStr(1) + Res
              Else
                     Res = CStr(0) + Res
              End If
              Param = Int(Param)
       Wend
       DecToBin = Replace( Space(NeedLen - Len(Res)), " ", "0") & Res
End Function

Sub BtnGetPacket_OnClick
  ClearStatus()
  On Error Resume Next
  Set Packet = CurrentTrace.GetBusPacket (TextPacketNumber.value)

  If Err.Number <> 0 Then
     MsgBox "GetBusPacket:" & Err.Number & ":" & Err.Description
  Else
    Timestamp = Packet.GetTimestamp()
    If Err.Number <> 0 Then
       MsgBox "GetTimestamp:" & Err.Number & ":" & Err.Description
    End If

    NumberOfUnits = Packet.GetPacketData ( PACKETFORMAT_BYTES, PacketData)

    If Err.Number <> 0 Then
       MsgBox "GetPacketData:" & Err.Number & ":" & Err.Description
    Else

      For Each PacketByte In PacketData
        PacketStr = PacketStr & DecToBin(PacketByte, 8) & " "
        NBytes = NBytes + 1
      Next

      StatusText.innerText = "Packet ( " & NumberOfUnits  & " bytes ): " & PacketStr
    End If
  End If
End Sub
-->
</SCRIPT>
```

C++:

```
ISASPacket* custom_packet;
LONG packet_number;

. . .

VARIANT packet_data;
double timestamp_ns;
VariantInit( &packet_data );
long number_of_bytes;
try
{
    number_of_bytes = custom_packet->GetPacketData( PACKETFORMAT_BYTES, &packet_data );
    timestamp_ns    = custom_packet->GetTimestamp ( );
}
catch ( _com_error& er)
{
    if (er.Description().length() > 0)
       ::MessageBox( NULL, er.Description(), _T("SASTracer client"), MB_OK );
    else
       ::MessageBox( NULL, er.ErrorMessage(),_T("SASTracer client"), MB_OK );
    return 1;
}
```

```
        if ( packet_data.vt == ( VT_ARRAY | VT_VARIANT) )
        {
            SAFEARRAY* packet_safearray = packet_data.parray;

            TCHAR* packet_message = new TCHAR [ 3*packet_safearray->rgsabound[0].cElements + 64
];
            TCHAR elem[64];
            _stprintf( packet_message, _T("packet #%ld: "), GetDlgItemInt(IDC_PACKET_NUMBER) );

            _stprintf( elem, _T(" %.0lf ns"), timestamp_ns );
            _tcscat( packet_message, elem );

            _stprintf( elem, _T(", %d bytes:  "), number_of_bytes );
            _tcscat( packet_message, elem );

            for ( long i=0; i<(long)packet_safearray->rgsabound[0].cElements; i++)
            {
                VARIANT var;
                HRESULT hr = SafeArrayGetElement(packet_safearray, &i, &var);
                if (FAILED(hr))
                {
                    ::MessageBox( NULL, _T("Error accessing array"), _T("SASTracer client"),
MB_OK );
                    return 1;
                }
                if ( var.vt != ( VT_UI1) )
                {
                    ::MessageBox( NULL, _T("Array of bytes expected"), _T("SASTracer client"),
MB_OK );
                    return 1;
                }

                _stprintf( elem, _T("%02X "), V_UI1(&var) );
                _tcscat( packet_message, elem );

            }

            ::MessageBox( NULL, packet_message, _T("packet"), MB_OK );

            delete [] packet_message;
        }
        else
        {
            ::MessageBox( NULL, _T("Invalid argument"), _T("SASTracer client"), MB_OK );
        }
```

6.2.2 ISASPacket::GetDirection

```
HRESULT GetDirection (
       [out, retval] long* direction )
```

Returns direction (host/device for SATA or initiator/target for SAS) of this packet.

Parameters

`direction`	- 0 – host (initiator) packet 1 – device (target) packet

Return values

Remarks

Example

WSH:

```
CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
Set Trace = Analyzer.OpenFile( CurrentDir & "Input\errors.sas" )
Set Packet = Trace.GetBusPacket( 0 )
MsgBox "Direction: " & Packet.GetDirection
```

C++:

6.2.3 ISASPacket::GetErrors

```
HRESULT GetErrors (
       [out] VARIANT* error_array,
       [out, retval] long* number_of_errors )
```

Returns an array of errors present in this packet.

Parameters

`error_array`	- array of error id present in this packet. See *ITrace::AnalyzerErrors* on page 33 for error id values
`number_of_errors`	- total number of errors in this packet

Return values

Remarks

Example

WSH:

C++:

7 SASTraceErrors Object

The *SASTraceErrors* object represents the collection of errors that occurred in the recorded trace file.

The *SASTraceErrors* object can be created by calling *ITrace::AnalyzerErrors* (on page 33).

The *IAnalyzerErrors* interface is a primary interface for the *SASTraceErrors* object.

7.1 IAnalyzerErrors Dispinterface

This is a standard collection interface for collection of packet numbers with errors of a specified type (see *ITrace::AnalyzerErrors* on page 33).

It has the following methods, which are standard for the collection interfaces:

get_Item
get_Count

7.1.1 IAnalyzerErrors::get_Item

```
HRESULT get_Item(
       [in] long index,
       [out, retval] long* packet_number )
```

Returns a zero based packet number from error collection

Parameters

`index`	- index of the error in the collection
`packet_number`	- error packet number

7.1.2 IAnalyzerErrors::get_Count

```
HRESULT get_Count(
       [out, retval] long* number_of_errors )
```

Returns the number of errors in the trace.

Parameters

`number_of_errors`	– number of elements in the collection

Remarks

Example

```
WSH:
       ' makes recording, saves the portions of the recorded trace
       ' where "Running Disparity" errors occured
       CurrentDir = Left( WScript.ScriptFullName, InstrRev( WScript.ScriptFullName, "\" ) )
       Set Analyzer = WScript.CreateObject( "CATC.SASTracer" )
       Set Trace = Analyzer.MakeRecording( CurrentDir & "Input\test_ro.rec" )
       Set Errors = Trace.AnalyzerErrors( 32 ) ' Running Disparity Error
       For Each ErrorPacketNumber In Errors
              ErrorFile = CurrentDir & "\Output\PckLen_error_span_" &
                     CStr(ErrorPacketNumber) & ".sas"
              Trace.Save ErrorFile, CInt(ErrorPacketNumber)-5, CInt(ErrorPacketNumber)+5
       Next
```

```
C++:
        ISASTrace* sas_trace;

        . . .

        IAnalyzerErrors* analyser_errors;
        try
        {
                analyser_errors = sas_trace->AnalyzerErrors(error_type).Detach();
        }
        catch ( _com_error& er)
        {
                if (er.Description().length() > 0)
                        ::MessageBox( NULL, er.Description(), _T("SASTracer client"), MB_OK );
                else
                        ::MessageBox( NULL, er.ErrorMessage(),_T("SASTracer client"), MB_OK );
                return 1;
        }

        TCHAR all_errors[2048];
        _stprintf( all_errors, _T("Errors: ") );
        try
        {
                long errors_count = analyser_errors->GetCount();
                long analyzer_error;
                if ( !errors_count )
                {
                        _tcscat( all_errors, _T("none") );
                }
                for ( long i=0; i<errors_count && i<2048/32; i++ )
                {
                        analyzer_error = analyser_errors->GetItem(i);
                        TCHAR cur_error[32];
                        _stprintf( cur_error, _T(" %ld"), analyzer_error );
                        _tcscat( all_errors, cur_error );
                }
                if ( i>2048/32 )
                        _tcscat( all_errors, _T(" ...") );
        }
        catch ( _com_error& er)
        {
                if (er.Description().length() > 0)
                        ::MessageBox( NULL, er.Description(), _T("SASTracer client"), MB_OK );
                else
                        ::MessageBox( NULL, er.ErrorMessage(),_T("SASTracer client"), MB_OK );
                return 1;
        }


        analyser_errors->Release();

        ::SetWindowText( m_hwndStatus, all_errors );
```

8 SASVScriptEngine Object

The *SASVScriptEngine* object allows a user to run verification scripts over the recorded trace. It extends the functionality of the *ISASVerificationScript* interface of a *SASTrace* object. The main advantage of a *SASVScriptEngine* object is that it allows clients implementing *_IVScriptEngineEvents* a callback interface to receive notifications when a verification script is running.

The *SASVScriptEngine* object can be created by calling *ISASVerificationScript:: GetVScriptEngine* (on page 38).

The *SASVScriptEngine* object supports the following interfaces:

Interfaces	Description
IVScriptEngine	Provides advanced control over the verification script and allows you to execute the script asynchronously
_IAnalyzerEvents	Events from *SASVScriptEngine* object

The *IVScriptEngine* interface is a primary interface for *SASVScriptEngine* object.

Remarks

Verification scripts are scripts written in a special manner using the *CATC Script Language (CSL)*. These scripts can be "run" over a recorded trace to "verify" the trace for some verification conditions or to extract more advanced information from the trace. Such scripts utilize a special feature of the SASTracer application, its *Verification Script Engine*.

Please refer to the *SASTracer Manual*, the *SASTracer Verification Script Engine Manual*, and the *SASTracer File Based Decoding Manual* for more details.

8.1 IVScriptEngine Interface

The *IVScriptEngine* interface is the primary dual interface for the *SASVScriptEngine* object.

It implements the following properties and methods:

VscriptName
Tag
RunVScript
RunVScriptEx
LaunchVScript
Stop
GetScriptVar
SetScriptVar

8.1.1 IVScriptEngine::VScriptName

```
[propget] HRESULT VScriptName( [out, retval] BSTR *pVal )

[propput] HRESULT VScriptName( [in] BSTR newVal )
```

Property putting and getting current verification script name.

Parameters

`pVal`	– address of the variable where the current verification script name is kept
`newVal`	– the name of the verification script to initialize script verification engine

Return values

Remarks

The name of verification script is the name of verification script file (**.pevs*). If only the name of the script without file extension is specified, the SASTracer server is going to search for the named script among the scripts loaded from the \Scripts\VFScripts folder under SASTracer installation folder. If the full path to the script is specified, then the server is going to attempt loading the script from the specified path prior to running it.

Example

```
C++:
       // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* sas_trace;

        . . .

        ISASVerificationScript* sas_vscript = NULL;

        sas_trace->QueryInterface( IID_ISASVerificationScript, (void**)&sas_vscript ) )
        assert( sas_vscript != NULL );

        IVScriptEngine* sas_vsengine = NULL;
        sas_vsengine = sas_vscript -> GetVScriptEngine("MyVSEngine");
        assert( sas_vsengine != NULL );


        sas_vsengine -> PutVScriptName("Test_1");
        assert( sas_vsengine -> GetVScriptName() == "Test_1" );


        VS_RESULT result = sas_vsengine ->RunVScript();
        if( result == PASSED )
        {
            ::MessageBox( NULL, "Test 1 passed !!!", "SASTracer client", MB_OK );
         }

        . . .
```

8.1.2 IVScriptEngine::Tag

```
[propget] HRESULT Tag( [out, retval] int* pVal )

[propput] HRESULT Tag( [in] int newVal )
```

Property assigning and getting a tag to the VSE object. This tag will be used in event notifications allowing a client event handler to determine which VSE object sent the event.

Parameters

`pVal`	– address of the variable where the current VSE tag is kept
`newVal`	– the new tag for VSE

Return values

Remarks

Example

C++:

```
       // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* sas_trace;

        . . .

        ISASVerificationScript* sas_vscript = NULL;

        sas_trace->QueryInterface( IID_ISASVerificationScript, (void**)&sas_vscript ) )
        assert( sas_vscript != NULL );

        IVScriptEngine* sas_vsengine = NULL;
        sas_vsengine = sas_vscript -> GetVScriptEngine("Test_1");
        assert( sas_vsengine != NULL );

        sas_vsengine ->PutTag( 0xDDAADDAA );
        assert( sas_vsengine -> GetTag() == 0xDDAADDAA );


        VS_RESULT result = sas_vsengine ->RunVScript();
        if( result == PASSED )
        {
            ::MessageBox( NULL, "Test 1 passed !!!", "SASTracer client", MB_OK );
        }

        . . .
```

8.1.3 IVScriptEngine::RunVScript

```
HRESULT RunVScript( [out, retval] int* pResult )
```

Runs the verification script currently specified for this engine.

Parameters

`pResult`	– address of a variable where the results of the verification is kept.

Return values

Remarks

This method makes a "synchronous" call – which means that this method doesn't return until the script stops running. See *ISASVerificationScript:: RunVerificationScript* on page 36, for details.

Example

See C++ example to *IVScriptEngine::VScriptName* on page 64.

8.1.4 IVScriptEngine:: RunVScriptEx

```
HRESULT RunVScriptEx(
      [in] BSTR script_name,
      [out, retval] int* pResult )
```

Changes the current verification script name and runs verification script .

Parameters

`script_name`	– the name of the verification script to initialize the script verification engine
`pResult`	– address of a variable where the results of a verification is kept

Return values

Remarks

This method makes a “synchronous” call – which means that this method doesn’t return until the script stops running.

The name of verification script is the name of verification script file (**.pevs*). If only the name of the script without file extension is specified, the SASTracer server is going to search for the named script among the scripts loaded from the \Scripts\VFScripts folder under SASTracer installation folder. If the full path to the script is specified, then the server is going to attempt loading the script from the specified path prior to running it. See *ISASVerificationScript:: RunVerificationScript* method, on page 36, for details.

Example

```
C++:
        // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* sas_trace;

        . . .

        ISASVerificationScript* sas_vscript = NULL;

        sas_trace->QueryInterface( IID_ISASVerificationScript, (void**)&sas_vscript ) )
        assert( sas_vscript != NULL );

        IVScriptEngine* sas_vsengine = NULL;
        sas_vsengine = sas_vscript -> GetVScriptEngine("Test_1");
        assert( sas_vsengine != NULL );

        VS_RESULT result = sas_vsengine ->RunVScript();
        if( result == PASSED )
        {
                ::MessageBox( NULL, "Test 1 passed !!!", "SASTracer client", MB_OK );
        }


        result = sas_vsengine ->RunVScriptEx("Test_2");
        if( result == PASSED )
        {
                ::MessageBox( NULL, "Test 2 passed !!!", "SASTracer client", MB_OK );
        }


        result = sas_vsengine ->RunVScriptEx("C:\\MyTests\\Test_3.pevs");
        if( result == PASSED )
        {
```

```
        ::MessageBox( NULL, "Test 3 passed !!!", "SASTracer client", MB_OK );
}

. . .
```

8.1.5 IVScriptEngine:: LaunchVScript

```
HRESULT LaunchVScript()
```

Launches verification script.

Return values

S_FALSE – if VS Engine was not successfully launched (either it is already running or verification script was not found)

Remarks

This method makes an "asynchronous" call – which means that this method immediately returns after the script starts running.

When the verification script stops running, the VSE object will send a special event notification _ *IVScriptEngineEvents:: OnVScriptFinished* (on page 79) to the client event handler. You can also terminate the running script using the method *IVScriptEngine:: Stop* (on page 70).

Example

```
C++:
       // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* sas_trace;

        . . .

        ISASVerificationScript* sas_vscript = NULL;

        sas_trace->QueryInterface( IID_ISASVerificationScript, (void**)&sas_vscript ) )
        assert( sas_vscript != NULL );

        IVScriptEngine* sas_vsengine = NULL;
        sas_vsengine = sas_vscript -> GetVScriptEngine("Test_1");
        assert( sas_vsengine != NULL );

        VS_RESULT result = sas_vsengine ->LaunchVScript();

        // You can go further without waiting the result from the VSE object.
        // If you interested in the result you should implement the client event handler for
        // OnVScriptFinished() notification.
        . . .
```

8.1.6 IVScriptEngine:: Stop

```
HRESULT Stop()
```

Stops verification script previously launched by the *IVScriptEngine:: LaunchVScript* (on page 69).

Parameters

Return values

Remarks

Example

```
C++:
       // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* sas_trace;

        . . .

        ISASVerificationScript* sas_vscript = NULL;

        sas_trace->QueryInterface( IID_ISASVerificationScript, (void**)&sas_vscript ) )
        assert( sas_vscript != NULL );

        IVScriptEngine* sas_vsengine = NULL;
        sas_vsengine = sas_vscript -> GetVScriptEngine("Test_1");
        assert( sas_vsengine != NULL );

        VS_RESULT result = sas_vsengine ->LaunchVScript();
        . . .

        if( NotEnoughResourcesToProcessVS )
               sas_vsengine ->Stop();
        . . .
```

8.1.7 IVScriptEngine:: GetScriptVar

```
        HRESULT GetScriptVar (
              [in] BSTR var_name,
              [out, retval] VARIANT* var_value )
```

Returns the value of some verification script global variables before/after executing the script (refer to the *SASTracer Verification Script Engine Manual* and the *File Based Decoding Manual* for information on how a script can declare and set global variables). The resulting value may contain an integer, a string, or an array of VARIANTs (if a requested script variable is a list object—see the *SASTracer File Based Decoding Manual* for more details about list objects)

Parameters

`var_name`	– string providing the name of the global variable or constant used in the verification script running
`var_value`	– address of a VARIANT variable where the result will be kept

Return values

`E_PENDING` – if this method is called when the script is already running

Remarks

If there is no such global variable or constant with the name *var_name*, the resulting value will contain an empty VARIANT.

Example

```
C++:
       // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* sas_trace;

        . . .

        ISASVerificationScript* sas_vscript = NULL;

        sas_trace->QueryInterface( IID_ISASVerificationScript, (void**)&sas_vscript ) )
        assert( sas_vscript != NULL );

        IVScriptEngine* sas_vsengine = NULL;
        sas_vsengine = sas_vscript -> GetVScriptEngine("Test_1");
        assert( sas_vsengine != NULL );

        VS_RESULT result = sas_vsengine ->RunVScript();
        . . .

        VARIANT my_var;
        VariantInit( &my_var );

        sas_vsengine->GetScriptVar( _bstr_t("MyVar"), &my_var );

        if( my_var.vt == VT_BSTR ) ProcessString( my_var.bstrVal );
        . . .


        WSH:
                . . .
                Set Trace    = Analyzer.OpenFile( TraceName )    ' Open the trace
               Set VSEngine = Trace.GetVScriptEngine( VScript ) ' Get VS Engine object

               Result = VSEngine.RunVScript

               MyIntVar = VSEngine.GetScriptVar( "MyIntVar" ) ' Let's suppose that MyIntVar
contains an integer
               MyStrVar = VSEngine.GetScriptVar( "MyStrVar" ) ' Let's suppose that MyStrVar
contains a string

               MsgBox " MyIntVar = " & CStr(MyIntVar) & ", MyStrVar = " & MyStrVar
```

8.1.8 IVScriptEngine:: SetScriptVar

```
HRESULT SetScriptVar ( [in] BSTR var_name, [in] VARIANT var_value )
```

This method allows you to set the value of some verification script global variable before/after executing the script (refer to the *SASTracer Verification Script Engine Manual* and the *File Based Decoding Manual* for information on how a script can declare, set, and use global variables). Only integers, strings, or arrays of VARIANTs are allowed as correct values. Arrays of VARIANTs will be converted into list values inside of scripts—see the *SASTracer File Based Decoding Manual* for more details about list objects.

Parameters

`var_name`	– string providing the name of the global variable used in the verification script being run
`var_value`	– VARIANT value containing the new variable value

Return values

`E_PENDING` – if this method is called when the script is already running

Remarks

This function may be very useful because it allows you to set internal script variables before running a script, giving you the opportunity to make run-time customization from COM/Automation client applications.
In order for this operation to take effect during execution of the script, a global variable with the name specified by *var_name* should be declared by the script.

Example

```
C++:
       // In this example we use wrapper functions provided by #import directive
       //
        ISASTrace* sas_trace;

        . . .

        ISASVerificationScript* sas_vscript = NULL;

        sas_trace->QueryInterface( IID_ISASVerificationScript, (void**)&sas_vscript ) )
        assert( sas_vscript != NULL );

        IVScriptEngine* sas_vsengine = NULL;
        sas_vsengine = sas_vscript -> GetVScriptEngine("Test_1");
        assert( sas_vsengine != NULL );

        VARIANT my_var;
        VariantInit( &my_var );

        my_var.vt = VT_I4; // Integer
        my_var.lVal = 100;

        // set internal script variable 'MyVar' to 100
        sas_vsengine->SetScriptVar( _bstr_t("MyVar"), my_var );

        VS_RESULT result = sas_vsengine ->RunVScript();
        . . .
```

```
WSH:
        . . .

        Set Trace    = Analyzer.OpenFile( TraceName )    ' Open the trace
        Set VSEngine = Trace.GetVScriptEngine( VScript ) ' Get VS Engine object

        VSEngine.GetScriptVar( "MyIntVar" , 100 )
        VSEngine.GetScriptVar( "MyStrVar" , "Hello !!!" )
        Result = VSEngine.RunVScript
```

9 SASVScriptEngine Object Events

9.1 _IVScriptEngineEvents Interface

In order to retrieve the event notifications from SASTracer application when a verification script engine object is running the script, you must implement the *_IVScriptEngineEvents* callback interface. Since this interface is a default source interface for the *SASVScriptEngine* object, there is a very simple implementation from such languages like Visual Basic, VBA, VBScript, WSH, etc.

Some script engines impose restrictions on handling events from “indirect” automation objects in typeless script languages (when an automation interface to the object is obtained from a call of some method rather than from creation function—like *CreateObject()* in VBScript). The SASTracer provides a special COM class allowing the receiving and handling of notifications from a VSE object even in script languages not supporting event handling from "indirect" objects. Please refer to *CATCAnalyzerAdapter*, on page 86, for details.

Example

`C++:`

C++ implementation used in the examples below implements an event sink object by deriving it from *IdispEventImpl*, but not specifying the type library as a template argument. Instead, the type library and default source interface for the object are determined using *AtlGetObjectSourceInterface()*. A *SINK_ENTRY()* macro is used for each event from each source interface that is to be handled:

```
class CVSEngineSink : public IDispEventImpl<IDC_SRCOBJ_VSE, CVSEngineSink >
{
public:

...


BEGIN_SINK_MAP(CVSEngineSink)
        //Make sure the Event Handlers have __stdcall calling convention
        SINK_ENTRY( IDC_SRCOBJ_VSE, 1, OnVScriptReportUpdated )
        SINK_ENTRY( IDC_SRCOBJ_VSE, 2, OnVScriptFinished      )
        SINK_ENTRY( IDC_SRCOBJ_VSE, 3, OnNotifyClient         )
END_SINK_MAP()


HRESULT __stdcall OnVScriptReportUpdated ( BSTR newLine, int TAG );
HRESULT __stdcall OnVScriptFinished( BSTR script_name, VS_RESULT result, int TAG );
HRESULT __stdcall OnNotifyClient ( int eventId, VARIANT eventBody, int TAG );


HRESULT Advise(IUnknown* pUnk)
{
        AtlGetObjectSourceInterface(pUnk,
                &m_libid, &m_iid, &m_wMajorVerNum, &m_wMinorVerNum);
        return DispEventAdvise(pUnk, &m_iid);
}

HRESULT Unadvise(IUnknown* pUnk)
{
        AtlGetObjectSourceInterface(pUnk,
                &m_libid, &m_iid, &m_wMajorVerNum, &m_wMinorVerNum);
        return DispEventUnadvise(pUnk, &m_iid);
}

...

};
```

Then, after you have established the connection with the server, you need to advise your implementation of the event interface:

```
        IVScriptEngine vscript_engine = NULL;

        try
        {
                vscript_engine = vscript ->GetVScriptEngine( "Test_1" );
        }
        catch ( _com_error& er )
        {
                SetStatusError( er );
        }


        if ( vscript_engine == NULL )
        {
                vscript = NULL;
                return E_FAIL;
        }

        CVSEngineSink vse_sink;
        HRESULT hr = vse_sink . Advise( vscript_engine ); // "Subscribe" for receiving events

        ...

        VS_RESULT res = SCRIPT_NOT_FOUND;
        try
        {
                res = (VS_RESULT)vscript_engine ->RunVScript();
        }
        catch ( _com_error& er)
        {
                SetStatusError( er );
        }


        // Tear connection with the test case
        vse_sink.Unadvise( vscript_engine );

        ...
```

```
VBA: ( MS Excel )

        Public SASTracer As SASAnalyzer
        Public Trace   As SASTrace
        Public GVSEngine As VScriptEngine

        '
        ' VSEngineEventsModule - is a special class implementing VSE event handlers.
        ' It should have in global declaration section the line like this:
        ' Public WithEvents VSEEvents As VScriptEngine
        '
        Dim X As New VSEngineEventsModule...

        Private Sub RunVScritButton_Click()
                Dim VSEngine As VScriptEngine
                Dim IVScript As ISASVerificationScript
                Dim ScriptName, fileToOpen As String


        ScriptName = ThisWorkbook.Sheets("Sheet1").Cells(2, 2)

        If SASTracer Is Nothing Then
                Set SASTracer = New SASAnalyzer

                If SASTracer Is Nothing Then
                        MsgBox "Unable to connect to SASTracer", vbExclamation
                Exit Sub
            End If
        End If


        fileToOpen = ThisWorkbook.Sheets("Sheet1").Cells(1, 2)
        Set Trace = SASTracer.OpenFile( fileToOpen )

        Set IVScript = Trace       'Get the IfcVerificationScript interface
        Set VSEngine = IVScript.GetVScriptEngine( ScriptName )

        ' "Subscribe" for receiving VSE events -
        ' the X variable ( an instance of VSEngineEventsModule class ) will handle them.
        '
        Set X.VSEEvents = VSEngine

        ...

        VSEngine.Tag = 12          ' Assign a tag for VSE object
        VSEngine.RunVScript         ' Run verification script

        Set X.VSEEvents = Nothing  ' "Unsubscribe" for receiving VSE events
        Set VSEngine = Nothing     ' Release external
        Set IVScript = Nothing     ' objects...
End Sub
```

9.1.1 _ IVScriptEngineEvents::OnVScriptReportUpdated

```
HRESULT OnVScriptReportUpdated (
       [in] BSTR newLine,
       [in] int TAG )
```

Fired when running a verification script, calls the *ReportText(newLine)* function (please refer to the *SASTracer Verification Script Engine Manual* for details on the *ReportText* function).

Parameters

`newLine`	– new portion of text reported by the verification script
`TAG`	– the VSE object's tag

Return values

Remarks

Make sure that C++ event handlers have *__stdcall* calling convention.

Example

C++:
```
        HRESULT __stdcall OnVScriptReportUpdated (BSTR newLine, int TAG )
        {
               TRACE( "Line: %s, TAG: %d\n", newLine, TAG );
               . . .

               return S_OK;
        }
```

VBA (MS Excel):
```
        Public WithEvents VSEEvents As VScriptEngine
        Public LineIndex As Integer
        . . .
        Private Sub VSEEvents_OnVScriptReportUpdated(ByVal newLine As String, ByVal Tag As Long)

            ThisWorkbook.Sheets("Sheet1").Cells(LineIndex, 1) = newLine
            LineIndex = LineIndex + 1

        End Sub
```

9.1.2 _ IVScriptEngineEvents:: OnVScriptFinished

```
HRESULT OnVScriptFinished (
       [in] BSTR script_name,
       [in] VS_RESULT result,
       [in] int TAG )
```

Fired when the verification script stops running.

Parameters

`script_name`	- the name of the verification script
`result`	- the result of the "verification", see *ISASVerificationScript::RunVerificationScript* method, Page 43, for details
`TAG`	- the VSE object's tag

Return values

Remarks

Make sure that C++ event handlers have *__stdcall* calling convention.

Example

```
C++:
       HRESULT __stdcall CComplTestSink::OnVScriptFinished(
              BSTR script_name,
              VS_RESULT result, int TAG )
       {
              USES_CONVERSION;

              TCHAR tmp[220];
              sprintf( tmp, "Script completed, name : %s, result = %d, TAG = %d",
                     W2A(script_name),
                     result, TAG );

              . . .

              return S_OK;
       }



VBA (MS Excel):

       Public WithEvents VSEEvents As VScriptEngine

       . . .

       Private Sub VSEEvents_OnVScriptFinished( ByVal script_name As String,
              ByVal result As SASAutomationLib.VS_RESULT,
              ByVal Tag As Long )

           Dim ResString As String
           ResString = "Script name : " & script_name & ", result = " &
              CStr(result) & ", TAG = " & CStr(Tag)

           ThisWorkbook.Sheets("Sheet1").Cells(7, 2) = ResString
       End Sub
```

9.1.3 _ IVScriptEngineEvents:: OnNotifyClient

```
HRESULT OnNotifyClient(
       [in] int eventId,
       [in] VARIANT eventBody,
       [in] int TAG )
```

Fired when running a verification script, calls the *NotifyClient()* function.

Parameters

`eventId`	- the event Id
`eventBody`	- the body of event packed in a VARIANT object
`TAG`	- the VSE object's tag

Return values

Remarks

The information packed in the event body is opaque for VSE – it only packs the information given to *NotifyClient()* function inside of verification script into a VARIANT object and sends it to client applications. See the *SASTracer Verification Script Engine Manual* for details about the *NotifyClient()* script function.

Example

```
SASTracer Verification script:

        ProcessEvent()
        {
            . . .
            NotifyClient( 2, [in.Index, in.Level, GetChannelName(), GetEventName(), TimeToText(
        in.Time )] );
            . . .
        }

VBA (MS Excel):
        Public WithEvents VSEEvents As VScriptEngine
        . . .

        Private Sub VSEEvents_OnNotifyClient( ByVal eventId As Long,
                       ByVal eventBody As Variant,
                       ByVal Tag As Long )
                Dim Col As Integer
                Dim Item As Variant


                ThisWorkbook.Sheets("Sheet1").Cells(LineIndex, 1) = eventId

                If IsArray(eventBody) Then
                    Col = 3

                    For Each Item In eventBody
                        ThisWorkbook.Sheets("Sheet1").Cells(LineIndex, Col) = Item
                        Col = Col + 1
                    Next
                Else
                    ThisWorkbook.Sheets("Sheet1").Cells(LineIndex, 2) = eventBody
                End If

                LineIndex = LineIndex + 1
        End Sub
```

10 SASAnalyzer Object Events

10.1 _IAnalyzerEvents Dispinterface

In order to retrieve the events from a *SASAnalyzer* object, you must implement the *_IAnalyzerEvents* interface. Since this interface is the default source interface for the *SASAnalyzer* object, there is a very simple implementation from such languages like Visual Basic, VBA, VBScript, WSH, etc.

Some script engines impose restrictions on handling events from “indirect” automation objects in typeless script languages (when the automation interface to the object is obtained from a call of some method rather than from a creation function—like *CreateObject()* in VBScript). The SASTracer provides a special COM class allowing receiving and handling notifications from the VSE object even in script languages not supporting event handling from "indirect" objects. Please refer to *CATCAnalyzerAdapter*, on page 86, for details.

C++ implementation used in the examples below utilizes a sink object by deriving it from *IdispEventImpl*, but not specifying the type library as a template argument. Instead, the type library and default source interface for the object are determined using *AtlGetObjectSourceInterface()*. A *SINK_ENTRY()* macro is used for each event from each source interface that is to be handled:

```
class CAnalyzerSink : public IDispEventImpl<IDC_SRCOBJ, CAnalyzerSink>
{
BEGIN_SINK_MAP(CAnalyzerSink)
        //Make sure the Event Handlers have __stdcall calling convention
        SINK_ENTRY(IDC_SRCOBJ, 1, OnTraceCreated)
        SINK_ENTRY(IDC_SRCOBJ, 2, OnStatusReport)
END_SINK_MAP()
. . .
}
```

Then, after you’ve established a connection with the server, you need to advise as to your implementation of the event interface:

```
hr = CoCreateInstance( CLSID_SASAnalyzer, NULL,
         CLSCTX_SERVER, IID_ISASAnalyzer, (LPVOID *)&m_poSASAnalyzer );

poAnalyzerSink = new CAnalyzerSink();

// Make sure the COM object corresponding to pUnk implements IProvideClassInfo2 or
// IPersist*. Call this method to extract info about source type library if you
// specified only 2 parameters to IDispEventImpl
hr = AtlGetObjectSourceInterface(m_poSASAnalyzer, &poAnalyzerSink->m_libid,
                      &poAnalyzerSink->m_iid, &poAnalyzerSink->m_wMajorVerNum,
                      &poAnalyzerSink->m_wMinorVerNum);

 if ( FAILED(hr) )
        return 1;

// connect the sink and source, m_poSASAnalyzer is the source COM object
hr = poAnalyzerSink->DispEventAdvise(m_poSASAnalyzer, &poAnalyzerSink->m_iid);

if ( FAILED(hr) )
        return 1;
```

10.1.1 _IAnalyzerEvents::OnTraceCreated

```
HRESULT OnTraceCreated (
       [in] IDispatch* trace )
```

Fired when a trace is created. This event is a result of *IAnalyzer::StartRecording* and *IAnalyzer::StopRecording* method calls (on pages 10 and 12).

Parameters

`trace`	– interface pointer to the *SASTrace* object

Remarks

Make sure the event handlers have *__stdcall* calling convention.

Example

```
VBScript:

        <OBJECT
                ID = Analyzer
                CLASSID = "clsid: 297CD804-08F5-4A4F-B3BA-779B2654B27C "
        >
        </OBJECT>
        <P ALIGN=LEFT ID=StatusText></P>

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Dim CurrentTrace
        Sub Analyzer_OnTraceCreated(ByRef Trace)
                On Error Resume Next
                Set CurrentTrace = Trace
                If Err.Number <> 0 Then
                        MsgBox Err.Number & ":" & Err.Description
                End If
                StatusText.innerText = "Trace '" & CurrentTrace.GetName & "' created"
        End Sub
        -->
        </SCRIPT>

C++:
        HRESULT __stdcall OnTraceCreated( IDispatch* trace )
        {
                ISASTrace* sas_trace;
                HRESULT hr;
                hr = trace->QueryInterface( IID_ISASTrace, (void**)&sas_trace );

                if (FAILED(hr))
                {
                        _com_error er(hr);
                        if (er.Description().length() > 0)
                                ::MessageBox( NULL, er.Description(), _T("SASTracer client"), MB_OK
                        );
                        else
                                ::MessageBox( NULL, er.ErrorMessage(),_T("SASTracer client"), MB_OK
                        );
                        return hr;
                 }

                . . .

                return hr;
        }
```

10.1.2 _IAnalyzerEvents::OnStatusReport

```
HRESULT OnStatusReport (
      [in] short subsystem,
      [in] short state,
      [in] long percent_done )
```

Fired when there is a change in the analyzer's state, or there is a change in progress (percent_done) of the analyzer's state.

Parameters

`subsystem` – subsystem sending event has the following values:

`RECORDING_PROGRESS_REPORT`	(1)	– recording subsystem
`GENERATION_PROGRESS_REPORT`	(2)	– generation subsystem

`state` – current analyzer state; has the following values:

if the *subsystem* is `RECORDING_PROGRESS_REPORT`:

`ANALYZERSTATE_IDLE`	(-1)	– idle
`ANALYZERSTATE_WAITING_TRIGGER`	(0)	– recording in progress, analyzer is waiting for trigger
`ANALYZERSTATE_RECORDING_TRIGGERED`	(1)	– recording in progress, analyzer triggered
`ANALYZERSTATE_UPLOADING_DATA`	(2)	– uploading in progress
`ANALYZERSTATE_SAVING_DATA`	(3)	– saving data in progress

If the *subsystem* is `GENERATION_PROGRESS_REPORT`:

`ANALYZERSTATE_GEN_IDLE`	(400)	- Generator is idle
`ANALYZERSTATE_GEN_DOWLOADING`	(401)	- Generator is downloading object code
`ANALYZERSTATE_GEN_GENERATING`	(402)	- Generator is working
`ANALYZERSTATE_GEN_PAUSED`	(403)	- Generator is paused

`percent_done` – shows the progress of currently performing operation

if *subsystem* is `RECORDING_PROGRESS_REPORT`:

- when analyzer state is ANALYZERSTATE_IDLE this parameter is not applicable
- when analyzer state is ANALYZERSTATE_WAITING_TRIGGER or ANALYZERSTATE_RECORDING_TRIGGERED this parameter shows analyzer memory utilization
- when analyzer state is `ANALYZERSTATE_UPLOADING_DATA` this parameter shows the percent of data uploaded
- when analyzer state is `ANALYZERSTATE_SAVING_DATA` this parameter shows the percent of data saved

if *subsystem* is `GENERATION_PROGRESS_REPORT`:

- represent current position of the script execution

Return values

Remarks

Make sure the event handlers have **__***stdcall* calling convention.

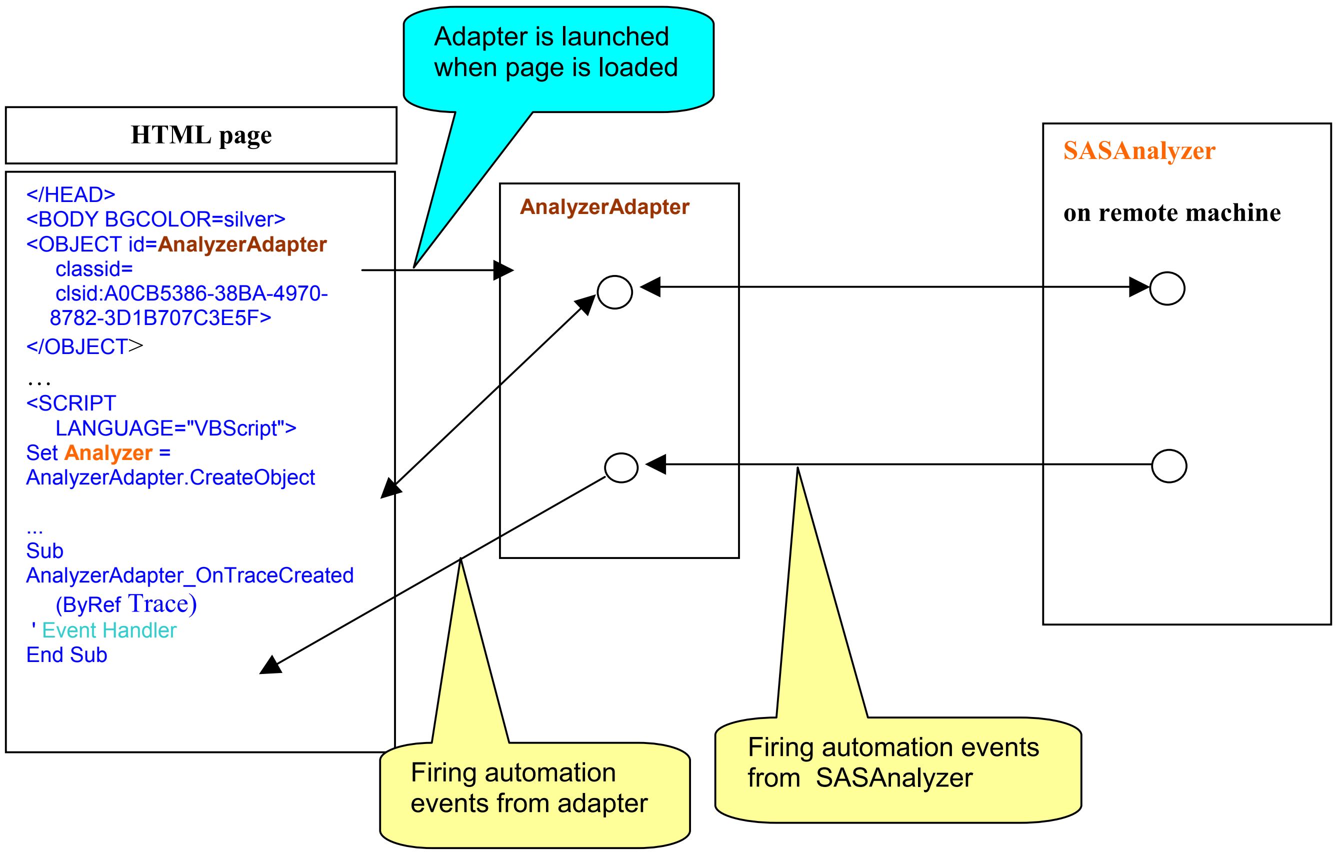
Example

```
VBScript:

        <OBJECT
                ID = Analyzer
                CLASSID = "clsid: 0B179BB7-DC61-11d4-9B71-000102566088 "
        >
        </OBJECT>
        <P ALIGN=LEFT ID=StatusText></P>

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Function GetRecordingStatus(ByVal State, ByVal Percent)
                Select Case State
                        Case -1: GetRecordingStatus = "Idle"
                        Case  0: GetRecordingStatus = "Recording - Waiting for trigger"
                        Case  1: GetRecordingStatus = "Recording - Triggered"
                        Case  2: GetRecordingStatus = "Uploading"
                        Case  3: GetRecordingStatus = "Saving Data"
                        Case Else: GetRecordingStatus = "Invalid recording status"
                End Select
                GetRecordingStatus = GetRecordingStatus & ", " & Percent & "% done"
        End Function


        Dim RecordingStatus
        Sub Analyzer_OnStatusReport(ByVal System, ByVal State, ByVal Percent)
                Select Case System
                        Case 1  RecordingStatus = GetRecordingStatus( State, Percent )
                End Select

        End Sub
        -->
        </SCRIPT>


C++:
        #define RECORDING_PROGRESS_REPORT             ( 1 )

        #define ANALYZERSTATE_IDLE                    ( -1 )
        #define ANALYZERSTATE_WAITING_TRIGGER         ( 0 )
        #define ANALYZERSTATE_RECORDING_TRIGGERED     ( 1 )
        #define ANALYZERSTATE_UPLOADING_DATA          ( 2 )
        #define ANALYZERSTATE_SAVING_DATA             ( 3 )

        HRESULT __stdcall OnStatusReport( short subsystem, short state, long percent_done )
        {
                switch ( subsystem )
                {
                case RECORDING_PROGRESS_REPORT:
                        UpdateRecStatus( state, percent_done );
                        break;
                }
                TCHAR buf[1024];
                _stprintf( buf, _T("%s"), m_RecordingStatus );
                ::SetWindowText( m_hwndStatus, buf );

                return S_OK;
        }

        void UpdateRecStatus( short state, long percent_done )
        {
                TCHAR status_buf[64];
                switch ( state )
                {
                case ANALYZERSTATE_IDLE:
                        _tcscpy( status_buf, _T("Idle") );
                        break;
                case ANALYZERSTATE_WAITING_TRIGGER:
                        _tcscpy( status_buf, _T("Recording - Waiting for trigger") );
                        break;
                case ANALYZERSTATE_RECORDING_TRIGGERED:
                        _tcscpy( status_buf, _T("Recording - Triggered") );
                        break;
                case ANALYZERSTATE_UPLOADING_DATA:
                        _tcscpy( status_buf, _T("Uploading") );
                        break;
                case ANALYZERSTATE_SAVING_DATA:
                        _tcscpy( status_buf, _T("Saving data") );
                        break;
                default:
                        _tcscpy( status_buf, _T("Unknown") );
                        break;
                }
                _stprintf( m_RecordingStatus, _T("%s, done %ld%%"), status_buf, percent_done );
        }
```

11 CATCAnalyzerAdapter

CATCAnalyzerAdapter is an automation server that allows the launching and accessing of CATC analyzer automation servers. If all necessary DCOM settings and permissions are set on the remote server, the server can be run remotely over an IP network. The example below shows how the *CATCAnalyzerAdapter* is used as an intermediary between an HTML page and the SASTracer analyzer server.

The following diagram illustrates how this functionality works:


Adapter is launched when page is loaded
HTML page
SASAnalyzer
on remote machine
AnalyzerAdapter
Firing automation events from adapter
Firing automation events from SASAnalyzer

```
</HEAD>
<BODY BGCOLOR=silver>
<OBJECT id=AnalyzerAdapter
    classid=
    clsid:A0CB5386-38BA-4970-
   8782-3D1B707C3E5F>
</OBJECT>
…
<SCRIPT
    LANGUAGE="VBScript">
Set Analyzer =
AnalyzerAdapter.CreateObject

...
Sub
AnalyzerAdapter_OnTraceCreated
    (ByRef Trace)
 ' Event Handler
End Sub
```

The Class ID and App ID for *SASAnalyzer* object are the following.

Class ID	:	A0CB5386-38BA-4970-8782-3D1B707C3E5F
App ID	:	CATC.AnalyzerAdapter

Primary interface : *IAnalyzerAdapter*.

11.1 IAnalyzerAdapter Interface

11.1.1 IAnalyzerAdapter::CreateObject

```
HRESULT CreateObject (
        [in] BSTR class_id,
        [in, optional] BSTR host_name,
        [out, retval] IDispatch** ppNewObj )
```

This method instantiates the CATC analyzer object on a local or remote machine and attaches it to the adapter.

Parameters

`class_id`	- string representation of *classid* or *ProgId* (*clsid: 0B179BB7-DC61-11d4-9B71-000102566088* or *CATC.SASTracer* for *SASAnalyzer* object)
`host_name`	- the network name of the remote server where the analyzer object should be instantiated. Empty value means local host.
`ppNewObj`	- pointer to the created remote object, NULL if the object has not been instantiated or accessed.

Return values

Remarks

Only CATC analyzer COM servers can be instantiated through this method. The method *Detach*, below, should be called when the work with the remote object is completed. **NOTE**: The pointer returned in *ppNewObj* should be released separately.

Example

VBScript:

```
        </HEAD>
        <OBJECT id=AnalyzerAdapter
                classid=clsid:A0CB5386-38BA-4970-8782-3D1B707C3E5F>
        </OBJECT>
        ...
        <input type="button" value="Connect" name="BtnConnect">
        <INPUT NAME="RemoteServer">

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnConnect_onclick
                On Error Resume Next

                Set Analyzer = AnalyzerAdapter.CreateObject("CATC.SASTracer", RemoteServer.value )


                if Not Analyzer Is Nothing Then
                        window.status = "SASTracer connected"
                else
                        msg = "Unable to connect to SASTracer"
                        MsgBox msg, vbCritical
                        window.status = msg
                End If
        End Sub
        -->
        </SCRIPT>
WSH:

        ' Create CATC analyzer adapter first..
        Set AnalyzerAdapter = WScript.CreateObject("CATC.AnalyzerAdapter", "Analyzer_")

        RemoteServer = "EVEREST"
        Set Analyzer = AnalyzerAdapter.CreateObject("CATC.SASTracer", RemoteServer)

        Analyzer.StartRecording ( Analyzer.ApplicationFolder & "my.rec" )
        ...
```

11.1.2 IAnalyzerAdapter:: Attach

```
HRESULT Attach(
       [in] IDispatch* pObj )
```

This method attaches the CATC analyzer object to the adapter.

Parameters

`pObj`	– pointer to the CATC analyzer object to be attached.

Return values

Remarks

Only CATC analyzer COM servers can be attached to the adapter. If some other analyzer object were previously attached to the adapter, it will be detached by this call. When the analyzer object gets attached to the adapter, a client application using the adapter becomes able to handle automation events fired by the remote analyzer object through the adapter.

Example

```
VBScript:

        </HEAD>
        <OBJECT id=AnalyzerAdapter
               classid=clsid:A0CB5386-38BA-4970-8782-3D1B707C3E5F>
        </OBJECT>
        ...
        <input type="button" value="Connect" name="BtnConnect">

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnConnect_onclick
               On Error Resume Next

               Set Analyzer = CreateObject("CATC.SASTracer" ) 'VBScript function   creates object
        locally

               if Not Analyzer Is Nothing Then
                      AnalyzerAdapter.Attach Analyzer ' attach analyzer to the adapter

                      window.status = "SASTracer connected"
               else
                      msg = "Unable to connect to SASTracer"
                      MsgBox msg, vbCritical
                      window.status = msg
               End If
        End Sub


        -->
        </SCRIPT>

WSH:
        ' Create CATC analyzer adapter first..
        Set AnalyzerAdapter = WScript.CreateObject("CATC.AnalyzerAdapter", "Analyzer_")

        'VBScript functioncreates object locally
        Set Adapter = WScript.CreateObject("CATC.AnalyzerAdapter")
```

```
AnalyzerAdapter.Attach Analyzer ' Attach analyzer object to the adapter
Analyzer.StartRecording ( Analyzer.ApplicationFolder & "my.rec" )
...
```

11.1.3 IAnalyzerAdapter:: Detach

```
HRESULT Detach()
```

This method detaches the CATC analyzer object from the adapter.

Parameters

Return values

Remarks
This method detaches an analyzer object from the adapter. This method doesn't guarantee that all resources associated with the detached object will be freed. All existing pointers to that object should be released to destroy the remote object.

Example

```
VBScript:

        </HEAD>
        <OBJECT id=AnalyzerAdapter
                classid=clsid:A0CB5386-38BA-4970-8782-3D1B707C3E5F>
        </OBJECT>
        ...
        <input type="button" value="Connect"    name="BtnConnect">
        <input type="button" value="Disconnect" name="BtnDisconnect">
        <INPUT NAME="RemoteServer">

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnConnect_onclick
                On Error Resume Next

                Set Analyzer = AnalyzerAdapter.CreateObject("CATC.SASTracer", RemoteServer.value )


                if Not Analyzer Is Nothing Then
                        window.status = "SASTracer connected"
                else
                        msg = "Unable to connect to SASTracer"
                        MsgBox msg, vbCritical
                        window.status = msg
                End If
        End Sub

        Sub BtnDisconnect_OnClick
                AnalyzerAdapter.Detach   ' Detach the analyzer object from adapter
                Set Analyzer = Nothing   ' Release the pointer to the analyzer returned by
CreateOject()

                window.status = "SASTracer disconnected"
        End Sub
        -->
        </SCRIPT>

WSH:
        ' Create CATC analyzer adapter first..
        Set AnalyzerAdapter = WScript.CreateObject("CATC.AnalyzerAdapter", "Analyzer_")

        RemoteServer = "EVEREST"
        Set Analyzer = AnalyzerAdapter.CreateObject("CATC.SASTracer", RemoteServer)

        Analyzer.StartRecording ( Analyzer.ApplicationFolder & "my.rec" )
```

```
...
AnalyzerAdapter.Detach    ' - Disconnect the remote analyzer from the adapter
Set Analyzer = Nothing    ' - Release the analyzer ...

'Release the adapter ...
Set AnalyzerAdapter = Nothing
```

11.1.4 IAnalyzerAdapter::IsValidObject

```
HRESULT IsValidObject(
       [in] IDispatch *pObj,
       [out,retval] VARIANT_BOOL* pVal )
```

This method helps to determine whether some automation object can be attached to the adapter.

Parameters

`pObj`	- pointer to the object validated
`pVal`	- pointer to the varable receiving result. TRUE if the validated object can be attached, FALSE otherwise

Return values

Remarks

Only CATC analyzer COM servers can be attached to the adapter.

Example

```
VBScript:

        </HEAD>
        <OBJECT id=AnalyzerAdapter
               classid=clsid:A0CB5386-38BA-4970-8782-3D1B707C3E5F>
        </OBJECT>
        ...
        <input type="button" value="Connect"    name="BtnConnect">
        <input type="button" value="Disconnect" name="BtnDisconnect">
        <INPUT NAME="RemoteServer">

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnConnect_onclick

               'Launch MS Excel instead of SASTracer !!!
               Set Analyzer = CreateObject("Excel.Application")
               Analyzer.Visible = True

               If Not AnalyzerAdapter.IsValidObject( Analyzer ) Then
                      MsgBox "The object cannot be attached", vbCritical
                      Set Analyzer = Nothing
                      Exit Sub
               End If
        End Sub

        -->
        </SCRIPT>
```

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/