TELEDYNE LECROY
Everywhereyoulook™

Protocol Solutions Group

Automation API Reference Manual

for

USB Protocol Suite

Teledyne LeCroy Advisor T3, Mercury T2™/T2C™/T2P™, and Voyager M3i/M3x/M310™/M310C™/M310P™ /M310e™, M4x™

USB Protocol Suite Software Version 9.52 and above

Manual Version 6.24 September 2024

Document Disclaimer

The information contained in this document has been carefully checked and is believed to be reliable. However, no responsibility can be assumed for inaccuracies that may not have been detected.

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

Trademarks and Servicemarks

CATC Trace, Mercury, Advisor, and *Voyager* are trademarks of Teledyne LeCroy

Microsoft and Windows are registered trademarks of Microsoft Inc.

All other trademarks are property of their respective companies.

Copyright

Contents

1 Introduction

The USB Protocol Suite Automation is an Application Program Interface (API) that allows users to create scripts or programs of commands and run these scripts or programs locally or remotely over a network. The name Automation is derived from the goal of allowing engineers to automate test procedures.

The Automation API is intended to support a single (1) Teledyne LeCroy Analyzer/Exerciser unit.

The USB*Tracer*, USB Advisor, and Voyager USB Protocol Suite Automation API is composed of a command set that duplicates most of the functionality of the USB Protocol Suite Graphical User Interfaces. Automation is implemented through the use of scripts or programs that the user can write using late binding scripting languages such as VBScript and WSH or early binding languages such as C++. Teledyne LeCroy provides examples of scripts written in WSH, VBScript, and C++.

Once an Automation script or program has been created, it can be run on the PC attached to or transmitted to the PC over a network from a remote location. Automation uses the Distributed Component Object Model (DCOM) protocol to transmit automation commands over a network. When run over a network, the Host Controller is configured as a DCOM server and the remote PC is configured as a DCOM client.

Note:
If you are using any application that uses the Automation Interface to the USB Protocol Suite, and the system prompts you that it cannot write a trace file to disk:
1. Make sure that the trace-file destination folder has write/create permissions. (For example, the target directory might be the network file system, which typically does not have write/create permissions.)
2. Make sure that the Windows (or other) Firewall Settings for USB Protocol Suite are set to Public.

1.1 System Requirements

Automation is supported with USB Protocol Suite software version 3.0 or higher. If you have an older version of the software, you must upgrade. You can get a new version of software from the Teledyne LeCroy web site, **www.teledynelecroy.com/support**. The Automation software including examples is included in the installation of the USB Protocol Suite software.

IMPORTANT! To compile with the correct linkage to the USB Automation API code, you must move the **USBAutomation.tlb** file from the application's installed directory into a directory from which your compiler can access the file. For example, for Visual Studio, this typically is the directory that contains your project file and sources.

1.2 Setting Up Automation for Local Use

If you intend to run Automation on the Analyzer Host Controller (the PC attached to the Analyzer), you do not need to perform any special configuration. You can simply execute the scripts or programs you have created, and they run the Analyzer.

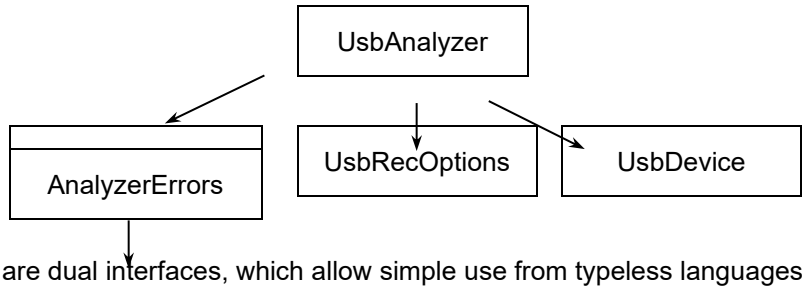
1.3 Setting Up Automation for Remote Use

If you intend to run automation remotely over a network, you must perform DCOM configuration. These steps are described in the Appendix.

The USB Protocol Suite API exposes the following objects and interfaces:

Objects	Interfaces	Description
UsbAnalyzer		
	IUsbAnalyzer IUsbAnalyzer3	Primary Analyzer interface
	_IAnalyzerEvents	Analyzer event source
UsbTrace		
	IUsbTrace IUsbTrace2	Trace file interfaces
UsbRecOptions		
	IUsbRecOptions	Recording options interface
UsbDevice		
	IUsbDevice	USB device interface
AnalyzerErrors		
	IAnalyzerErrors	Error collection interface

Only the **UsbAnalyzer** object is creatable at the top level (that is, via the **CoCreateInstance** call), instantiation of an object of other classes requires API calls. The following diagram represents the object dependencies:



All interfaces are dual interfaces, which allow simple use from typeless languages as well as from C++.

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

The examples of C++ code given in this document assume using the “import” technique of creating COM clients, using the corresponding **include** statement:

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

and creating appropriate wrapper classes in **.tli** and **.tlh** files by the compiler.

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

1.4 Error Return Codes

ANALYZERCOMERROR_UNABLEOPENFILE = 0x80040201
ANALYZERCOMERROR_INVALIDTRACEHANDLE = 0x80040202
ANALYZERCOMERROR_UNABLESTARTRECORDING = 0x80040203
ANALYZERCOMERROR_UNABLELOADDO = 0x80040204
ANALYZERCOMERROR_UNABLESTOPRECORDING = 0x80040205
ANALYZERCOMERROR_UNABLESAVE = 0x80040206
ANALYZERCOMERROR_EVENTSINKNOTINSTANTIATED = 0x80040207
ANALYZERCOMERROR_INVALIDPACKETNUMBER = 0x80040208

ANALYZERCOMERROR_INVALIDERROR = 0x80040209
ANALYZERCOMERROR_INVALIDVERSIONTYPE = 0x80040210
ANALYZERCOMERROR_ANALYZERNOTCONNECTED = 0x80040211
ANALYZERCOMERROR_UNABLESTARTGENERATION = 0x80040212
ANALYZERCOMERROR_WRONGCALL = 0x80040213
ANALYZERCOMERROR_UNABLETOMERGEFILES = 0x80040214
ANALYZERCOMERROR_LEVEL_NOT_AVAILABLE = 0x80040300
ANALYZERCOMERROR_VARIANT_TYPE_ERROR = 0x80040301
ANALYZERCOMERROR_REGISTER_WRITE_FAIL = 0x80040302
ANALYZERCOMERROR_DAC_CALIBRATION = 0x80040303
ANALYZERCOMERROR_INDEX_OUT_OF_RANGE = 0x80040304
ANALYZERCOMERROR_REGISTER_READ_FAIL = 0x80040305
ANALYZERCOMERROR_UNABLE_TO_LOAD = 0x80040306

1.5 Note Regarding VBS Return Values

A recent update to the Microsoft Visual Basic Interpreter causes it to no longer perform the implicit type conversions that it once did.

There is a very simple workaround for these issues. The "CLng" VBS function does needed conversions and helps to avoid the runtime errors. Example:

Old Code:

```
SwVersion = Analyzer.GetVersion(0)
```

New Code:

```
SwVersion = CLng(Analyzer.GetVersion(0))
```

2 Primary Dual Interface for Analyzer

2.1 IUsbAnalyzer Dual Interface

IUsbAnalyzer interface is the primary interface for the **UsbAnalyzer** object. It derives from the **IAnalyzer** interface that implements the common functionality for all Teledyne LeCroy Analyzers.

Class ID: **136D64A4-3CD5-4b41-974A-C7039E3FC292**
App ID: **CATC.USBTracer**
COM server: **UsbSuite.exe**
IUsbAnalyzer IID: **C37E7603-E3E5-48b4-8F79-080DBB543F0C**

2.1.1 IAnalyzer::GetVersion

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

Retrieves the current version of the specified subsystem.

Parameters

`version_type`	Subsystem whose version is requested; `EAnalyzerVersionType` enumerator has the following values: `ANALYZERVERSION_SOFTWARE ( 0 )` Software `ANALYZERVERSION_BUSENGINE ( 1 )` Bus engine `ANALYZERVERSION_FIRMWARE ( 2 )` Firmware
`analyzer_version`	Current version of subsystem requested. Value is encoded in BCD. Example: 0x0312 would mean version 3.12

Return values

`ANALYZERCOMERROR_INVALIDVERSIONTYPE`	Specified version type is invalid.
`ANALYZERCOMERROR_ANALYZERNOTCONNECTED`	Analyzer device is not connected.

Remarks

Example

```
WSH:
       CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
       Set Analyzer = WScript.CreateObject("CATC.USBTracer")
       SwVersion = CLng(Analyzer.GetVersion(0))
       SwOutput = ((SwVersion And &HFF00) / 256) & "." & String(2 - Len(SwVersion And &HFF),
"0") & (SwVersion And &HFF)
       BEVersion = CLng(Analyzer.GetVersion(1))
       BEOutput = ((BEVersion And &HFF00) / 256) & "." & String(2 - Len(BEVersion And &HFF),
"0") & (BEVersion And &HFF)
       FwVersion = CLng(Analyzer.GetVersion(2))
       FwOutput = ((FwVersion And &HFF00) / 256) & "." & String(2 - Len(FwVersion And &HFF),
"0") & (FwVersion And &HFF)
       MsgBox "Software: " & SwOutput & VbCrLf & "BusEngine: " & BEOutput & VbCrLf &
              "Firmware: " & FwOutput
```

C++:

```
HRESULT      hr;
IUsbAnalyzer*   poUsbAnalyzer;

// create UsbAnalyzer object
if ( FAILED( CoCreateInstance(
         CLSID_UsbAdvisor,
         NULL, CLSCTX_SERVER,
         IID_IUsbAnalyzer,
         (LPVOID *)&poUsbAnalyzer ) )
        return;

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

TCHAR buffer[20];
_stprintf( buffer, _T("Software version:%X.%02X"), // BCD, so Hex value -> Decimal value
                      HIBYTE(sw_version),
                      LOBYTE(sw_version) );
```

2.1.2 IAnalyzer::GetSerialNumber

```
HRESULT GetSerialNumber (
       [out, retval] WORD* serial_number );
```

Retrieves the serial number of the Analyzer device.

Parameters

Return values

ANALYZERCOMERROR_INVALIDVERSIONTYPE	Specified version type is invalid.
ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer device is not connected.

Remarks

Example

```
WSH:
        CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
        Set Analyzer = WScript.CreateObject("CATC.USBTracer")
        MsgBox "Serial number: " & CLng(Analyzer.GetSerialNumber())


C++:

        HRESULT         hr;
        IUsbAnalyzer*   poUsbAnalyzer;

        // create UsbAnalyzer object
        if ( FAILED( CoCreateInstance(
                 CLSID_UsbAdvisor,
                 NULL, CLSCTX_SERVER,
                 IID_IUsbAnalyzer,
                 (LPVOID *)&poUsbAnalyzer ) )
                return;

        WORD serial_number;
        try
        {
                serial_number = poUsbAnalyzer ->GetSerialNumber();
        }
        catch ( _com_error& er)
        {
                if (er.Description().length() > 0)
                ::MessageBox( NULL, er.Description(),  _T("UsbAnalyzer client"), MB_OK );
                else
                ::MessageBox( NULL, er.ErrorMessage(), _T("UsbAnalyzer client"), MB_OK );
                return 1;
        }

        TCHAR buffer[20];
        _stprintf( buffer, _T("Serial number: %X"),
                                HIBYTE(serial_number),
                                LOBYTE(serial_number) );
```

2.1.3 IAnalyzer::OpenFile

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

Opens a trace file.

Parameters

`file_name`	String providing the full pathname to the trace file
`trace`	Address of a pointer to the `UsbTrace` object primary interface

Return values

`ANALYZERCOMERROR_UNABLEOPENFILE`	Unable to open file.
`ANALYZERCOMERROR_WRONGCALL`	Error when trying to access the opened trace when it is being released and destroyed.

Remarks

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

In rare cases, the method may return `ANALYZERCOMERROR_WRONGCALL` error. This may happen when client code is trying to open an already opened trace file, while it is being released and destroyed. Subsequent method calls will reopen the trace file.

Example

```
WSH:
        CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
        Set Analyzer = WScript.CreateObject("CATC.USBTracer")
        Set Trace = Analyzer.OpenFile (CurrentDir & "Input\errors.usb")

C++:
        HRESULT         hr;
        IUsbAnalyzer*   poUsbAnalyzer;

        // create UsbAnalyzer object
        if ( FAILED( CoCreateInstance(
                 CLSID_UsbAdvisor,
                 NULL, CLSCTX_SERVER,
                 IID_IUsbAnalyzer,
                 (LPVOID *)&poUsbAnalyzer ) )
                return;

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

        // query for VTBL interface
        IUsbTrace* usb_trace;
        hr = trace->QueryInterface( IID_IUsbTrace, (LPVOID *)&usb_trace );
        trace->Release();

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

2.1.4 IAnalyzer::StartGeneration

```
HRESULT StartGeneration (
      [in] BSTR gen_file_name,
      [in] long gen_mode,
      [in] long loop_count );
```

Starts USB 2.0 traffic generation from the file.

Parameters

`gen_file_name` String providing the full pathname to the generation file.
If a valid name, the file is opened after any pre-existing Generation File is closed in accordance with the behavior set forth by Bit 31 in the **gen_mode** parameter.
If NULL string, it just closes any existing Generation File.

`gen_mode` Generation mode:
Bit 0: 0 = bitstream; 1 = IntelliFrame
Bit 31: 0 = Load **gen_file_name** only if:
1. Different than pre-existing Generation Filename OR
2. The **gen_file_name** file is already loaded but has been changed since the time it was loaded OR
3. The generation mode (bit 0) is different than the previous invocation. Leaving this bit set to 0 allows a file to be generated repeatedly without having to be re-parsed or downloaded to the Analyzer hardware, saving time.

1 = Reload **gen_file_name** unconditionally.

`loop_count` Number of times to repeat:
-1 = loop forever
1 through 16381 executes generation file that number of times.

Return values

`ANALYZERCOMERROR_UNABLEOPENFILE`	Unable to open file
`ANALYZERCOMERROR_UNABLESTARTGENERATION`	Unable to start generation (USB 3.0 generation is running, invalid state, or other)
`E_INVALIDARG`	

Remarks

Used to load a **.utg** file and begin generating traffic.

Example

2.1.5 IAnalyzer::StopGeneration

```
HRESULT StopGeneration ( );
```

Stops any current USB 2.0 generation in progress.

Return values

ANALYZERCOMERROR_UNABLESTARTGENERATION	Unable to stop generation (invalid state, etc.).

Remarks

Stop any current traffic generation.

Example

2.1.6 IAnalyzer::StartRecording

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

Starts recording with 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 the default recording options. To start recording without applying any recording options, pass “DO_NOT_CHANGE_RECORDING_OPTIONS”.

Return values

ANALYZERCOMERROR_EVENTSINKNOTINSTANTIATED	Event sink was not instantiated.
ANALYZERCOMERROR_UNABLESTARTRECORDING	Unable to start recording.

Remarks

After recording starts, this function returns. The Analyzer continues recording until it is finished or until a **StopRecording** method call is performed. During the recording, the events are sent to the event sink (see the IAnalyzerEvents interface).

The recording options file is the file with extension **.rec** created by the USB Protocol Suite application. You can create such a file by selecting **Setup – Recording Options…** from the USB Protocol Suite application menu, changing the recording options in the dialog, and selecting the **Save** button.

Example

VBScript:

```
<OBJECT
        RUNAT=Server
        ID = Analyzer
        CLASSID = "clsid:0B179BB3-DC61-11d4-9B71-000102566088"
>
</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++:

```
IUsbAnalyzer* usb_analyzer;
BSTR         ro_file_name;

. . .

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

2.1.7 IAnalyzer::StopRecording

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

Stops a recording started by the **StartRecording** method.

Parameters

abort_upload	TRUE, if caller wants to abort upload, no trace file is created. FALSE, if you want to upload the recorded trace

Return values

ANALYZERCOMERROR_EVENTSINKNOTINSTANTIATED	Event sink was not instantiated.
ANALYZERCOMERROR_UNABLESTOPRECORDING	Error stopping recording

Remarks

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

Example

VBScript:

```
<OBJECT
	RUNAT=Server
	ID = Analyzer
	CLASSID = "clsid:0B179BB3-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++:

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

2.1.8 IAnalyzer::MakeRecording

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

Makes a recording with the specified recording options file.

Parameters

`ro_file_name`	String providing the full pathname to recording options file; If the parameter is omitted, then recording starts with the default recording options.
`trace`	Address of a pointer to the `UsbTrace` object primary interface

Return values

Remarks

This method acts like **StartRecording** method but does not return until recording is completed. **UsbTrace** object is created via this method call, if call was successful.

The recording options file is the file with extension **.rec** created by the USB Protocol Suite application. You can create such a file by selecting **Setup – Recording Options…** from the USB Protocol Suite application menu, changing the recording options in the dialog, and selecting the **Save** button.

Example

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

C++:
        IDispatch* trace;
        IUsbAnalyzer* usb_analyzer;
        BSTR       ro_file_name;
        HRESULT    hr;

        . . .

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

        // query for VTBL interface
        IUsbTrace* usb_trace;
        hr = trace->QueryInterface( IID_IUsbTrace, (LPVOID *)&usb_trace );
        trace->Release();
```

2.1.9 IAnalyzer::LoadDisplayOptions

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

Loads display options that apply for traces opened or recorded later.

Parameters

`do_file_name`	String providing the full pathname to display options file

Return values

`ANALYZERCOMERROR_UNABLELOADDO`	Unable to load display options file.

Remarks

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

Display options file is the file with extension **.opt** created by the USB Protocol Suite application. You can create such a file by selecting **Setup – Display Options…** from the USB Protocol Suite application menu, changing the display options in the dialog, and selecting the **Save** button.

Example

See `ITrace::ApplyDisplayOptions`.

2.1.10 IAnalyzer::GetRecordingOptions

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

Retrieves the primary interface for access to recording options.

Parameters

`recording_options`	Address of a pointer to the `UsbRecOptions` object primary interface

Return values

Remarks

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

Example

```
WSH:

       Set Analyzer = WScript.CreateObject("CATC.USBTracer")
       Set RecOptions = Analyzer.GetRecordingOptions

C++:

IUsbAnalyzerPtr poUsbAnalyzer;

// Create UsbAnalyzer object.
if( FAILED( poUsbAnalyzer.CreateInstance( CLSID_UsbAdvisor ) ) )
     return;

// Get recording options
IDispatchPtr       rec_opt;
IUsbRecOptions Ptr usb_rec_opt;

try
{
     rec_opt = poUsbAnalyzer->GetRecordingOptions();
     usb_rec_opt = rec_opt; // Query for IUsbRecOptions interface

     // Use IUsbRecOptions methods
     ...
}
catch (_com_error& er)
{
     if (er.Description().length() > 0)
          ::MessageBox( NULL, er.Description(), _T("UsbAnalyzer client"), MB_OK );
     else
          ::MessageBox( NULL, er.ErrorMessage(), _T("UsbAnalyzer client"), MB_OK );
     return 1;
}
```

2.1.11 IUsbAnalyzer::StopRecordingAndWaitForTrace

```
HRESULT StopRecordingAndWaitForTrace
          ( [out, retval] IDispatch** trace );
```

Stops recording started by the **StartRecording** method and returns the pointer to the trace object created.

Parameters

`trace`	Address of a pointer to the `UsbTrace` object primary interface

Return values

`ANALYZERCOMERROR_UNABLESTOPRECORDING`	Error when stopping recording
`ANALYZERCOMERROR_WRONGCALL`	Error when trying to access trace when it is being destroyed.

Remarks
This method stops recording started by the **StartRecording** method and does not return until the recording is uploaded and a trace object related to this recording is created. In contrast, for **StopRecording**, no event is issued when recording is stopped by this method.

In rare cases, the method may return `ANALYZERCOMERROR_WRONGCALL` error. This may happen when recording stopped by itself (for example, because buffer is full), and the method call occurs when the recorded trace is being released and destroyed. Subsequent method calls will reopen the recorded trace. To avoid this issue, use a recording-buffer size that is big enough, so that the method call will occur when recording is still in progress and recording buffer is not full.

Example

WSH:

```
' Extract the script name.
ScriptName = Right(WScript.ScriptFullName, Len(WScript.ScriptFullName) -
InstrRev(WScript.ScriptFullName, "\") )

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

CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))

' In the code below, we perform four sequential recordings and
' save the traces recorded in the current folder.

For I = 1 To 4

    'Tell the CATC UWB analyzer to start recording.
    Analyzer.StartRecording ( CurrentDir & "Input\test_ro.rec" )

    ' Imitation of some activity - just sleep for 3 seconds.
    ' Some real traffic generation code might be put here.
    WScript.Sleep 3000

    ' Tell the analyzer to stop recording and give the trace acquired.
    Set Trace = Analyzer.StopRecordingAndWaitForTrace

    ' Save the trace in the current folder.
    Trace.Save CurrentDir & "Output\usb_seqrec_data" & CStr(I) & ".usb"
Next

Set Trace = Nothing

'Release the analyzer.
Set Analyzer = Nothing

MsgBox "Sequential recordings are completed." & String(2, 13) & "Look for recorded traces in the
folder : "  &  CurrentDir & "Output", 64, ScriptName
```

C++: (Raw code without using type library. All error handling is omitted for simplicity)

```
        IUsbAnalyzer* pAnalyzer = NULL;

        // Create UsbAnalyzer object.
        if ( FAILED( CoCreateInstance( CLSID_UsbAdvisor, NULL, CLSCTX_SERVER,
                IID_IUsbAnalyzer, (LPVOID *)&pAnalyzer ) )
                return;

        BSTR ro_file_name = SysAllocString( L"test_ro.rec" );
        BSTR bstr_trace_name;

        IDispatch* trace = NULL;

        for( int i = 0; i < 4; i++ )
        {
                if( FAILED( pAnalyzer ->StartRecording( ro_file_name ) )
                        return;  // Error handling and resources releasing should be done.

               // Fake generation.
               Sleep(3000);

               if( FAILED( Analyzer->StopRecordingAndWaitForTrace( &trace ) ) )
                        return;  // Error handling and resources releasing  should be done.


                IUsbTrace2* usbTrace = NULL;

                // Query for IUsbTrace2 interface.
                if( FAILED( trace->QueryInterface( IID_IUsbTrace2, (LPVOID*)usbTrace ) ) )
                        return; // Error handling and resources releasing should be done.


               OLECHAR trace_name[_MAX_PATH];
               swprintf ( trace_name, "test_data%u.usb", i );
               SysAllocString( bstr_trace_name );


               // Save the trace recorded to the current folder
               usbTrace->Save( bstr_trace_name );

               // Release the trace object.
               trace->Release();
               usbTrace->Release();

               SysFreeString( bstr_trace_name ); // Free BSTR.
        }

        pAnalyzer->Release();          // Release the analyzer object.
        SysFreeString( ro_file_name ); // Free BSTR.
```

C++: (Microsoft® C++ specific, using type library. All error handling is omitted for simplicity.)

```
IUsbAnalyzerPtr   pAnalyzer = NULL;
pAnalyzer.CreateInstance( __uuidof(UsbAdvisor) );

if( !pAnalyzer ) return;

IUsbTrace2Ptr trace = NULL;

for( int i = 0; i < 4; i++ )
{
        // Start recording using recording options file 'test_ro.rec' located in
        // the current folder.
        pAnalyzer->StartRecording( _bstr_t("test_ro.rec") );

        // Fake generation: just sleep for 3 seconds.
        Sleep(3000);

        trace = pAnalyzer->StopRecordingAndWaitForTrace();

        TCHAR trace_name[_MAX_PATH];
        sprintf( trace_name, "test_data%u.uwb", i );

        // Save the trace recorded in the current folder.
        trace->Save( _bstr_t(trace_name) );

        // Release the trace object.
        trace = NULL;
}
```

2.1.12 IUsbAnalyzer::ApplicationFolder (property)

```
HRESULT get_ApplicationFolder( [out, retval] BSTR *app_folder );
```

Retrieves the full local path to the folder where the USB Protocol Suite application was installed and registered as a COM server.

Parameters

`app_folder` Pointer to string variable indicating USB Protocol Suite application folder

Return values

Remarks

This property allows client applications to access some important files used by the USB Protocol Suite application, such as recording options, display options, and trace files, in places relative to the USB Protocol Suite application folder. If later the USB Protocol Suite application is re-installed to another location, the client code does not have to be changed to reflect this event.

Example

WSH:

```
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")

' Open trace file 'some_trace.usb' in the USB Suite folder.
Set Trace = Analyzer.OpenFile( Analyzer.ApplicationFolder & "some_trace.usb" )

' Save opened trace as 'some_trace1.usb' in the USB Suite folder.
Trace.Save Analyzer.ApplicationFolder & "some_trace1.usb"
...
```

C++:

```
HRESULT  hr;
IUsbAnalyzer* poUsbAnalyzer;

// Create UsbAnalyzer object.
if ( FAILED( CoCreateInstance( CLSID_UsbAdvisor, NULL, CLSCTX_SERVER,
               IID_IUsbAnalyzer, (LPVOID *)&poUsbAnalyzer ) )
       return;

BSTR app_folder;
hr = poUsbAnalyzer->get_ApplicationFolder( &app_folder );

// . . . Do something with app_folder.

SysFreeString( app_folder ); // Free BSTR resources.
poUsbAnalyzer->Release();    // Release analyzer object.
```

2.1.13 IUsbAnalyzer::ApplicationDataFolder (property)

```
HRESULT get_ApplicationDataFolder( [out, retval] BSTR *app_folder );
```

Retrieves the full local path to the folder for user-alterable data, such as scripts, traces, and default options files.

Parameters

`app_folder`	Pointer to string variable indicating the USB Protocol Suite user-modifiable data folder

Return values

Remarks

This property allows client applications to access some important files used by the USB Protocol Suite application, such as recording options, display options, and trace files, in places relative to the USB Protocol Suite application folder, Windows XP **Documents and Settings\All Users** folder, and Windows Vista **Users\Public** folder. If later the USB Protocol Suite application is re-installed to another location, the client code does not have to be changed to reflect this event.

Example

WSH:

```
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")

' Open trace file 'some_trace.usb' in a user-modiable folder.
Set Trace = Analyzer.OpenFile(Analyzer. ApplicationDataFolder & "some_trace.usb")

' Save opened trace as 'some_trace1.usb' in a user-modiable folder.
Trace.Save Analyzer. ApplicationDataFolder & "some_trace1.usb"
...
```

C++:

```
HRESULT  hr;
IUsbAnalyzer* poUsbAnalyzer;

// Create UsbAnalyzer object.
if ( FAILED( CoCreateInstance( CLSID_UsbAdvisor, NULL, CLSCTX_SERVER,
       IID_IUsbAnalyzer, (LPVOID *)&poUsbAnalyzer ) )
       return;

BSTR app_folder;
hr = poUsbAnalyzer->get_ApplicationDataFolder( &app_folder );

// . . .  Do something with app_folder.

SysFreeString( app_folder ); // Free BSTR resources.
poUsbAnalyzer->Release();    // Release analyzer object.
```

2.1.14 IUsbAnalyzer::StartUsb3Generation

```
HRESULT StartUsb3Generation([in,defaultvalue("")] BSTR gen_file_name);
```

Starts generating USB traffic.

Parameters

`gen_file_name`	USB 3.0 Exerciser script file name to execute. If this parameter is an empty string, the last executed script is repeated.

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED`	No Analyzer connected
`ANALYZERCOMERROR_UNABLESTARTGENERATION`	Analyzer cannot start for one of the following reasons: - Script has compiling error. - Analyzer is currently running or paused. - Analyzer is currently used by another client. - License for trainer device was not purchased. - USB 2.0 generation is running.

Remarks

This function returns after generation starts. Analyzer continues generating until it finishes or until the **StopUsb3Generation** or **PauseUsb3Generation** method call is performed.

During generation, events are sent to the event sink (see the _IAnalyzerEvents interface section).

The script file should have the file extension "**.usb3g**". It can be created by either the USB Protocol Suite text editor or any plain-text editor.

Example

```
WSH (1):

See Automation\wsh\Usb3Exerciser.vbs


WSH(2):

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Tell the USB 3.0 Voyager Exerciser to start generation.
Analyzer. StartUsb3Generation CurrentDir & " \Input\Usb3ScriptExample.usb3g"

Dim doneGeneration
doneGeneration = 0

' Repeat Generation 49 more times.
For RepeatCount = 1 To 49
        Do While doneGeneration = 0
                WScript.Sleep 100
        Loop
        Analyzer.StartUsb3Generation
        DoneGeneration = 0
Next

' Release the analyzer.
WScript.DisconnectObject Analyzer

' WScript.Echo "USBAnalyzer object has been disconnected."
Set Analyzer = Nothing

' WScript.Echo "Quitting WScript..."
WScript.Quit

' Handler of the event fired when recorded trace is created
' (after recording and uploading).
'
Sub Analyzer_OnStatusReport(ByVal subsystem, ByVal state, ByVal percent_done )
On Error Resume Next
  if state = 400 Then
    doneGeneration = 1
    WScript.Echo "Generation finished"
  End If
End Sub

VBScript:

        <OBJECT
                RUNAT=Server
                ID = Analyzer
                CLASSID = "clsid:136D64A4-3CD5-4b41-974A-C7039E3FC292"
        >
        </OBJECT>

        ...<INPUT TYPE=TEXT   VALUE="" NAME="PathToScript"> ...
        ...<INPUT TYPE=BUTTON VALUE="" NAME="BtnStartUSB3Generaion"> ...

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnStartUSB3Generaion_OnClick
                On Error Resume Next
                Analyzer.StartUsb3Generation PathToScript.value
                If Err.Number <> 0 Then
                        MsgBox Err.Number & ":" & Err.Description
                End If
        End Sub
        -->
        </SCRIPT>
```

C++:

```
IUsbAnalyzerPtr usb_analyzer;

. . .

try
{
	const char* gen_script_name =   "C:\\Usb3ScriptExample.usb3g";
	Usb_analyzer->StartUsb3Generation(_bstr_t( gen_script_name ) );
}
catch (_com_error& er)
{
	if (er.Description().length() > 0)
	::MessageBox( NULL, er.Description(),  _T("USBAnalyzer client"), MB_OK );
	else
	::MessageBox( NULL, er.ErrorMessage(), _T("USBAnalyzer client"), MB_OK );
	return 1;
}
```

2.1.15 IUsbAnalyzer::StopUsb3Generation

```
HRESULT StopUsb3Generation ();
```

Stops USB 3.0 generation started by the **StartUsb3Generation** method.

Parameters

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected

Remarks

Stops generation started by the **StartUsb3Generation** method. Generation stops if the analyzer is either running or paused, and the analyzer then returns to the idle state.

Example

```
WSH:

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Tell the CATC USB analyzer to start generation.
Analyzer.StartUsb3Generation "example.usb3g"
WScript.Sleep 3000

' Tell the analyzer to stop recording.
Analyzer.StopUsb3Generation()


VBScript:

	<OBJECT
		RUNAT=Server
		ID = Analyzer
		CLASSID = "clsid:136D64A4-3CD5-4b41-974A-C7039E3FC292"
	>
	</OBJECT>

	<SCRIPT LANGUAGE="VBScript">
	<!--
	Sub BtnStopUsb3Generation_OnClick
		On Error Resume Next
		Analyzer.StopUsb3Generation
		If Err.Number <> 0 Then
			MsgBox Err.Number & ":" & Err.Description
		End If
	End Sub
	-->
	</SCRIPT>
```

C++:

```
IUsbAnalyzerPtr usb_analyzer;

. . .

try
{
	usb_analyzer ->StopUsb3Generation();
}
catch (_com_error& er)
{
	if (er.Description().length() > 0)
	::MessageBox( NULL, er.Description(),  _T("UsbAnalyzer client"), MB_OK );
	else
	::MessageBox( NULL, er.ErrorMessage(), _T("UsbAnalyzer client"), MB_OK );
	return 1;
}
```

2.1.16 IUsbAnalyzer::PauseUsb3Generation

```
HRESULT PauseUsb3Generation ();
```

Pauses the currently running USB 3.0 generation.

Parameters

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected
ANALYZERCOMERROR_WRONGCALL	Analyzer is not running

Remarks

Pauses the generation started by the **StartUsb3Generation** method. Generation pauses if the Exerciser is running. However, the pause command cannot immediately pause traffic generation in all cases. During execution of a generation script (**.usb3g**) file, some **Send Packet** instructions might be queued in an output FIFO, and then pausing only takes effect after this buffer empties.

Example

```
WSH:

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Tell the CATC USB analyzer to start generation.
Analyzer.StartUsb3Generation "some_example.usb3g"
WScript.Sleep 3000

' Tell the analyzer to stop recording.
Analyzer.PauseUsb3Generation


VBScript:

        <OBJECT
                RUNAT=Server
                ID = Analyzer
                CLASSID = "clsid:136D64A4-3CD5-4b41-974A-C7039E3FC292"
        >
        </OBJECT>

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnPauseUsb3Generation_OnClick
                On Error Resume Next
                Analyzer.PauseUsb3Generation
                If Err.Number <> 0 Then
                        MsgBox Err.Number & ":" & Err.Description
                End If
        End Sub
        -->
        </SCRIPT>
```

C++:

```
IUsbAnalyzerPtr usb_analyzer;

. . .

try
{
	usb_analyzer->PauseUsb3Generation();
}
catch (_com_error& er)
{
	if (er.Description().length() > 0)
	::MessageBox( NULL, er.Description(),  _T("UsbAnalyzer client"), MB_OK );
	else
	::MessageBox( NULL, er.ErrorMessage(), _T("UsbAnalyzer client"), MB_OK );
	return 1;
}
```

2.1.17 IUsbAnalyzer::ResumeUsb3Generation

```
HRESULT ResumeUsb3Generation ();
```

Resumes the USB 3.0 generation paused by the **PauseUsb3Generation** command.

Parameters

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected

Remarks

Resumes the generation paused by the **PauseUsb3Generation** method. The generation returns to the running state from the paused state.

Example

```
WSH:

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Tell the CATC USB analyzer to start generation.
Analyzer.StartUsb3Generation "example.Usbg"
WScript.Sleep 3000

' Tell the analyzer to stop recording.
Analyzer.PauseUsb3Generation()
WScript.Sleep 3000
Analyzer.ResumeUsb3Generation()


VBScript:

        <OBJECT
                RUNAT=Server
                ID = Analyzer
                CLASSID = "clsid:136D64A4-3CD5-4b41-974A-C7039E3FC292"
        >
        </OBJECT>

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnResumeUsb3Generation_OnClick
                On Error Resume Next
                Analyzer.ResumeUsb3Generation
                If Err.Number <> 0 Then
                        MsgBox Err.Number & ":" & Err.Description
                End If
        End Sub
        -->
        </SCRIPT>
```

C++:

```
IUsb3AnalyzerPtr usb_analyzer;

. . .

try
{
        usb_analyzer->ResumeUsb3Generation();
}
catch (_com_error& er)
{
        if (er.Description().length() > 0)
        ::MessageBox( NULL, er.Description(),  _T("UsbAnalyzer client"), MB_OK );
        else
        ::MessageBox( NULL, er.ErrorMessage(), _T("UsbAnalyzer client"), MB_OK );
        return 1;
}
```

2.1.18 IUsbAnalyzer::UsbUnplugPlug

```
HRESULT UsbUnplugPlug();
```

Calling this method causes the VBus power between the Host and the Device connected through the Analyzer A and B USB ports to be broken for 1 second, simulating a unplug-plugin cycle. This is the recommended method of creating plug-in scenarios.

Parameters

Return values

Remarks

Example

2.2 IUsbAnalyzer3 interface

The **IUsbAnalyzer3** interface is the third interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 1D96A60E-AA3A-45e1-B6D5-C6AA27E65910

2.2.1 IUsbAnalyzer3:: IssueManualTrig

```
HRESULT IssueManualTrig ();
```

Issues manual trigger event to cause force trigger in analyzer engine.

Parameters

Return values

Remarks
Note that this method works with recording options that trigger mode is defined as manual trigger.

2.3 IUsbAnalyzer4 interface

The **IUsbAnalyzer4** interface is the forth interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : D793E9EE-58EE-4237-9D2A-41DBB61DCDF8

2.3.1 IUsbAnalyzer4::GetRecordingStatus

```
HRESULT GetRecordingStatus (
[out] VARIANT* is_trigged,
[out] VARIANT* is_done);
```

This method could be used to poll recording state of analyzer after recording start. The method returns useful running state information like happening trigger condition and recording finish. User may use this method to poll status of recording and go to next step according to polling result.

Parameters

is_trigged	Pointer to boolean variable indicating analyzer triggered or not.
is_done	Pointer to boolean variable indicating recording process is finished or not

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected

Remarks

A new method named as **WaitForRecordingStatus** is added to Anlyzer interface that could be used instead of this method. The new method doesn't block software normal operation and recommended to use new method.

2.3.2 IUsbAnalyzer4::ResetUsb3Trainer

```
HRESULT ResetUsb3Trainer ([in] int is_device_mode);
```

This method reset and initialize analyzer unit in "Host" trainer or "Device" trainer modes without runnig any script.

Parameters

is_device_mode	Specify desired mode. "1" for "Device" mode, "0" for "Host" mode

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected

Remarks

2.3.3 IUsbAnalyzer4::IsUsb3GenerationIdle

```
HRESULT IsUsb3GenerationIdle ([out] VARIANT* gen_idle);
```

This method could be used to poll generation state after starting generation. The method returns status of generation process. User may use this method to poll status of generation and go to next step according to polling result.

Parameters

`gen_idle` Pointer to boolean variable indicating generation is done or not.

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED` Analyzer is not connected

Remarks

A new method named as **`WaitForUsb3GenerationIdle`** is added to Anlyzer interface that could be used instead of this method. The new method doesn't block software normal operation and recommended to use new method.

2.3.4 IUsbAnalyzer4::SwitchVBus

```
HRESULT  SwitchVBus ( [in] long  switch_off);
```

This method provides facilty to switch on/off VBus in USB3 trainer without running any script.

Parameters

`switch_off` if "1" switch off VBus, other switch on VBus.

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED` Analyzer is not connected

Remarks

2.4 IUsbAnalyzer5 interface

The **IUsbAnalyzer5** interface is the fifth interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 47CCA8A8-C4AC-4b90-ABD1-16DC2A5351FE

2.4.1 IUsbAnalyzer5::BindUnit (Restricted and Unsupported: Not for Customer Use.)

```
HRESULT BindUnit ([in] WORD box_sn);
```

The method should be called when the API is used to control multiple anlayzer units. After creating any analyzer object (for each unit), it is required to bind analyzer object to desired unit by calling this method.

Note: This method is used by Teledyne LeCroy to enable the USB 3.1 Hub Compliance licensed feature on Voyagers. It is only enabled when that license key is purchased. **The use of this method by end customers outside of Hub Compliance is not supported by Teledyne LeCroy.**

Parameters

box_sn	Specifies target UNIT serial number.

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected
ANALYZERCOMERROR_UNABLESTARTRECORDING	Multiple Unit Automation License is not available

Remarks

Method will fail if it is not licensed.

2.4.2 IUsbAnalyzer5::MergeTraceFiles

```
HRESULT MergeTraceFiles   (
[in] BSTR file_name_1,
[in] BSTR file_name_2,
[in] BSTR output_file_name)
```

The method is used for merging two trace file in one trace.

Parameters

file_name_1	String parameter specifies first input file path
file_name_2	String parameter specifies second input file path
output_file_name	String parameter specifies output file path

Return values

```
      ANALYZERCOMERROR_UNABLETOMERGEFILES   Unable to merge files,
                  check files path and make sure they are not open
```

2.5 IUsbAnalyzer6 interface

The **IUsbAnalyzer6** interface is the sixth interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 866DA1BD-9AD4-4627-A32A-C0F229684300

2.5.1 IUsbAnalyzer6::WaitForUsb3GenerationIdle

```
HRESULT WaitForUsb3GenerationIdle (
[in] long wait_time_milisec,
[out] VARIANT* spent_time_milisec,
[out] VARIANT* is_done);
```

The method is introduced as replacement for legacy method “IsUsb3GenerationIdle”. The method polls internally for generation running and returns when generation finishes. Method will return if specified timeout is passed and generation is still running to prevent hangs/block conditions.

Parameters

wait_time_milisec	Specifies time-out for waiting before return in milliseconds
spent_time_milisec	Pointer to long value which returns spent time in this method
is_done	Pointer to boolean value indicating Generation done or time-out happened

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected

Remarks

2.5.2 IUsbAnalyzer6::WaitForRecordingStatus

```
HRESULT WaitForUsb3GenerationIdle (
[in] long wait_time_milisec,
[in] long desired_status,
[out] VARIANT* spent_time_milisec,
[out] VARIANT* is_done);
```

The method is introduced as replacement for legacy method “GetRecordingStatus”. The method polls internally for analyzer running and returns when analyzer reaches to desired state(s). Method will return if specified timeout is passed and analyzer has not reached to desired state(s).

Parameters

wait_time_milisec	Specifies time-out for waiting before return in milliseconds

`desired_state` Specify desired analyzer state(s) [see remarks] to return from method

`spent_time_milisec` Pointer to long value which returns spent time in this method

`is_done` Pointer to boolean value indicating desired state has been reached or time-out happened

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED` Analyzer is not connected

Remarks

State value could be combination of following values:

`RSTAT_DONE`	`0x4000`
`RSTAT_TRIGGERED`	`0x2000`
`RSTAT_WAITING_FOR_SYNC_REC`	`0x1000`

2.6 IUsbAnalyzer7 interface

The **IUsbAnalyzer7** interface is the seventth interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 13D7A1C9-6753-42A5-BFE0-6F55FB0C0FA4

2.6.1 IUsbAnalyzer7::GetLicensedCapability

```
HRESULT GetLicensedCapability(
[in] BSTR feature_title,
[out, retval] BOOL* is_licensed)
```

The method provides the cabability to identify license availabilty of any feature in any connected USBAnalyzer device. This method helps API users to change/limit their own software functionality according to the licensed features in the connected unit.

Parameters

`feature_title` Specifies the feature name in a string. The list of available features can be found in the license dialog in Teledyne LeCroy's "USB Protocol Suite" application.

`is_licensed` Pointer to boolean value indicates whether the feature is available or not.

Return values

`E_INVALIDARG` feature_name parameter is not a valid feature title

Remarks

If there is no connected device, the is_licensed parameter returns false for any feature. Therefore, it is required that you check the device connection before any license check to get a valid result.
If there is more than one device, the API should call the BindUnit() method to bind the object to a specific device before calling this method, otherwise the methods returns the minimum license from all connected devices.

2.7 IUsbAnalyzer8 interface

The **IUsbAnalyzer8** interface is the eightth interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 3A362B48-4222-403A-9B65-CED0E0A59DE8

2.7.1 IUsbAnalyzer8::GetHardwareInfo

```
HRESULT GetHardwareInfo (
 [out, retval] int* info)
```

The method returns the product hardware type. The UsbAnalyzer software and automation API is capable of working with different types of USB analyzer hardware. However, because some capablities are different between different products and analyzer hardware, API users may use this method to find what analyzer hardware is connected.

Parameters

`info` Pointer to int value indicates the analyzer hardware type.

The value enumeration is:

```
TARGETANALZYER_5K          = 0,
TARGETANALZYER_2500        = 1,
TARGETANALZYER_2500H       = 2,
TARGETANALZYER_ADVISOR     = 3,
TARGETANALZYER_MOBILE_HS   = 4,
TARGETANALZYER_MOBILE_T2   = 5,
// The above analyzers are no longer supported

TARGETANALZYER_VOYAGER     = 6,
TARGETANALZYER_ADVISOR_T3  = 7,
TARGETANALZYER_VOYAGER_M3X = 8,
TARGETANALZYER_VOYAGER_M310 = 9,
TARGETANALZYER_reserved    = 10,
TARGETANALZYER_MERCURY_T2  = 11,
TARGETANALZYER_ reserved2  = 12,
TARGETANALZYER_VOYAGER_M310C= 13,
TARGETANALZYER_MERCURY_T2C = 14,
TARGETANALZYER_MERCURY_T2P = 15,
TARGETANALZYER_VOYAGER_M310P = 16
TARGETANALZYER_VOYAGER_M4X = 17
TARGETANALZYER_VOYAGER_M310E = 18
```

Return values

`ANALYZERCOMERROR_INVALIDVERSIONTYPE` If there is no connected unit, or more than one unit connected, or the connected hardware could not be identified.

Remarks

If there is no connected device, the method returns an error.
If there is more than one device, the API should call the BindUnit() method to bind the object to a specific device before calling this method, otherwise the methods returns an error.

2.7.1 IUsbAnalyzer8::IsSupportSSP

```
HRESULT IsSupportSSP (
 [out, retval] BOOL* is_support)
```

The method returns that product supports USB3.1 SSP protocol or not. This includes both analyzer/exerciser engines. Please noted that in addition to supporting SSP protocol by product, specific licenses should be included to be able to work with SSP protcol engines.

Parameters

`is_suppot` Pointer to BOOL value indicates SSP support.

Return values

`ANALYZERCOMERROR_INVALIDVERSIONTYPE` If there is no connected unit, or more than one unit connected, or the connected hardware could not be identified.

Remarks

If there is no connected device, the method returns an error.
If there is more than one device, the API should call the BindUnit() method to bind the object to a specific device before calling this method, otherwise the methods returns an error.

2.8 IusbAnalyzer9 interface

The **IUsbAnalyzer9** interface is the 9th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 38024874-326B-4C5E-A5A1-820B0311E7DF

2.8.1 IusbAnalyzer9::ApplyRecordingOptions

```
HRESULT ApplyRecordingOptions
       ( [in] BSTR ro_file_name );
```

This method applies recording option settings before start recording/exerciser. This method is another alternative to passing recordng options through StartRecording() method and is used in scenarioes that analyzer hardware needs some other initializations like CrossSync settings before recording start.

Parameters

`ro_file_name` reference to desired recording option path.

Return values

`ANALYZERCOMERROR_UNABLESTARTRECORDING` If loading recording option fails for any reason.

Remarks

2.9 IusbAnalyzer10 interface

The **IUsbAnalyzer8** interface is the 10th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : B00652D4-B649-42AB-BCFF-13123DA33A00

2.9.1 IUsbAnalyzer10::SetPersistentDecoding

```
HRESULT SetPersistentDecoding ( [in] BOOL enable )
```

USB protocol analyzer provides capability to keep all decoding information persistent, so it would boost performance of opening files in next tries. This capality is **off by default** in calling OpenFile for opening trace files, but API users could use this new API method to enable/disable this functionality.

Parameters

`enable` A Boolean value which specify persistent decoding state.

Return values

Remarks

By enabling this capability, each files keeps some extra decoding information besides trace file in .tmp folder. This will increase hard disk space usage for each trace, so users need to consider that.

2.9.1 IUsbAnalyzer10::GetPersistentDecoding

```
HRESULT GetPersistentDecoding ( [out, retval] BOOL* is_enabled )
```

Parameters

`is_enabled` A pointer to a Boolean value which returns persistent decoding state.

Return values

Remarks

2.10 IUsbAnalyzer11 interface

The **IUsbAnalyzer11** interface is the 11th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 0D79B6FE-B74D-4441-804C-BFCB345EC736

2.10.1 IUsbAnalyzer11::StartPDGeneration

```
HRESULT StartPDGeneration ([in, defaultvalue("")] BSTR gen_file_name);
```

Starts generating Power Delivery traffic.

Parameters

gen_file_name	Power Delivery generation script file name to execute. If this parameter is an empty string, the last executed script is repeated.

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	No Analyzer connected
ANALYZERCOMERROR_UNABLESTARTGENERATION	Analyzer cannot start for one of the following reasons: - Script has compiling error. - Analyzer is currently running or paused. - Analyzer is currently used by another client. - License for trainer device was not purchased.

Remarks

This function returns after PD generation starts. Analyzer continues generating until it finishes or until the **StopPDGeneration** method is called.

The script file should have the file extension "**.updg**". Either the USB Protocol Suite text editor or any plain-text editor can create it.

Example

```
WSH:

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Tell the PD Exerciser to start generation.
Analyzer.StartPDGeneration "C:\\PDScriptExample.updg"

Dim doneGeneration
doneGeneration = 0

' Repeat Generation 49 more times.
For RepeatCount = 1 To 49
```

```
        Do While doneGeneration = 0
                WScript.Sleep 100
        Loop
        Analyzer.StartPDGeneration
        DoneGeneration = 0
Next

' Release the analyzer.
WScript.DisconnectObject Analyzer

' WScript.Echo "USBAnalyzer object has been disconnected."
Set Analyzer = Nothing

' WScript.Echo "Quitting WScript..."
WScript.Quit

' Handler of the event fired when recorded trace is created
' (after recording and uploading).
'
Sub Analyzer_OnStatusReport(ByVal subsystem, ByVal state, ByVal percent_done )
On Error Resume Next
  if state = 400 Then
    doneGeneration = 1
    WScript.Echo "Generation finished"
  End If
End Sub


VBScript:

        <OBJECT
                RUNAT=Server
                ID = Analyzer
                CLASSID = "clsid:136D64A4-3CD5-4b41-974A-C7039E3FC292"
        >
        </OBJECT>

        ...<INPUT TYPE=TEXT   VALUE="" NAME="PathToScript"> ...
        ...<INPUT TYPE=BUTTON VALUE="" NAME="BtnStartPDGeneraion"> ...

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


C++:

        IUsbAnalyzerPtr usb_analyzer;

        . . .

        try
        {
                const char* gen_script_name = "C:\\PDScriptExample.updg";
                Usb_analyzer->StartPDGeneration(_bstr_t( gen_script_name ) );
        }
        catch (_com_error& er)
        {
                if (er.Description().length() > 0)
                ::MessageBox( NULL, er.Description(),  _T("USBAnalyzer client"), MB_OK );
                else
                ::MessageBox( NULL, er.ErrorMessage(), _T("USBAnalyzer client"), MB_OK );
                return 1;
        }
```

2.10.2 IUsbAnalyzer11::StopPDGeneration

```
HRESULT StopPDGeneration();
```

Stops Power Delivery generation started by the **StartPDGeneration** method.

Parameters

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected

Remarks

Stops generation started by the **StartPDGeneration** method. Generation stops if the analyzer is either running or paused, and the analyzer then returns to the idle state.

Example

```
WSH:

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Tell the PD Exerciser to start generation.
Analyzer.StartPDGeneration "example.updg"
WScript.Sleep 3000

' Tell the analyzer to stop recording.
Analyzer.StopPDGeneration()


VBScript:

        <OBJECT
                RUNAT=Server
                ID = Analyzer
                CLASSID = "clsid:136D64A4-3CD5-4b41-974A-C7039E3FC292"
        >
        </OBJECT>

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnStopUsb3Generation_OnClick
                On Error Resume Next
                Analyzer.StopPDGeneration
                If Err.Number <> 0 Then
                        MsgBox Err.Number & ":" & Err.Description
                End If
        End Sub
        -->
        </SCRIPT>


C++:

        IUsbAnalyzerPtr usb_analyzer;

        . . .

        try
        {
```

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

2.10.3 IUsbAnalyzer11::IsPDGenerationIdle

```
HRESULT IsPDGenerationIdle ([out] VARIANT* gen_idle);
```

This method could be used to poll Power Delivery generation state after starting generation. The method returns status of generation process. User may use this method to poll status of generation and go to next step according to polling result.

Parameters

`gen_idle` Pointer to boolean variable indicating generation is done or not.

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED` Analyzer is not connected

Remarks

Another method named as **`WaitForPDGenerationIdle`** is added to Anlyzer interface that could be used instead of this method. That method doesn't block software normal operation and recommended to use it.

2.10.1 IUsbAnalyzer11::ResumePDGeneration

```
HRESULT  ResumePDGeneration ();
```

Resumes the PD generation paused internally by generation script commands.

Parameters

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED` Analyzer is not connected

Remarks

The generation returns to the running state from the paused state after calling this method. Note that pausing Power Delivery generation is possible only by generation script. For more information, refer to the USB Power Delivery Exerciser Manual.

2.10.1 IUsbAnalyzer11::WaitForPDGenerationIdle

```
HRESULT WaitForPDGenerationIdle (
[in] long wait_time_milisec,
[out] VARIANT* spent_time_milisec,
[out] VARIANT* is_done);
```

The method polls internally for Power Delivery generation running and returns when generation finishes. Method will return if specified timeout is passed and generation is still running to prevent hangs/block conditions.

Parameters

`wait_time_milisec`	Specifies time-out for waiting before return in milliseconds
`spent_time_milisec`	Pointer to long value that returns spent time in this method
`is_done`	Pointer to long value indicating as follows: 0: Time-out happened. Non-Zero: The passed argument to the PD_Stop(ret_val) function in the PD-Exerciser script. A call to PD_Stop() without any arguments returns 256.

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED`	Analyzer is not connected

Remarks

2.11 IUsbAnalyzer12 interface

The **IUsbAnalyzer12** interface is the 12th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 475A317C-E69B-4CE1-8562-02F46F554BE3

2.11.1 IUsbAnalyzer12:: WaitForPDGenerationPause

```
HRESULT WaitForPDGenerationPause (
[in] long wait_time_milisec,
[out] VARIANT* spent_time_milisec,
[out] VARIANT* is_done);
```

The method polls internally for Power Delivery generation running and returns when generation is in Paused state. Method will return if specified timeout elapsed and generation is still paused to prevent hangs/block conditions.

Parameters

`wait_time_milisec`	Specifies time-out for waiting before return in milliseconds
`spent_time_milisec`	Pointer to long value that returns spent time in this method
`is_done`	Pointer to boolean value indicating Generation paused or time-out happened

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED`	Analyzer is not connected

Remarks

Use this method if you want client to have an opportunity to do some actions when generation is in paused state. You can use **ResumePDGeneration** to resume generation subsequently.

2.12 IUsbAnalyzer13 interface

The **IUsbAnalyzer13** interface is the 13th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 1EFE4CB0-F231-4994-A61B-6E89BB8F39CD

2.12.1 IUsbAnalyzer13:: WaitForPDGenerationPauseTag

```
HRESULT WaitForPDGenerationPauseTag (
[in] long wait_time_milisec,
[out] VARIANT* spent_time_milisec,
[in] VARIANT* tag,
[out] VARIANT* is_done);
```

The method polls internally for Power Delivery generation running and returns when generation is in paused state. Returned tag is the assigned number to the current paused state. Method will return if specified timeout elapsed and generation is still paused to prevent hangs/block conditions.

Parameters

`wait_time_milisec` Specifies time-out for waiting before return in milliseconds

`spent_time_milisec` Pointer to long value that returns spent time in this method

`tag` Pointer to long value that returns corresponding tag of the current pause state.

`is_done` Pointer to boolean value indicating Generation paused or time-out happened

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED` Analyzer is not connected

Remarks

Use this method if you want client to have an opportunity to do some actions when generation is in paused state. To use this method, generation script should assign different tag to each pause state so that client can differentiate between paused states. You can use **ResumePDGeneration** to resume generation subsequently.

2.13 IUsbAnalyzer16 interface

The **IUsbAnalyzer16** interface is the 16th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 6DA5368A-AC86-44BE-810A-914829BA3B5C

2.13.1 IUsbAnalyzer16::StartUSB4Generation

```
   HRESULT StartUSB4Generation ([in, defaultvalue("")] BSTR
gen_file_name);
```

Starts generating USB4 traffic.

Parameters

gen_file_name	USB4 generation script file name to execute. If this parameter is an empty string, the last executed script is repeated.

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	No Analyzer connected
ANALYZERCOMERROR_UNABLESTARTGENERATION	Analyzer cannot start for one of the following reasons: - Script has compiling error. - Analyzer is currently running or paused. - Analyzer is currently used by another client. - License for trainer device was not purchased.

Remarks

This function returns after USB4 generation starts. Analyzer continues generating until it finishes or until the **StopUSB4Generation** method is called.

The script file should have the file extension "**.usb4g**". Either the USB Protocol Suite text editor or any plain-text editor can create it.

Example

```
WSH:

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Tell the USB4 Exerciser to start generation.
Analyzer.StartUSB4Generation "C:\\USB4ScriptExample.usb4g"

Dim doneGeneration
doneGeneration = 0

' Repeat Generation 49 more times.
For RepeatCount = 1 To 49
       Do While doneGeneration = 0
              WScript.Sleep 100
       Loop
       Analyzer.StartUSB4Generation
       DoneGeneration = 0
Next
```

```
' Release the analyzer.
WScript.DisconnectObject Analyzer

' WScript.Echo "USBAnalyzer object has been disconnected."
Set Analyzer = Nothing

' WScript.Echo "Quitting WScript..."
WScript.Quit

' Handler of the event fired when recorded trace is created
' (after recording and uploading).
'
Sub Analyzer_OnStatusReport(ByVal subsystem, ByVal state, ByVal percent_done )
On Error Resume Next
  if state = 400 Then
    doneGeneration = 1
    WScript.Echo "Generation finished"
  End If
End Sub
```

VBScript:

```
        <OBJECT
                RUNAT=Server
                ID = Analyzer
                CLASSID = "clsid:136D64A4-3CD5-4b41-974A-C7039E3FC292"
        >
        </OBJECT>

        ...<INPUT TYPE=TEXT   VALUE="" NAME="PathToScript"> ...
        ...<INPUT TYPE=BUTTON VALUE="" NAME="BtnStartUSB4Generaion"> ...

        <SCRIPT LANGUAGE="VBScript">
        <!--
        Sub BtnStartUSB4Generaion_OnClick
                On Error Resume Next
                Analyzer.StartUSB4Generation PathToScript.value
                If Err.Number <> 0 Then
                        MsgBox Err.Number & ":" & Err.Description
                End If
        End Sub
        -->
        </SCRIPT>
```

C++:

```
        IUsbAnalyzerPtr usb_analyzer;

        . . .

        try
        {
                const char* gen_script_name = "C:\\USB4ScriptExample.usb4g";
                Usb_analyzer->StartUSB4Generation(_bstr_t( gen_script_name ) );
        }
        catch (_com_error& er)
        {
                if (er.Description().length() > 0)
                ::MessageBox( NULL, er.Description(),  _T("USBAnalyzer client"), MB_OK );
                else
                ::MessageBox( NULL, er.ErrorMessage(), _T("USBAnalyzer client"), MB_OK );
                return 1;
        }
```

2.13.2 IUsbAnalyzer16::StopUSB4Generation

```
HRESULT StopUSB4Generation();
```

Stops USB4 generation started by the **StartUSB4Generation** method.

Parameters

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected

Remarks

Stops generation started by the **StartUSB4Generation** method. Generation stops if the analyzer is either running or paused, and the analyzer then returns to the idle state.

Example

WSH:

```
Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Tell the USB4 Exerciser to start generation.
Analyzer.StartUSB4Generation "example.usub4g"
WScript.Sleep 3000

' Tell the analyzer to stop recording.
Analyzer.StopUSB4Generation()
```

VBScript:

```
<OBJECT
        RUNAT=Server
        ID = Analyzer
        CLASSID = "clsid:136D64A4-3CD5-4b41-974A-C7039E3FC292"
>
</OBJECT>

<SCRIPT LANGUAGE="VBScript">
<!--
Sub BtnStopUsb4Generation_OnClick
        On Error Resume Next
        Analyzer.StopUSB4Generation
        If Err.Number <> 0 Then
                MsgBox Err.Number & ":" & Err.Description
        End If
End Sub
-->
</SCRIPT>
```

C++:

```
IUsbAnalyzerPtr usb_analyzer;

. . .

try
{
        usb_analyzer ->StopUSB4Generation();
}
catch (_com_error& er)
```

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

2.13.3 IUsbAnalyzer16::ResumeUSB4Generation

```
HRESULT  ResumeUSB4Generation ();
```

Resumes the USB4 generation paused internally by generation script commands.

Parameters

Return values

ANALYZERCOMERROR_ANALYZERNOTCONNECTED	Analyzer is not connected

Remarks

The generation returns to the running state from the paused state after calling this function. Note that pausing USB4 generation is possible only by generation script commands. For more information, refer to the USB4 Exerciser Manual.

2.13.4 IUsbAnalyzer16::WaitForUSB4GenerationIdle

```
HRESULT WaitForUSB4GenerationIdle (
[in] long wait_time_milisec,
[out] VARIANT* spent_time_milisec,
[out] VARIANT* is_done);
```

Polls internally for USB4 generation running and returns when generation finishes. Method will return if specified timeout is passed and generation is still running to prevent hangs/block conditions.

Parameters

wait_time_milisec	Specifies time-out for waiting before return in milliseconds
spent_time_milisec	Pointer to long value that returns spent time in this method

`is_done` Pointer to boolean value indicating Generation done or time-out happened

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED` Analyzer is not connected

Remarks

2.13.5 IUsbAnalyzer16::WaitForUSB4GenerationPauseTag

```
HRESULT WaitForUSB4GenerationPauseTag (
[in] long wait_time_milisec,
[out] VARIANT* spent_time_milisec,
[in] VARIANT* tag,
[out] VARIANT* is_done);
```

Polls internally for USB4 generation running and returns when generation is in paused state. Returned tag is the assigned number to the current paused state. The function will return if the specified timeout elapsed and generation is still paused to prevent hangs/block conditions.

Parameters

`wait_time_milisec` Specifies time-out for waiting before return in milliseconds

`spent_time_milisec` Pointer to long value that returns spent time in this method

`tag` Pointer to long value that returns corresponding tag of the current pause state.

`is_done` Pointer to boolean value indicating Generation paused or time-out happened

Return values

`ANALYZERCOMERROR_ANALYZERNOTCONNECTED` Analyzer is not connected

Remarks

Use this method if you want a client to have an opportunity to do some actions when generation is in paused state. A generation script can assign different tag to each pause state so that the client can differentiate between paused states. You can use **ResumeUSB4Generation** to resume the generation subsequently.

2.14 UsbAnalyzer17 interface

The **IUsbAnalyzer17** interface is the 17th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : D6A8DC56-7503-11EC-90D6-0242AC120003

2.14.1 IUsbAnalyzer17::GetConnectedAnalyzersSN

```
HRESULT GetConnectedAnalyzersSN (
[out, retval] SAFEARRAY(VARIANT)* serial_numbers);
```

Returns serial numbers of all connected devices.

Parameters

serial_numbers	Serial numbers array for all connected devices.

Remarks

You can use this method to check the availability of a desired Analyzer.

Example

```
WSH:

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

' Get serial numbers of all connected devices and store them in an array
Dim array_sn
array_sn = Analyzer.GetConnectedAnalyzersSN

' If no Analyzer is connected, check for any connected device every 1 sec
While UBound(array_sn) < 0
    WScript.Sleep 1000
    array_sn = Analyzer.GetConnectedAnalyzersSN
Wend

' Display all serial numbers in the array
For i = 0 To UBound(array_sn)
    MsgBox array_sn(i)
Next

WScript.DisconnectObject Analyzer
Set Analyzer = Nothing
WScript.Quit
```

2.15 UsbAnalyzer18 interface

The **IUsbAnalyzer18** interface is the 18th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : D037764C-93E5-435D-A06E-208C8B8337D9

2.15.1 IUsbAnalyzer18::IsRestartRequired

```
HRESULT IsRestartRequired (
[in] int functionality, [out] VARIANT* restart);
```

Checks whether the analyzer must be restarted in order to use a specified functionality.

Parameters

`functionality`	Analyzer functionality to be queried 2 – USB4/TBT3 recording 4 – USB3 recording
`restart`	Pointer to boolean value indicating whether analyzer restart is required.

Remarks

This method is usually used in conjunction with `RestartAnalyzer`. On a Voyager M4x, the restart flag shall return True when the indicated functionality differs from what was previously selected in the Recording Options.

Example

See 2.15.2 IUsbAnalyzer18::RestartAnalyzer.

2.15.2 IUsbAnalyzer18::RestartAnalyzer

```
HRESULT RestartAnalyzer (
[in, defaultvalue(-1)] int functionality);
```

Initiates an analyzer restart sequence to enable a specified functionality.

Parameters

`functionality`	Specifies desired analyzer functionality -1 – Force analyzer restart 2 – Restart analyzer and enable USB4/TBT3 recording 4 – Restart analyzer and enable USB3 recording

Remarks

You can use this method to automatically restart the analyzer. On a Voyager M4x, the unit will boot up with the specified BE loaded and ready to capture.

Example

```
WSH:

' Analyzer Device Functionalities
Const DF_USB4_ANALYZER = 2
Const DF_USB3_ANALYZER = 4

Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")

'
' Ensure USB4 BusEngine is loaded and initialized
'

' Restart flag will be set to True if the current BE is USB3
Dim Restart
Analyzer.IsRestartRequired DF_USB4_ANALYZER, Restart

If Restart Then
    ' Restart analyzer with USB4 set as desire functionality
    Analyzer.RestartAnalyzer DF_USB4_ANALYZER

    ' We need to recreate the Analyzer object after the Restart
    WScript.DisconnectObject Analyzer
    Set Analyzer = WScript.CreateObject("CATC.UsbTracer", "Analyzer_")
End If

'
' Continue with USB4 Analyzer API call
' eg. Analyzer.StartUSB4Generation

. . .

WScript.Quit
```

2.16 UsbAnalyzer20 interface

The **IUsbAnalyzer20** interface is the 20th interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID : 2bc88cb6-61a4-11ee-8c99-0242ac120002

2.16.1 IUsbAnalyzer20::ImportSimPass

```
HRESULT ImportSimPass(
[in, defaultvalue("")] BSTR simpass_file_name)
```

Imports the SimPass file.

Parameters

simpass_file_name	Absolute address for the SimPass file.

Remarks

Example

```
VBScript:

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

Analyzer.ImportSimPass ("C:\SimPass\test.simpass")

WScript.DisconnectObject Analyzer
Set Analyzer = Nothing

WScript.Quit
```

2.17 UsbAnalyzer21 interface

The **IUsbAnalyzer21** interface is the 21st interface for the **UsbAnalyzer** object. It inherits and extends some analyzer-related functionality exposed via the **IUsbAnalyzer** interface.

IID: e40558b9-bddc-40cf-b755-2a6fe025b775

2.17.1 IUsbAnalyzer21::OpenUserFile

```
HRESULT OpenUserFile(
[in] BSTR file_name,
[in] UINT mode,
[out, retval] LONGLONG* handle)
```

Open a file with a specified name and mode. It returns a handle that can be used for subsequent file operations.

Note: The maximum number of files that can be open at the same time is 10.

Parameters

`file_name` This parameter specifies the name of the file to be opened. It can be:

- A name without a path, in which case the file will be located in the application's current directory.
- A relative path, relative to the application's data directory.
- An absolute path, specifying the exact location of the file.

`mode` This parameter specifies the mode in which the file should be opened. The mode can be a combination of the following constants:

Constant	Value	Description
ReadOnly	0x0001	The file is open for reading.
WriteOnly	0x0002	The file is open for writing. The file is truncated unless combined with ReadOnly, Append, or NewOnly.
ReadWrite	0x0003	The file is open for both reading and writing. This combines ReadOnly (0x0001) and WriteOnly (0x0002).
Append	0x0004	The file is opened in append mode, so all data is written to the end of the file.
Truncate	0x0008	If possible, the file is truncated before it is opened. All earlier contents of the file are lost.
Text	0x0010	When reading, end-of-line terminators are translated to '\n'. When writing, end-of-line terminators are translated to the local encoding (e.g., '\r\n' for Win32).
Unbuffered	0x0020	Any buffer in the device is bypassed.
NewOnly	0x0040	The operation fails if the file already exists. The file is created and opened only if it does not exist. This mode implies WriteOnly but can be combined with ReadWrite.
ExistingOnly	0x0080	The operation fails if the file does not exist. This flag must be specified alongside ReadOnly, WriteOnly, or ReadWrite. Using this flag with ReadOnly alone is redundant, as ReadOnly already fails when the file does not exist.

`handle` This is an output parameter that returns a handle to the opened file. The handle is a unique identifier that can be used in subsequent file operations.

Remarks

Example

VBScript:

```
' Create an instance of the COM object
Dim Analyzer
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")

' Define the file path and mode
Dim filePath, fileMode, fileHandle, writeContent, filePos, readContent, readLen
filePath = "C:\Users\Public\Public Documents\sample_file.txt"
fileMode = &H10 Or &H03 ' text mode 0x10 | ReadWrite 0x03

' Open the file
fileHandle = Analyzer.OpenUserFile(filePath, fileMode)

' Write to the file
writeContent = "This is a message from a COM client. Unusual chars @#$%^&*()!!~~"
Analyzer.WriteUserFile fileHandle, writeContent

' Seek the file
filePos = 0
Analyzer.SeekUserFile fileHandle, filePos

' Read from the file
readLen = 100 ' up to 100 chars
Analyzer.ReadUserFile fileHandle, readLen, readContent

msgbox readContent

Analyzer.CloseUserFile(fileHandle)
Set Analyzer = Nothing
WScript.Quit
```

2.17.2 IUsbAnalyzer21::CloseUserFile

```
HRESULT CloseUserFile(
[in] LONGLONG handle)
```

The CloseUserFile function closes the opened file that was previously opened using the OpenUserFile function.

Parameters

Handle	This parameter is the unique identifier assigned by the OpenUserFile API. It identifies the file that needs to be closed.

Remarks

Example

VBScript:

```
' Create an instance of the COM object
Dim Analyzer
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")

' Define the file path and mode
Dim filePath, fileMode, fileHandle, writeContent, filePos, readContent, readLen
filePath = "C:\Users\Public\Public Documents\sample_file.txt"
fileMode = &H10 Or &H03 ' text mode 0x10 | ReadWrite 0x03

' Open the file
fileHandle = Analyzer.OpenUserFile(filePath, fileMode)

' Write to the file
writeContent = "This is a message from a COM client. Unusual chars @#$%^&*()!!~~"
Analyzer.WriteUserFile fileHandle, writeContent

Analyzer.CloseUserFile(fileHandle)
Set Analyzer = Nothing
WScript.Quit
```

2.17.3 IUsbAnalyzer21::SeekUserFile

```
HRESULT SeekUserFile (
[in] LONGLONG handle,
[in] LONGLONG pos)
```

This function sets the current file position to pos.

Seeking beyond the end of the file: If the position is beyond the end of the file, then SeekUserFile() will not immediately extend the file. If a write is performed at this position, then the file will be extended. The content of the file between the previous end of the file and the newly written data is UNDEFINED and varies between platforms and file systems.

Parameters

handle	This parameter is the unique identifier assigned by the OpenUserFile API. It identifies the file where the data will be written.
pos	This parameter specifies the position in the file to be set as the current position. • If pos = -1, the pos will be set to the end of the file. • The initial position is 0 when the file is opened.

Remarks

Example

```
VBScript:

' Create an instance of the COM object
Dim Analyzer
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")

' Define the file path and mode
Dim filePath, fileMode, fileHandle, writeContent, filePos, readContent, readLen
filePath = "C:\Users\Public\Public Documents\sample_file.txt"
fileMode = &H10 Or &H03 ' text mode 0x10 | ReadWrite 0x03

' Open the file
fileHandle = Analyzer.OpenUserFile(filePath, fileMode)

' Seek the file
filePos = 0
Analyzer.SeekUserFile fileHandle, filePos

' Read from the file
readLen = 100 ' up to 100 chars
Analyzer.ReadUserFile fileHandle, readLen, readContent

msgbox readContent

Analyzer.CloseUserFile(fileHandle)
Set Analyzer = Nothing
WScript.Quit
```

2.17.4 IUsbAnalyzer21::GetUserFileSize

```
HRESULT GetUserFileSize(
[in] LONGLONG handle,
[out, retval] LONGLONG* size)
```

The GetUserFileSize function retrieves the size of the file.

Parameters

handle	This parameter is the unique identifier assigned by the OpenUserFile API. It identifies the file where the data will be written.
size	This is an output parameter that returns the size of the file in bytes. If the file is closed, the size returned will not reflect the actual size of the device.

Remarks

Example

VBScript:

```
' Create an instance of the COM object
Dim Analyzer
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")

' Define the file path and mode
Dim filePath, fileMode, fileHandle, fileSize
filePath = "C:\Users\Public\Public Documents\sample_file.txt"
fileMode = &H01 ' ReadOnly 0x01

' Open the file
fileHandle = Analyzer.OpenUserFile(filePath, fileMode)

' Read the file size
fileSize = Analyzer. GetUserFileSize(fileHandle)

msgbox fileSize

Analyzer.CloseUserFile(fileHandle)
Set Analyzer = Nothing
WScript.Quit
```

2.17.5 IUsbAnalyzer21::WriteUserFile

```
HRESULT WriteUserFile(
[in] LONGLONG handle,
[in] VARIANT content)
```

The WriteUserFile function writes content to the file. The data is flushed to the file immediately after the call.

Parameters

handle	This parameter is the unique identifier assigned by the OpenUserFile API. It identifies the file where the data will be written.
content	This parameter specifies the content to be written to the file. The type of content depends on the mode in which the file was opened. If the file is opened in text mode, the content should be a BSTR. Otherwise, the content should be a SAFEARRAY. • The SAFEARRAY elements type should be VT_UI1.

Remarks

Example

```
VBScript:

' Create an instance of the COM object
Dim Analyzer
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")

' Define the file path and mode
Dim filePath, fileMode, fileHandle, writeContent, filePos, readContent, readLen
filePath = "C:\Users\Public\Public Documents\sample_file.txt"
fileMode = &H10 Or &H03 ' text mode 0x10 | ReadWrite 0x03

' Open the file
fileHandle = Analyzer.OpenUserFile(filePath, fileMode)

' Write to the file
writeContent = "This is a message from a COM client. Unusual chars @#$%^&*()!!~~"
Analyzer.WriteUserFile fileHandle, writeContent

' Seek the file
filePos = 0
Analyzer.SeekUserFile fileHandle, filePos

' Read from the file
readLen = 100 ' up to 100 chars
Analyzer.ReadUserFile fileHandle, readLen, readContent

msgbox readContent

Analyzer.CloseUserFile(fileHandle)
Set Analyzer = Nothing
WScript.Quit
```

2.17.6 IUsbAnalyzer21::ReadUserFile

```
HRESULT ReadUserFile(
[in] LONGLONG handle,
[in] LONGLONG len,
[out] VARIANT* content)
```

The ReadUserFile function reads content from the file for the specified length.

Parameters

handle	This parameter is the unique identifier assigned by the OpenUserFile API. It identifies the file from which data will be read.
len	This parameter specifies the number of bytes to read. Reads up to *len* bytes from the device and returns the read data.
content	This is an output parameter that points to a VARIANT where the read content will be stored. The type of content depends on the mode in which the file was opened: • If the file is opened in text mode, the content will be a BSTR. • Otherwise, the content will be a SAFEARRAY.

Remarks

Example

VBScript:

```
' Create an instance of the COM object
Dim Analyzer
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")

' Define the file path and mode
Dim filePath, fileMode, fileHandle, writeContent, filePos, readContent, readLen
filePath = "C:\Users\Public\Public Documents\sample_file.txt"
fileMode = &H10 Or &H03 ' text mode 0x10 | ReadWrite 0x03

' Open the file
fileHandle = Analyzer.OpenUserFile(filePath, fileMode)

' Write to the file
writeContent = "This is a message from a COM client. Unusual chars @#$%^&*()!!~~"
Analyzer.WriteUserFile fileHandle, writeContent

' Seek the file
filePos = 0
Analyzer.SeekUserFile fileHandle, filePos

' Read from the file
readLen = 100 ' up to 100 chars
Analyzer.ReadUserFile fileHandle, readLen, readContent

msgbox readContent

Analyzer.CloseUserFile(fileHandle)
Set Analyzer = Nothing
WScript.Quit
```

3 Primary Dual Interface for Trace

3.1 IUsbTrace Dual Interface

IUsbTrace interface is the primary interface for the **UsbTrace** object. It derives from the **ITrace** interface that implements the common functionality for all Teledyne LeCroy Analyzers.

3.1.1 ITrace::GetName

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

Retrieves the trace name.

Parameters

`trace_name` 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.USBTracer")
        CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
        Set Trace = Analyzer.MakeRecording (CurrentDir & "Input\test_ro.rec")
        MsgBox "Trace name " & Trace.GetName

C++:
        IUsbTrace* usb_trace;

        . . .

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

3.1.2 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 display options file

Return values

`ANALYZERCOMERROR_UNABLELOADDO` Unable to load 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 USB Protocol Suite application. You can create such a file by selecting **Setup – Display Options…** from the USB Protocol Suite application menu, changing the display options in the dialog, and selecting the **Save** button.

Example

```
WSH:
        Set Analyzer = WScript.CreateObject("CATC.USBTracer")
        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.usb"

C++:

        IUsbTrace* usb_trace;
        TCHAR file_name[_MAX_PATH];

        . . .

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

3.1.3 ITrace::Save

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

Saves the trace into a file and allows you to save a range of packets.

Parameters

`file_name`	String providing the full pathname to file where trace is saved
`packet_from`	Beginning packet number when you are saving a range of packets. Value –1 means that the first packet of saved trace would be 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 saved trace would be the last packet of this trace.

Return values

`ANALYZERCOMERROR_UNABLESAVE`	Unable to save trace file.

Remarks

Use this method if you want to save a recorded or opened trace into a file. If display options apply to this trace (see ITrace::ApplyDisplayOptions or IAnalyzer::LoadDisplayOptions), then hidden packets are not saved.

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

Example

```
WSH:
        Set Analyzer = WScript.CreateObject("CATC.USBTracer")
        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.usb"

C++:
        IUsbTrace* usb_trace;
        TCHAR file_name[_MAX_PATH];
        LONG packet_from;
        LONG packet_to;
        . . .
        try
        {
                usb_trace->Save( file_name, packet_from, packet_to );
        }
        catch (_com_error& er)
        {
                if (er.Description().length() > 0)
                ::MessageBox( NULL, er.Description(),  _T("UsbAnalyzer client"), MB_OK );
                else
                ::MessageBox( NULL, er.ErrorMessage(), _T("UsbAnalyzer client"), MB_OK );
                return 1;
        }
```

3.1.4 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 and allows you to export a range of packets.

Parameters

`file_name` String providing the full pathname to file where trace is exported

`packet_from` Beginning packet number when you are exporting a range of packets. Value –1 means that the first packet of exported trace would be 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 exported trace would be 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 opened trace into a text file. If display options apply to this trace (see ITrace::ApplyDisplayOptions or IAnalyzer::LoadDisplayOptions), then hidden packets are not exported.

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

Here is a snippet of an export file for a Bluetooth Trace. A USB Trace file would look similar:

```
File E:\AnalyzerSw\Chief20\OfficialUTGFiles\SRP.usb.
From Packet #13 to Packet #24.

Packet#
_______|_____________________________________________________________________
13_____| Dir(-->) Suspend( 37.000 ms) Time-stamp(00016.0097 3086)
_______|_____________________________________________________________________
14_____| Dir(-->) Reset(  3.017 µs) Time-stamp(00016.0393 3090)
_______|_____________________________________________________________________
15_____| Dir(-->) F(S) Sync(00000001) SOF(0xA5) Frame #(6) CRC5(0x09)
_______| EOP(266 ns) Time-stamp(00016.0393 3336)
_______|_____________________________________________________________________
16_____| Dir(-->) F(S) Sync(00000001) SOF(0xA5) Frame #(7) CRC5(0x16)
_______| EOP(266 ns) Time-stamp(00016.0401 3336)
_______|_____________________________________________________________________
17_____| Dir(-->) F(S) Sync(00000001) SOF(0xA5) Frame #(8) CRC5(0x06)
_______| EOP(266 ns) Time-stamp(00016.0409 3336)
_______|_____________________________________________________________________
18_____| Dir(-->) F(S) Sync(00000001) SOF(0xA5) Frame #(9) CRC5(0x19)
_______| EOP(266 ns) Time-stamp(00016.0417 3336)
_______|_____________________________________________________________________
19_____| Dir(-->) F(S) Sync(00000001) OUT(0x87) ADDR(2) ENDP(3) CRC5(0x0C)
_______| EOP(266 ns) Time-stamp(00016.0417 3536)
_______|_____________________________________________________________________
20_____| Dir(-->) F(S) Sync(00000001) DATA0(0xC3)-BAD Data(8 bytes)
_______| CRC16(0xBB29) EOP(266 ns) Time-stamp(00016.0417 3726)
_______|_____________________________________________________________________
21_____| Dir(<--) F(S) Sync(00000001) ACK(0x4B) EOP(266 ns)
_______| Time-stamp(00016.0417 4256)
_______|_____________________________________________________________________
22_____| Dir(-->) F(S) Sync(00000001) SOF(0xA5) Frame #(10) CRC5(0x1B)
_______| EOP(266 ns) Time-stamp(00016.0425 3336)
_______|_____________________________________________________________________
23_____| Dir(-->) F(S) Sync(00000001) SOF(0xA5) Frame #(11) CRC5(0x04)
_______| EOP(266 ns) Time-stamp(00016.0433 3336)
_______|_____________________________________________________________________
24_____| Dir(-->) Suspend(0 ns) Time-stamp(00016.0457 3466)
_______|_____________________________________________________________________
```

Example

```
WSH:
        Set Analyzer = WScript.CreateObject("CATC.USBTracer")
        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++:

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

3.1.5 ITrace::Close

```
HRESULT Close ( );
```

Closes the trace.

Parameters

Return values

Remarks

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

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

Example

3.1.6 ITrace::ReportFileInfo

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

Saves trace information into an HTML file.

Parameters

`file_name` String with full pathname to file where trace information report will be created.

Return values

`ANALYZERCOMERROR_UNABLESAVE` Unable to create trace information report.

Remarks

Stores trace information in specified file. If the file specified in the *file_name* parameter does not exist, one will be created.

Example

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

C++:
        IUsbTrace* usb_trace;
        TCHAR file_name[_MAX_PATH];

        . . .

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

3.1.7 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 file where error summary report is created

Return values

`ANALYZERCOMERROR_UNABLESAVE` Unable to create trace information report.

Remarks

Creates error summary file if necessary. Stores error summary in specified file. Here is an example of data stored using this method call:

```
Error report for SRP.usb recording file.

_______|_____________________________________________________________________
Bad PID (0):
_______|_____________________________________________________________________
Bad CRC5 (0):
_______|_____________________________________________________________________
Bad CRC16 (0):
_______|_____________________________________________________________________
Bad Packet Length (0):
_______|_____________________________________________________________________
Bad Stuff Bits (0):
_______|_____________________________________________________________________
Bad EOP (0):
_______|_____________________________________________________________________
Babble Start (0):
_______|_____________________________________________________________________
Babble End (LOA) (0):
_______|_____________________________________________________________________
Bad Frame Length (0):
_______|_____________________________________________________________________
Bad Turnaround/Timeout (0):
_______|_____________________________________________________________________
Bad Data Toggle (1):
_______| 0.20;
_______|_____________________________________________________________________
Bad Frame/uFrame Number (0):
_______|_____________________________________________________________________
Analyzer Internal Error (0):
_______|_____________________________________________________________________
Last Byte Incomplete (0):
_______|_____________________________________________________________________
```

Example

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

C++:
        IUsbTrace* usb_trace;
        TCHAR file_name[_MAX_PATH];

        . . .

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

3.1.8 ITrace::ReportTrafficSummary

```
HRESULT ReportTrafficSummary(
      [in] BSTR file_name );
```

Saves trace traffic summary information into the specified text file.

Parameters

file_name	String providing the full pathname to file where traffic summary report is created

Return values

ANALYZERCOMERROR_UNABLESAVE	Unable to create or save traffic summary report.

Remarks

Creates traffic summary report file, if necessary. Stores traffic summary in the specified file. Here is an example of data stored using this method call:

Example

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

C++:
       IUsbTrace* usb_trace;
       TCHAR file_name[_MAX_PATH];

       . . .

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

3.1.9 ITrace::GetPacket

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

Retrieves the raw packet representation.

Parameters

`packet_number`	Number of packet to retrieve
`packet`	Raw packet representation
`number_of_bits`	Number of bits in raw packet representation

Return values

`ANALYZERCOMERROR_INVALIDPACKETNUMBER`	Specified packet number is invalid

Remarks

The **packet** parameter has `VT_ARRAY | VT_VARIANT` actual automation type. Each element of this array has `VT_UI1` automation type. Since the last element of the array may contain extra data, you need to use the **number_of_bits** parameter to determine the actual packet data.

Example

```
VBScript:
	<OBJECT
		ID = Analyzer
		CLASSID = "clsid:0B179BB3-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
		On Error Resume Next
		Dim Packet
		NumberOfBits = (CLng(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, NumberOfBits )
			StatusText.innerText = "Packet ( " & NumberOfBits & " bits ): " &
			PacketStr
		End If
	End Sub
	-->
	</SCRIPT>
```

C++:

```
        IUsbTrace* usb_trace;
        LONG packet_number;

        . . .

        VARIANT packet;
        VariantInit( &packet );
        long number_of_bits;
        try
        {
            number_of_bits = usb_trace->GetPacket( packet_number, &packet );
        }
        catch ( _com_error& er)
        {
            if (er.Description().length() > 0)
               ::MessageBox( NULL, er.Description(), _T("UsbAnalyzer client"), MB_OK );
            else
               ::MessageBox( NULL, er.ErrorMessage(),_T("UsbAnalyzer 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("UsbAnalyzer client"), MB_OK );
                    return 1;
                }
                if ( var.vt != ( VT_UI1) )
                {
    ::MessageBox( NULL, _T("Array of bytes expected"), _T("UsbAnalyzer client"), MB_OK );
                    return 1;
                }

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

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

3.1.10 ITrace::GetPacketsCount

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

Retrieves total number of packets in the trace.

Parameters

number_of_packets	Points to long value where number of packets in the trace is retrieved.

Return values

Remarks

Example

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

C++:
       IUsbTrace* usb_trace;

       . . .

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

3.1.11 ITrace::GetTriggerPacketNum

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

Retrieves trigger packet number.

Parameters

`packet_number` Points to long value where trigger packet number is retrieved.

Return values

Remarks

Example

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

C++:
        See an example for ITrace::GetPacketsCount.
```

3.1.12 ITrace::AnalyzerErrors

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

Retrieves trace file errors.

Parameters

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

`0x00000001`	- Pid Error
`0x00000002`	- CRC5 Error
`0x00000004`	- CRC16 Error
`0x00000008`	- Packet Length Error
`0x00000010`	- Stuff Bit Error
`0x00000020`	- EOP Error
`0x00000040`	- Babble Start Error
`0x00000080`	- Babble End Error (LOA)
`0x00000100`	- Frame Length Error
`0x00000200`	- Handshake Timeout Error
`0x00000400`	- Analyzer Internal Error
`0x00000800`	- Data Toggle Error
`0x00001000`	- Microframe Error
`0x00002000`	- Short Byte Error (bits missing)

`analyzer_errors` Address of a pointer to the AnalyzerErrors object primary interface

Return values

`ANALYZERCOMERROR_INVALIDERROR`	Invalid error type specified

Remarks

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

3.2 IUsbTrace2 interface

The **IUsbTrace2** interface is the second interface for the **UsbTrace** object. It inherits and extends some trace-related functionality exposed via the **IUsbTrace** interface.

IID : 149B95E9-C17D-43b3-AC0D-30419A485058

3.2.1 IUsbTrace2::GotoTime

```
HRESULT GotoTime ( [in] double time );
```

Instructs the trace view to jump to the specified timestamp.

Parameters

`time` Time (in nanoseconds) to jump

Example

WSH:

```
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))

Analyzer.StartRecording (CurrentDir & "test_ro.rec")
'... Do something.
Set Trace = Analyzer.StopRecordingAndWaitForTrace

Trace.GotoTime( 1000000 ) ' Go to 1 millisecond timestamp.
```

C++:

```
IUsbTrace2* Usb_trace;

. . .

try
{
        double t = 1000000.0;
        Usb_trace ->GotoTime( t );
}
catch (_com_error& er)
{
        if (er.Description().length() > 0)
        ::MessageBox( NULL, er.Description(),  _T("UsbAnalyzer client"), MB_OK );
        else
        ::MessageBox( NULL, er.ErrorMessage(), _T("UsbAnalyzer client"), MB_OK );
        return 1;
}
```

3.2.2 IUsbTrace2::GotoUnit

```
HRESULT GotoUnit ([in] long level, [in] long index );
```

Instructs the trace view to jump to the specified transaction unit.

Parameters

`level` Transaction level
Possible values are:
0 = USB Packets
1 = USB Transactions
2 = USB Split Transactions
3 = USB Transfers
4 = Host WireAdapter Segments
5 = Host WireAdapter Transfers
6 = Device WireAdapter Segments
7 = Device WireAdapter Transfers
8 = PTP Transactions
9 = PTP Objects
10 = PTP Sessions

`index` Index of unit inside transaction level

Return values

`ANALYZERCOMERROR_INVALIDPACKETNUMBER`	Unable to jump to specified transaction unit: Either transaction level or transaction index is invalid.

Example

```
WSH:
	Set Analyzer = WScript.CreateObject("CATC.UsbTracer")
	CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
	Analyzer.StartRecording (CurrentDir & "test_ro.rec")
	'... do something.
	Set Trace = Analyzer.StopRecordingAndWaitForTrace

	Trace.GotoUnit( 3, 2 ) ' jumps to USB transfer #2


C++:
	IUsbTrace3* Usb_trace;
	. . .
	try
	{
		int level = 3; // USB Transfers
		int index = 2;
		Usb_trace ->GotoUnit( 3, 2 );
	}
	catch (_com_error& er)
	{
		if (er.Description().length() > 0)
		::MessageBox( NULL, er.Description(),  _T("UsbAnalyzer client"), MB_OK );
		else
		::MessageBox( NULL, er.ErrorMessage(), _T("UsbAnalyzer client"), MB_OK );
		return 1;
	}
```

3.2.3 IUsbTrace2::ExportToCsv

```
HRESULT ExportToCsv(
      [in] BSTR file_name,
      [in] long level,
      [in, defaultvalue(-1)] long unit_from,
      [in, defaultvalue(-1)] long unit_to );
```

Exports a trace into a text file in CSV format and allows exporting a range of packets.

Parameters

`file_name`	String providing the full pathname of the trace export file
`level`	Transaction level 0 – Packets 1 – USB Transactions 2 – USB Split Level 3 – USB Transfers 8 – PTP Transactions 9 – PTP Object 10 – PTP Session 11 – SCSI Operation 13 – PTP Group 14 - OBEX (Currently only these are supported.)
`unit_from`	Beginning packet number when you are exporting a range of packets. Value –1 makes the first packet of the exported trace be the first packet of the trace.
`unit_to`	Ending packet number when you are exporting a range of packets. Value –1 makes the last packet of the exported trace be the last packet of the trace.

Return values

`ANALYZERCOMERROR_UNABLESAVE`	Unable to export trace file

Remarks

Use this method if you want to export a recorded or opened trace into a text file in CSV format. If display options apply to this trace (see `ITrace::ApplyDisplayOptions` or `IAnalyzer::LoadDisplayOptions`), then hidden units would not be exported.

If a unit range is specified and it is invalid (for example, **unit_to** is more than the last unit number in the trace, or **unit_from** is less than first unit number in the trace, or **unit_from** is more then **unit_to**) then unit range adjusts automatically.

Example

WSH:

```
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))

' Please specify the real path to the trace file instead of 'C:\test.usb'.
Set Trace = Analyzer.OpenFile ( "C:\test.usb" )

' Instructs the USB Protocol Suite application to locate packet # 15
' (transaction level 0) in the trace viewer.
Trace.GotoUnit 0, 15

' Instructs the USB Protocol Suite trace viewer to jump to the trace unit
' with the timestamp close to 6000 ns (6 microsecond).
Trace.GotoTime 6000

' Exports USB transactions (from #0 to #10 ) into CSV format.
' 1st parameter - file name of the export file
' 2nd parameter - transaction level ( 0 - packets, 1 - transactions )
' 3rd parameter - unit number to start with
' 4th parameter - unit number to end by
Trace.ExportToCsv CurrentDir & "Output\text_csv_export.csv", 1, 0, 10

MsgBox "Done"
```

C++:

```
IUsbTrace2* usb_trace;

. . .

IAnalyzerErrors* analyser_errors;
try
{
        analyser_errors = usb_trace-> ExportToCsv (_bstr_t( "c:\\test.csv" ),
                                                    1, 0 , 10 ) ;
}
catch (_com_error& er)
{
        if (er.Description().length() > 0)
        ::MessageBox( NULL, er.Description(), _T("UsbAnalyzer client"), MB_OK );
        else
        ::MessageBox( NULL, er.ErrorMessage(),_T("UsbAnalyzer client"), MB_OK );
        return 1;
}

. . .

analyser_errors->Release();
```

3.2.4 IUsbTrace2::SaveAs

```
HRESULT SaveAs ( [in] BSTR file_name );
```

Saves trace into a file and makes current IUsbTrace2 pointer refer to this file.

Parameters

`file_name` String providing the full pathname to file in which trace is saved

Return values

`ANALYZERCOMERROR_UNABLESAVE` Unable to save trace file

Remarks

Use this method if you want to move a recorded or opened trace into a file.

Example

WSH:

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

'... Do something.

Set Trace = Analyzer.StopRecordingAndWaitForTrace
Trace.SaveAs CurrentDir & "Output\saved_file.usb"
```

C++:

```
IUsbTrace2* USB_trace;
TCHAR file_name[_MAX_PATH];
LONG packet_from;
LONG packet_to;

...

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

3.3 IUsbTrace3 interface

The **IUsbTrace3** interface is the third interface for the **UsbTrace** object. It inherits and extends some trace-related functionality exposed via the **IUsbTrace3** interface.

IID : 4DA2A987-47B3-4384-BFA8-2052326FBE0E

3.3.1 IUsbTrace3:: GetPowerInfoByTime

```
HRESULT GetPowerInfoByTime (
     [in] double time,
     [out] VARIANT* voltage,
     [out] VARIANT* current,
     [out] VARIANT* power,
     [out,retval] BOOL* power_info_is_valid );
```

Returns voltage, current, and power values based on a time in nanoseconds.

Parameters

`time`	timestamp in nanoseconds
`voltage`	voltage in microvolts
`current`	current in microamps
`power`	power in microwatts.

Return values

`BOOL`	TRUE if power values were recorded at the time.

Remarks

3.3.2 IUsbTrace3:: GetPowerInfoByPacket

```
HRESULT GetPowerInfoByPacket (
     [in] long packet_number,
     [out] VARIANT* voltage,
     [out] VARIANT* current,
     [out] VARIANT* power,
     [out,retval] BOOL* power_info_is_valid );
```

Returns voltage, current, and power values based on a packet number..

Parameters

`packet_number`	Packet number in the trace file

`voltage`	voltage in microvolts
`current`	current in microamps
`power`	power in microwatts.

Return values

`BOOL`	TRUE if power values were recorded at that packet_number.

Remarks

3.4 IUsbTrace4 interface

The **IUsbTrace4** interface is the forth interface for the **UsbTrace** object. It inherits and extends some trace-related functionality exposed via the **IUsbTrace3** interface.

IID : C7D741F6-D425-417f-B7FC-4DB0FB01804B

3.4.1 IUsbTrace4:: GetFullName

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

Returns name of trace file. This means Filename.extension, as in **MyTrace.usb**. It does NOT include the path.

Parameters

trace_name	Pointer to string to retrieve trace file full filename

Return values

Remarks

3.5 IUsbTrace5 interface

The **IUsbTrace5** interface is the fifth interface for the **UsbTrace** object. It inherits and extends some trace-related functionality exposed via the **IUsbTrace4** interface.

IID : CAC0F43F-322E-4165-9BB6-E0DBDD95B682

3.5.1 IUsbTrace5::GetUsbPacket

```
HRESULT GetUsbPacket(
        [in] long packet_number,
        [out, retval] IDispatch** usb_packet )
```

Retrieves a pointer to the specified packet.

Parameters

Packet_number	Number of packet to retrieve.
Usb_packet	Address of a pointer to the UsbPacket object primary interface.

Return values

Remarks

3.6 IusbTrace6 interface

The **IUsbTrace6** interface is the sixth interface for the **UsbTrace** object. It inherits and extends some trace-related functionality exposed via the **IUsbTrace5** interface.

IID : A1F83FCC-265A-44E3-837F-42A8C27074C1

3.6.1 IUsbTrace6::IsLevelDecodeFinished

```
HRESULT IsLevelDecodeFinished(
        [in] long level_id,
        [out,retval] BOOL* decode_is_finished )
```

Retrieves progress status of decoding levels process. If level decoding is finished, returns true.

Parameters

`level_id` Desired level to retrieve decoding status like Transaction/Transfer.
A value of -1 refers to all level, in which case the status would be true if all level decodings are finished.

`decode_is_finished` Pointer to Boolean value that specifies decoding status

Return values

Remarks

After opening traces, decoding process of upper levels starts in background processes or through a function call for higher levels. This decoding is time consuming process, however any other process on desired level like transactions need to check of decoding finish before starting process.
The level ID parameter could be one of these values:

```
DECODE_LEVEL_PACKET             = 0,  // USB_TRA_LEVEL_PACKETS
DECODE_LEVEL_TRANSACTION        = 1,  // USB_TRA_LEVEL_TRANSACTIONS
DECODE_LEVEL_SPLIT              = 2,  // USB_TRA_LEVEL_SPLITS
DECODE_LEVEL_TRANSFER           = 3,  // USB_TRA_LEVEL_TRANSFERS
DECODE_LEVEL_PTP_TRANSACTION    = 8,  // USB_TRA_LEVEL_PTP_TRANSACTION
DECODE_LEVEL_PTP_OBJECT         = 9,  // USB_TRA_LEVEL_PTP_OBJECT
DECODE_LEVEL__PTP_SESSION       = 10, // USB_TRA_LEVEL_PTP_SESSION
DECODE_LEVEL_SCSI               = 11, // USB_TRA_LEVEL_SCSI_OPERATION
DECODE_LEVEL_PTP_GRP            = 13, // USB_TRA_LEVEL_PTP_GROUP
```

3.6.2 IusbTrace6::SetTag

```
HRESULT SetTag( [in] int tag)
```

Specify a optional unique id for trace. This id could be used as identifier of trace in trace related events when user works with mutiple traces at same time. See _ITraceEvent interface for furthuer detail.

Parameters

`tag` optional unique identifier of trace.

Return values

Remarks

3.6.3 IusbTrace6::GetTag

```
HRESULT GetTag( [out, retval] int* tag)
```

Retrieves the unique identifer of trace which previously specified by SetTag method.

Parameters

`tag` optional unique identifier of trace.

Return values

Remarks

3.7 IUsbTrace7 Interface

The IUsbTrace7 interface is an additional interface for the UsbTrace object. It inherits and extends some trace-related functionality exposed via the IUsbTrace6 interface.

IID: F3260A83-8F86-4356-81DC-D9B880E1293E

3.7.1 IUsbTrace7::GetPacketEx

```
HRESULT GetPacketEx ( [in] hyper packet_number, [in, out] VARIANT* packet,
[out, retval] long* number_of_bits )
```

Retrieves a raw packet in the PACKETFORMAT_BYTES format. It accepts 64 bits packet number. It is ITrace::GetPacket method extension.

Parameters

`packet_number`	Zero based number of packet to retrieve. It is a 64 bits integer.
`packet`	Raw packet representation
`number_of_bits`	Number of bits 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 elements of this array has the VT_UI1 automation type.

3.7.2 IUsbTrace7:: GetPacketsCountEx

```
HRESULT GetPacketsCountEx ( [out, retval] hyper* number_of_packets )
```

Retrieves the total number of packets in the trace. Number of packets can be a 64 bits integer. It is ITrace::GetPacketsCount method extension.

Parameters

`number_of_packets`	Total number of packets in the trace

Return values

Remarks

3.7.3 IUsbTrace7:: GetTriggerPacketNumEx

```
HRESULT GetTriggerPacketNumEx ( [out, retval] hyper* packet_number )
```

Retrieves the trigger packet number. The trigger packet number can be a 64 bits integer. It is ITrace::GetTriggerPacketNum method extension.

Parameters

`packet_number` Zero based number of the packet where trigger occurred

Return values

Remarks

3.7.4 IUsbTrace7:: GetUsbPacketEx

```
HRESULT GetUsbPacketEx ( [in] hyper packet_number, [out, retval] IDispatch**
packet )
```

Retrieves the interface for packet within a trace. It accepts 64 bits packet number. It is IUsbTrace5::GetUsbPacket method extension.

Parameters

`packet_number` Zero based number of packet to retrieve. It is a 64 bits integer.

`packet` Address of a pointer to the UsbPacket object interface.

Return values

Remarks

3.8 IUsbTrace8 Interface

The IUsbTrace8 interface is an additional interface for the UsbTrace object. It inherits and extends some trace-related functionality exposed via the IUsbTrace6 interface.

IID: 5B40CDDA-6FEE-4B81-9307-3D1512D70947

3.8.1 IUsbTrace8::GetFullPathAndName

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

Returns name of trace path and filename with extension. Example: **C:\Stuff\MyTrace.usb**

If you only want the filename, use IUsbTrace4::GetFullName().

Parameters

`trace_name`	Pointer to string to retrieve trace file full path and filename

Return values

Remarks

3.9 IUsbTrace10 interface

The IUsbTrace10 interface is an additional interface for the UsbTrace object. It inherits and extends some trace-related functionality exposed via the IUsbTrace9 interface.

IID : AC4DC05E-3B20-4233-BDE6-F92B3A185B28

3.9.1 IUsbTrace10::ExportTransactionsWithPacketsToCsv

```
HRESULT ExportTransactionsWithPacketsToCsv (
      [in] BSTR file_name,
      [in, defaultvalue(-1)] long unit_from,
      [in, defaultvalue(-1)] long unit_to );
```

Exports the USB Transactions and packets level of the trace into a text file in CSV format and allows exporting a range of packets.

Parameters

`file_name`	String providing the full pathname of the trace export file
`unit_from`	Beginning packet number when you are exporting a range of packets. Value –1 makes the first packet of the exported trace be the first packet of the trace.
`unit_to`	Ending packet number when you are exporting a range of packets. Value –1 makes the last packet of the exported trace be the last packet of the trace.

Return values

`ANALYZERCOMERROR_UNABLESAVE`	Unable to export trace file

Remarks

Use this method if you want to export a recorded or opened trace into a text file in CSV format. If display options apply to this trace (see `ITrace::ApplyDisplayOptions` or `IAnalyzer::LoadDisplayOptions`), then hidden units would not be exported.

If a unit range is specified and it is invalid (for example, **unit_to** is more than the last unit number in the trace, or **unit_from** is less than first unit number in the trace, or **unit_from** is more then **unit_to**) then unit range adjusts automatically.

Example

WSH:

```
Set Analyzer = WScript.CreateObject("CATC.UsbTracer")
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))

' Please specify the real path to the trace file instead of 'C:\test.usb'.
Set Trace = Analyzer.OpenFile ( "C:\test.usb" )

' Instructs the USB Protocol Suite application to locate packet # 15
' (transaction level 0) in the trace viewer.
Trace.GotoUnit 0, 15

' Instructs the USB Protocol Suite trace viewer to jump to the trace unit
' with the timestamp close to 6000 ns (6 microsecond).
Trace.GotoTime 6000

' Exports USB transactions with packets (from #0 to #10 ) into CSV format.
' 1st parameter - file name of the export file
' 2rd parameter - unit number to start with
' 3th parameter - unit number to end by
Trace.ExportTransactionsWithPacketsToCsv CurrentDir &"Output\text_csv_export.csv", 0, 10

MsgBox "Done"
```

C++:

```
IUsbTrace2* usb_trace;

. . .

IAnalyzerErrors* analyser_errors;
try
{
       analyser_errors = usb_trace-> ExportTransactionsWithPacketsToCsv (_bstr_t(
"c:\\test.csv" ), 0 , 10 );
}
catch (_com_error& er)
{
       if (er.Description().length() > 0)
       ::MessageBox( NULL, er.Description(), _T("UsbAnalyzer client"), MB_OK );
       else
       ::MessageBox( NULL, er.ErrorMessage(),_T("UsbAnalyzer client"), MB_OK );
       return 1;
}

. . .

analyser_errors->Release();
```

3.10 IUsbTrace11 interface

The IUsbTrace11 interface is an additional interface for the UsbTrace object. It inherits and extends some trace-related functionality exposed via the IUsbTrace10 interface.

IID : F1ED8DB5-2EC6-4fa0-8F8B-D645A1FA4A85

3.10.1 IusbTrace11::SaveTraceAs

```
      HRESULT SaveTraceAs ([ in ] BSTR file_name, [ in, defaultvalue( -1 ) ]
long packet_from, [ in, defaultvalue( -1 ) ] long packet_to, [in,
defaultvalue( 0 )] BOOL exclude_hidden_packets );
```

Saves specified range of packets into a new Trace file.

Parameters

file_name	String providing the full path name to file in which trace is saved
packet_from	The index of first packet as start point for save as
packet_to	The index of last packet as end point for save as
exclude_hidden_packets	A Boolean to include/exclude hidden packets in save as

Return values

ANALYZERCOMERROR_UNABLESAVE	Unable to save trace file

Remarks

Provide to save specific range of packets in a new trace file.

Example

WSH:

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

'... Do something.

Set Trace = Analyzer.StopRecordingAndWaitForTrace
Trace.SaveTraceAs CurrentDir & "Output\saved_file.usb", 0, 100, 1
```

C++:

```
IUsbTrace11* USB_trace;
TCHAR file_name[_MAX_PATH];
LONG packet_from = 0;
LONG packet_to = 100;
BOOL exclude_hidden_packets = 1;
...

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

3.10.2 IusbTrace11::ExportUSBTransferData

```
HRESULT ExportUSBTransferData( [in] long addr, [in] long endp, [in]
long direction, [in] long level_from, [in] long unit_from, [in] long
level_to, [in] long unit_to, [in] BSTR file_name, [in] BOOL file_type_binary,
[in] long size_kb );
```

Exports USB Transfers data into a text or binary file.

Parameters

addr	Specifies Device Address
endp	Specifies Endpoint Number
direction	The direction of endpoint (1= Either, 2 = Out, and 3 = IN)
level_from	levels (0 = Packet, 1 = Transaction, 3 = Transfer)
unit_from	The index of first packet as start point for export
level_to	levels (0 = Packet, 1 = Transaction, 3 = Transfer)
unit_to	The index of last packet as end point for export
filename	String providing the full path name of output file
file_type_binary	The output file format (0 = Text, 1 = Binary)
size_kb	Maximum file size in case of using binary as output format

Return values

ANALYZERCOMERROR_UNABLESAVE	Unable to save trace file
ANALYZERCOMERROR_INVALIDTRACEHANDLE	Returned value depends on the step of process

Example

WSH:

```
Const DATDIR_NONE   = 0
Const DATDIR_EITHER = 1
Const DATDIR_OUT    = 2
Const DATDIR_IN     = 3

Const USB_TRA_LEVEL_PACKETS      = 0
Const USB_TRA_LEVEL_TRANSACTIONS = 1
Const USB_TRA_LEVEL_TRANSFERS    = 3

Dim address          : address           = 2
Dim endpoint         : endpoint          = 1
Dim direction        : direction         = DATDIR_OUT
Dim packet_level_from: packet_level_from = USB_TRA_LEVEL_TRANSFERS
Dim from_packet      : from_packet       = 0
Dim packet_level_to  : packet_level_to   = USB_TRA_LEVEL_TRANSFERS
Dim to_packet        : to_packet         = 200
Dim file_path        : file_path         = D:\data.txt"
Dim file_type        : file_type         = 0
Dim size_kb          : size_kb           = 0

Set Analyzer = WScript.CreateObject("CATC.UsbTracer")
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Analyzer.StartRecording (CurrentDir & "Input\test_ro.rec")

'... Do something.

Set Trace = Analyzer.StopRecordingAndWaitForTrace
Trace.ExportUSBTransferData ddress, endpoint, direction, packet_level_from,from_packet,
packet_level_to, to_packet, file_path, file_type, size_kb
```

3.11 IUsbTrace12 interface

3.11.1 IUsbTrace12::ExportUSB4TunneledProtocolTraffic

```
HRESULT ExportUSB4TunneledProtocolTraffic
                ( [in] BSTR output_file_path,
                  [in] EUSB4TunneledProtocol protocol)
```

Exports tunneled USB4 traffic of the specified protocol to file format of native application that corresponds the specified protocol.

Parameters

output_file_path	Specifies path (with file name) of the output file
protocol	Specifies tunneled protocol
- TUN_PCIE	Tunneled PCIe
- TUN_DP	Tunneled Display Port Main Link
- TUN_DP_AUX	Tunneled Display Port AUX
- TUN_USB3	Tunneled USB3

3.12 IUsbTrace13 interface

3.12.1 IUsbTrace13::ExportVisibleTraList

```
HRESULT ExportVisibleTraList ( [in] BSTR file_path )
```

Exports visible transactions to JSON format.

Parameters

file_path Specifies path (with file name) of the output JSON file

3.13 IUsbTrace14 interface

3.13.1 IUsbTrace14:: LoadDecodingSettings

```
HRESULT LoadDecodingSettings([in] BSTR file_path)
```

Load the DecodingSettings from the saved xml file.

Parameters

file_path Specifies a saved xml path (with file name) of the Decoding Settings.

Remarks

- The saved decoding settings file should be corelated with the trace. For example, one (or more) entries of the HopId in the xml file, should match the hopId(s) of the trace file, in order to get overridden.

- Level decoding must be finished before calling this api. (For more infromation please refer to **IUsbTrace6::IsLevelDecodeFinished** section)
- Currently this is supported for USB4 trace only.

Exmaple

WSH:

```
TraceName   = "D:\Some trace files\CapturedTrace.usb"
xmlPath     = "D:\Some trace files\DecSettings.xml"
ExportPath  = "D:\Some trace files\CapturedTrace_Exported.usb"

Set Analyzer = WScript.CreateObject("CATC.UsbTracer")
Set Trace    = Analyzer.OpenFile( TraceName )
do while True
    if  Trace.IsLevelDecodeFinished(-1) Then exit do
loop
Trace.LoadDecodingSettings(xmlPath)
Trace.ExportUSB4TunneledProtocolTraffic(ExportPath, 3) 'TUN_USB3
```

3.14 _ITraceEvents interface

The **_IUsbEvents** interface provides an event interface over trace files objects. This interface is used to through some events from trace analysis to client application. Be noted that this interface is not independent object and sink object in client code [who implements the events methods] should bind to existing trace object to notified by events from trace.

IID : 81C4C56A-AB57-4402-AD11-7D49F7CF450E

3.14.1 ITraceEvents::OnLevelDecodeFinished

```
HRESULT OnLevelDecodeFinished
        ( [in] int levelId,
          [in] int tag )
```

This event notifies client when decoding of any levels in applications like transaction/split/transfer… is finished. Client code may start some other process on this level data after getting finish decoding event.

Parameters

level_id	Specifies level that decoding process is finished
tag	Specified the unique of trace that previously set by SetTag

Return values

Remarks

For detail infromation about level IDs please refer to **IUsbTrace5::IsLevelDecodeFinished** section.

3.15 IUsbVerificationScript Interface

The **IUsbVerificationScript** interface is the seventh interface for the **UsbTrace** object. It exposes the trace functionality for running verification scripts. It derives from the **IVerificationScript** interface, which implements the common functionality for running Teledyne LeCroy PSG verification scripts over recorded traces. This interface is not dual, so scripting languages cannot use it directly, though some or all of its methods are exposed to script languages through the primary **UsbTrace** automation interface.

IID : F1ED8DB5-2EC6-4fa0-8F8B-D645A1FA4A85

Remarks

Verification scripts are special scripts written using Teledyne LeCroy PSG Script Language (CSL), which can be “run” over the 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 USB Protocol Suite application called the “Verification Script Engine”. This feature must be enabled for you to run verification scripts.

(For more information, refer to the *USB Protocol Suite User Manual* and the *USB Protocol Suite VSEngine Manual*.)

3.15.1 IUsbVerificationScript::RunVerificationScript

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

Runs verification script over the recorded trace.

Parameters

`verification_script`	Name or full path of the verification script to run
`pResult`	Address of a variable in which to keep the result of verification

VS_RESULT is an enumeration type that can have these possible meanings:

```
enum VS_RESULT
{
    UNKNOWN          = -5,
    LICENSE_FAIL     = -4, // License for decoding traffic is unavailable.
    PARSE_ERROR      = -3, // VSE parser encountered an error.
    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.
    PASSED_WARNING   =  3, // Verification passed, but a warning should be issued.
    NOT_APPLICABLE   =  4, // Verification was not applicable.
    PASSED_WITH_NOTE =  5  // Verification passed, and a note should be displayed.
};
```

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 (***.vse**) without the file extension. For example, for verification script file **test.vse**, the test name is **test**. Please refer to the *Teledyne LeCroy UWB Script Verification Engine Manual* for details.

If a full path is specified (such as **C:\test.vse**), VSE uses it, instead of searching for the script by name in the **\Scripts\VFScripts** application subfolder.

Example

C++:

```
        // In this example, use wrapper functions provided by #import directive.

       IUsbTrace* trace;
       . . .

       IUsbVerificationScript* vscript = NULL;

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

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

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

       . . .
```

WSH:

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

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

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

       MsgBox("Done")
```

3.15.2 IUsbVerificationScript::GetVScriptEngine

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

Retrieves a verification script engine object.

Parameters

`script_name`	String that providing the full pathname of the verification script to initialize the script verification engine
`pVSEngine`	Address of a pointer to the **`vScriptEngine`** object primary interface

Return values

S_OK	If verification script engine object was successfully retrieved.

Remarks

The name of the verification script is the name of the verification script file (***.vse**). See remark to **RunVerificationScript(…)** for details.

Example

C++:

```
 // In this example, use wrapper functions provided by #import directive.

IusbTrace* Usb_trace;
. . .
IUsbVerificationScript* Usb_vscript = NULL;

Usb_trace->QueryInterface(IID_ IUsbVerificationScript, (void**)&Usb_vscript))
assert(Usb_vscript != NULL);

IVScriptEngine* Usb_vsengine = NULL;
Usb_vsengine = Usb_vscript -> GetVScriptEngine(ScriptName);

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

WSH:

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

Dim Result

Set VSEngine = Trace.GetVScriptEngine(ScriptName)
Result = VSEngine.RunVScript

If Result = 1 Then
       Msgbox "PASSED"
Else
       Msgbox "FAILED"
End If
MsgBox( "Done" )
```

3.15.3 IUsbVerificationScript::AddComments

```
HRESULT AddComments([in] BSTR comment);
```

Add a comment to the trace file. The value provided here is appended to the Comment field, which is found in the Trace Information window.

Parameters

`comment` Comment string

Return values

Remarks

The available space for comments in total is 127 characters.

Example

WSH:

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

Trace.AddComments("Recorded by James on the Rocinante")

MsgBox("Done")
```

4 Primary Dual Interface for Packet

4.1 IUsbPacket Interface

The **IUsbPacket** interface is the primary dual interface for the **UsbPacket** object.

IID : C7D7C98D-6AB2-4a25-8235-CE2E1E7A5428

4.1.1 IUsbPacket::GetTimestamp

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

Gets the timestamp of packet in nanosecond.

Parameters

`timestamp` Points to double value where timestamp is retrieved

Return values

Remarks

4.1.2 IUsbPacket::GetDuration

```
HRESULT GetDuration( [out, retval] double* duration )
```

Gets the duration of packet in nanosecond.

Parameters

`duration` Points to double value where duration is retrieved

Return values

Remarks

4.1.3 IUsbPacket::GetSpeed

```
HRESULT GetSpeed( [out, retval] int* speed )
```

Gets the speed of packet.

Parameters

`speed` Points to integer value where speed is retrieved

Return values

Remarks

Speeds has following values:
0 : High Speed
1 : Full Speed
2 : Low Speed
3 : Unknown
4 : Super Speed

4.1.4 IUsbPacket::GetChannel

```
HRESULT GetChannel( [out, retval] int* channel )
```

Gets the channel number of packet.

Parameters

`Channel` Points to integer value where channel is retrieved.

Return values

Remarks

Channel has following values:
0 : Usb2 Channel 0
1 : Usb2 Channel 1
2 : Usb2 Channel 2
3 : Usb2 Channel 3
4 : Usb3 Rx Channel 0
5 : Usb3 Tx Channel 0
6 : Usb3 Rx Channel 1
7 : Usb3 Tx Channel 1

4.1.5 IUsbPacket::GetType

```
HRESULT GetType(
        [out] VARIANT* is_usb3,
        [out] VARIANT* is_bus_condition,
        [out, retval] int* type )
```

Gets the type of packet.

Parameters

`is_usb3` Pointer to boolean variable indicating packet is usb3 or not.
`is_bus_condition` Pointer to boolean variable indicating packet is event or not.
`type` Pointer to integer variable where type of packet is retrieved.

Return values

Remarks

Type of packet has following values:

USB3 Packets and Events:
0 : Unknown
1 : TSEQ

2 : TSX
3 : LFPS
4 : Electrical Idle
5 : IPS
6 : LC
7 : SKP
8 : HP
9 : DPP
10 : Logical Idle
11 : TS1
12 : TS2
13 : DP
14 : BRST
15 : BERC
16 : BCNT
17 : TERM
18 : Deferred Packet
19 : CP

USB2 Packets:

-1 : Unknown
0 : MDATA
1 : DATA2
2 : DATA1
3 : DATA0
4 : SETUP
5 : SOF
6 : IN
7 : OUT
8 : STALL
9 : NYET
10 : NAK
11 : ACK
12 : PRE & ERR
13 : PING
14 : SPLIT
15 : EXT
19 : LPM

USB2 Events:

-1 : Unknown
0 : Chirp K
1 : Chirp J
2 : Full Speed K On High Speed
3 : Full Speed J On High Speed
4 : Suspend
5 : Resume
6 : SE1
7 : SE0
27 : Connect
28 : VBUS Voltage Change
30 : Keep Alive
31 : Reset

4.1.6 IUsbPacket::GetFieldValue

```
HRESULT GetFieldValue(
        [in] BSTR field_name,
        [out] VARIANT* field_value )
        [out, retval] VARIANT* field_value_bit_size )
```

Retrieves the value of specified field.

Parameters

field_name	String providing the field name that its value is retrieved.
field_value	Pointer to DWORD or bytes array where field value is retrieved. If the size of returned value is less than or equal to 4 bytes, the type is DWORD otherwise it is byte stream
field_value_bit_size	Pointer to Integer variable where size of field value in bit is retrived.

Return values

Remarks

To retrieve available fields names in this packet call **GetAllFieldsValues()** function.

4.1.7 IUsbPacket::GetAllFieldsValues

```
HRESULT GetAllFieldsValues(
        [out] VARIANT* fields_names,
        [out] VARIANT* fields_values,
        [out] VARIANT* fields_values_bit_size,
        [out, retval] int* fields_count  )
```

Retrieves the values of all available fields. It returns three arrays that contain name, value and size of fields.

Parameters

fields_names	Array of strings containing fields names.
fields_values	Array of DWORDs or bytes arrays containing fields values. If size of field value is less than or equal to 4 bytes it is DWORD otherwise bytes array.
fields_values_bit_size	Array of integers containing size of fields in bit.
fields_count	Pointer to integer variable containing count of fields.

Return values

Remarks

The bit order and byte order are little-Endian for all fields always.

4.1.8 IUsbPacket::GetMarker

```
HRESULT GetMarker( [out, retval] BSTR* marker )
```

Retrieves the marker text of current packet.

Parameters

`marker` Pointer to string variable where marker text is retrieved.

Return values

Remarks

4.1.9 IUsbPacket::GetErrorsBitmap

```
HRESULT GetErrorsBitmap( [out, retval] VARIANT* errors_bitmap )
```

Retrieves the errors of packet.

Parameters

`errors_bitmap` Pointer to unsigned 64 bits integer variable where errors are retrieved.

Return values

Remarks

Format of returned errors bitmap is as below:

Bit 0 : Reserved
Bit 1 : USB2 PID Error
Bit 2 : USB2 CRC5 Error
Bit 3 : USB2 CRC16 Error
Bit 4 : USB2 Packet Length Error
Bit 5 : USB2 Bit Stuff Error
Bit 6 : USB2 EOP Error
Bit 7 : USB2 Babble Start Error
Bit 8 : USB2 Babble End Error
Bit 9 : USB2 Frame Length Error
Bit 10 : USB2 Turnaround/Timeout Error
Bit 11 : USB2 Analyzer Internal Error
Bit 12 : USB2 Data Toggle Error
Bit 13 : USB2 Frame/uFrame Number Error
Bit 14 : USB2 Last Byte Incomplete Error
Bit 15 : USB2 OTG Signal Error

Bit 16 : USB3 CRC5 Error
Bit 17 : USB3 CRC16 Error
Bit 18 : USB3 CRC32 Error
Bit 19 : USB3 Disparity Error
Bit 20 : USB3 10 Bit Symbol Error
Bit 21 : USB3 Unknown Packet(IPS) Error
Bit 22 : USB3 Framing Symbol Error
Bit 23 : USB3 LC Data Symbol Error
Bit 24 : USB3 Bad Header Packet Length Error
Bit 25 : USB3 Bad Data Length Field Error

Bit 26 : USB3 SKP Symbol Error
Bit 27 : USB3 Control Endpoint Direction Error
Bit 28 : USB3 Missed DPH Error
Bit 29 : USB3 Missed DPP Error
Bit 30 : USB3 Setup DP Error
Bit 31 : USB3 Sequence Number Error

Bit 32 – 44 : Reserved

Bit 45 : USB3 TP Non-Zero Reserved Error
Bit 46 : USB3 Missed ITP Error
Bit 47 : USB3 Late/Early ITP Error

Bit 48 – 63 : Reserved

Note that below errors bits are valid after transaction level decoding:
Bit 4 : USB2 Packet Length Error
Bit 10 : USB2 Turnaround/Timeout Error
Bit 12 : USB2 Data Toggle Error
Bit 27 : USB3 Control Endpoint Direction Error
Bit 46 : USB3 Missed ITP Error
Bit 47 : USB3 Late/Early ITP Error

4.1.10 IUsbPacket::GetRawData

```
HRESULT GetRawData(
        [out] VARIANT* raw_data_array,
        [out, retval] int* array_size  )
```

Retrieves the marker text of current packet.

Parameters

`raw_data_array`	Array of bytes where raw data of packet is retrieved.
`array_size`	Pointer to integer variable containing returned bytes count.

Return values

Remarks

For bus conditions returned array is empty.

4.1.11 IUsbPacket::GetUsb3ScrambledData

```
HRESULT GetUsb3ScrambledData(
        [out] VARIANT* scrambled_data_array,
        [out, retval] int* array_size )
```

Retrieves the scrambled raw packet representation of USB3 packets.

Parameters

`scrambled_data_array`	Bytes array variable where scrambled bytes of packet are retrieved.
`array_size`	Pointer to integer variable containing count of returned bytes.

Return values

Remarks

4.1.12 IUsbPacket::GetUsb3TenBitData

```
HRESULT GetUsb3TenBitData(
        [out] VARIANT* symbles_array,
        [out, retval] int* array_size )
```

Retrieves the ten bit symbols of USB3 packets.

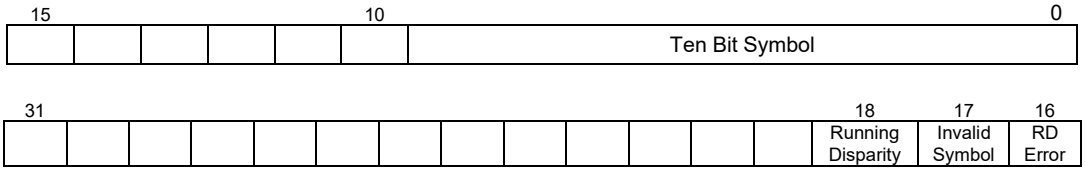
Parameters

`symbles_array`	Array of DWORDs containing returned symbols.
`array_size`	Pointer to integer variable containing count of returned symbols.

Return values

Remarks

Format of returned symbols are as below:



5 Primary Dual Interface for USB Verification Script Engine

5.1 IVScriptEngine interface

The **IVScriptEngine** interface is the primary dual interface for the **UsbVScriptEngine** object. It implements the common functionality for the Teledyne LeCroy PSG verification script engine (VSE). The main advantage of the **IVScriptEngine** interface is that it allows clients to implement the **_IVScriptEngineEvents** callback interface to receive notifications when a verification script is running.

IID : 67FC41C6-0FEE-4b95-8EE0-A5724661FE98

5.1.1 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 variable in which to store current verification script name
`newVal`	Name of the verification script to initialize script verification engine

Return values

Remarks

The name of the verification script is the name of the verification script file (***.vse**).

Example

C++:

```
// In this example, use wrapper functions provided by #import directive.

IUsbTrace* usb_trace;

. . .

IUsbVerificationScript* usb_vscript = NULL;

usb_trace->QueryInterface( IID_ IUsbVerificationScript, (void**)&usb_vscript ) )
assert( usb_vscript != NULL );

IVScriptEngine* usb_vsengine = NULL;
usb_vsengine = usb_vscript -> GetVScriptEngine(ScriptName);

usb_vsengine -> put_VScriptName(Test_1);
assert( usb_vsengine -> GetVScriptName() == Test_1 );

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

. . .
```

5.1.2 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 is used in event notifications, allowing a client event handler to determine which VSE object sent the event.

Parameters

`*pVal`	Address of variable in which to store current VSE tag
`newVal`	New tag for VSE

Return values

Remarks

Example

C++:

```
// In this example, use wrapper functions provided by #import directive.

IUsbTrace* usb_trace;

. . .

IUsbVerificationScript* usb_vscript = NULL;

usb_trace->QueryInterface( IID_ IUsbVerificationScript, (void**)&usb_vscript ) )
assert( usb_vscript != NULL );

IVScriptEngine* usb_vsengine = NULL;
usb_vsengine = usb_vscript -> GetVScriptEngine(ScriptName);


usb_vsengine ->put_Tag( 0xDDAADDAA );
assert( usb_vsengine -> GetTag() == 0xDDAADDAA );

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

. . .
```

5.1.3 IVScriptEngine::RunVScript

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

Runs verification script.

Parameters

`pResult`	Address of variable in which to store the result of verification. See IUsbVerificationScript::RunVerificationScript method for details.

Return values

Remarks

This method makes a “synchronous” call, so this method does not return until the script stops running.

Example

See C++ example to **VScriptName**.

5.1.4 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`	Name of the verification script to initialize script verification engine.
`PResult`	Address of variable in which to store the result of verification. See IUsbVerificationScript::RunVerificationScript method for details.

Return values

Remarks

This method makes a “synchronous” call, so this method does not return until the script stops running.

Example

C++:

```
// In this example, use wrapper functions provided by #import directive.

IUwbTrace* usb_trace;

. . .

IUsbVerificationScript* usb_vscript = NULL;

usb_trace->QueryInterface( IID_ IUsbVerificationScript, (void**)&usb_vscript ) )
assert( usb_vscript != NULL );

IVScriptEngine* usb_vsengine = NULL;
usb_vsengine = usb_vscript -> GetVScriptEngine(ScriptName);


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

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

result = usb_vsengine ->RunVScriptEx(Test_3);
if( result == PASSED )
{
        ::MessageBox( NULL, "Test 3 passed !!!", "UsbAnalyzer client", MB_OK );
}

. . .
```

5.1.5 IVScriptEngine::LaunchVScript

```
HRESULT LaunchVScript();
```

Launches verification script.

Parameters

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, so this method immediately returns after the script starts running.

When the verification script stops running, the VSE object sends a special event notification "OnVScriptFinished" to the client event handler. You can also terminate script running using the method “Stop”.

Example

C++:

```
// In this example, use wrapper functions provided by #import directive.

IUsbTrace* usb_trace;

. . .

IUsbVerificationScript* usb_vscript = NULL;

usb_trace->QueryInterface( IID_ IUsbVerificationScript, (void**)&usb_vscript ) )
assert( usb_vscript != NULL );

IVScriptEngine* usb_vsengine = NULL;
usb_vsengine = usb_vscript -> GetVScriptEngine(ScriptName);
assert( usb_vsengine != NULL );

VS_RESULT result = usb_vsengine ->LaunchVScript();

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

. . .
```

5.1.6 IVScriptEngine::Stop

```
HRESULT Stop();
```

Stops verification script previously launched by the IVScriptEngine::LaunchVScript method.

Parameters

Return values

Remarks

Example

C++:

```
// In this example, use wrapper functions provided by #import directive.

IUsbTrace* usb_trace;

. . .

IUsbVerificationScript* usb_vscript = NULL;

usb_trace->QueryInterface( IID_ IUsbVerificationScript, (void**)&usb_vscript ) )
assert( usb_vscript != NULL );

IVScriptEngine* usb_vsengine = NULL;
usb_vsengine = usb_vscript -> GetVScriptEngine(ScriptName);

VS_RESULT result = usb_vsengine ->LaunchVScript();

. . .

if( NotEnoughResourcesToProcessVS )
        usb_vsengine ->Stop();

. . .
```

5.1.7 IVScriptEngine::GetScriptVar

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

Returns the value of some verification script variable before/after executing the script. The resulting value may contain an integer, a string, or an array of VARIANTs (if the requested script variable is a list object – see the *CSL Manual* for more details about list objects).

Parameters

`var_name`	String providing the name of the global variable or constant used in the running verification script.
`var_value`	Address of a VARIANT variable in which result will be stored.

Return values

`E_PENDING`	If this method is called when script running is in progress

Remarks

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

Example

C++:

```
// In this example, use wrapper functions provided by #import directive.

IUsbTrace* usb_trace;

. . .

IUsbVerificationScript* usb_vscript = NULL;

usb_trace->QueryInterface( IID_ IUsbVerificationScript, (void**)&usb_vscript ) )
assert( usb_vscript != NULL );

IVScriptEngine* usb_vsengine = NULL;
usb_vsengine = usb_vscript -> GetVScriptEngine(ScriptName);

VS_RESULT result = usb_vsengine ->RunVScript();

. . .

VARIANT my_var;
VariantInit( &my_var );

usb_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(ScriptName) ' Get VS Engine object.

Result = VSEngine.RunVScript

MyIntVar = VSEngine.GetScriptVar( "MyIntVar" ) ' Suppose MyVar contains an integer
MyStrVar = VSEngine.GetScriptVar( "MyStrVar" ) ' Suppose MyVar contains a string.

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

5.1.8 IVScriptEngine::SetScriptVar

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

Allows you to set the value of some verification script variable before executing the script. Only integers, strings, or arrays of VARIANTs are allowed. Arrays of VARIANTs are converted into list values inside of scripts – see the *CSL Manual* for more details about list objects.

Parameters

`var_name`	String providing the name of the global variable used in the running verification script.
`var_value`	VARIANT value containing new variable value.

Return values

`E_PENDING`	If this method is called when script running is in progress.

Remarks

This function allows you to set internal script variables before/after script run, so you can make run-time customizations from COM/Automation client applications.

If there is no such global variable with the name 'var_name', a new variable with that name is created.

Example

C++:

```
// In this example, use wrapper functions provided by #import directive.

IUsbTrace* usb_trace;

. . .

IUsbVerificationScript* usb_vscript = NULL;

usb_trace->QueryInterface( IID_ IUsbVerificationScript, (void**)&usb_vscript ) )
assert( usb_vscript != NULL );

IVScriptEngine* usb_vsengine = NULL;
usb_vsengine = usb_vscript -> GetVScriptEngine(ScriptName);

VARIANT my_var;
VariantInit( &my_var );

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

// Set internal script variable 'MyVar' to 100.
usb_vsengine->SetScriptVar( _bstr_t("MyVar"), my_var );

VS_RESULT result = usb_vsengine ->RunVScript();

. . .
```

WSH:

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

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

6 Verification Script Engine Events Callback Interface

6.1 _IVScriptEngineEvents dispinterface

To retrieve the event notifications from the **UsbAnalyzer** application when a verification script engine object is running a script, you must implement the **_IVScriptEngineEvents** callback interface.

Because this interface is a default source interface for the **IVScriptEngine** object, there is a very simple implementation from such languages as Visual Basic, VBA, VBScript, and WSH.

Remarks

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). Teledyne LeCroy PSG USB Protocol Suite™ provides a special COM class that allows receiving and handling notifications from VSE objects, even in script languages not supporting event handling from "indirect" objects.
(Please refer to CATCAnalyzerAdapter for details.)

Example

The C++ implementation used in the examples below implements an event sink object by deriving 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 to be handled:

C++:

```
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 establish the connection with the server, you need to advise your implementation of the event interface:

```
IVScriptEngine vscript_engine = NULL;

    try
    { vscript_engine = vscript ->GetVScriptEngine(ScriptName ); }
    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 USBTracer As UsbAnalyzer
Public Trace     As UsbTrace
Public GVSEngine As VScriptEngine

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

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


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

        If USBTracer Is Nothing Then
                Set USBTracer = New UsbAnalyzer

                If USBTracer Is Nothing Then
                        MsgBox "Unable to connect to USB Protocol Suite", vbExclamation
                        Exit Sub
                End If
        End If

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

        Set IVScript = Trace      'Get the IUsbVerificationScript 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
```

6.1.1 _IVScriptEngineEvents::OnVScriptReportUpdated

```
[id(1)] HRESULT OnVScriptReportUpdated([in] BSTR newLine,
                                       [in] int TAG );
```

Fires when running verification script calls **ReportText(newLine)** function.

Parameters

`newLine`	New portion of text reported by verification script
`TAG`	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
```

6.1.2 _IVScriptEngineEvents::OnVScriptFinished

```
[id(2)] HRESULT OnVScriptFinished ( [in] BSTR script_name,
                                    [in] VS_RESULT result, [in] int TAG );
```

Fires when the verification script stops running.

Parameters

`script_name`	Name of the verification script
`result`	Result of the "verification" (See IUsbVerificationScript::RunVerificationScript for details.)
`TAG`	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 UWBAutomationLib.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
```

6.1.3 _IVScriptEngineEvents::OnNotifyClient

```
[id(3)] HRESULT OnNotifyClient( [in] int eventId,
                                [in] VARIANT eventBody,
                                [in] int TAG );
```

Fires when running verification script calls **NotifyClient()** function.

Parameters

`EventId`	Event Id
`eventBody`	Body of event packed in a VARIANT object
`TAG`	VSE object's tag

Return values

Remarks

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

Example

Teledyne LeCroy PSG USB 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
```

7 Primary Dual Interface for Recording Options

7.1 IUsbRecOptions Dual Interface

The **IUsbRecOptions** interface is the primary interface for the **IBRecOption** object. It derives from the **IRecOptions** interface that implements the common functionality for all Teledyne LeCroy Analyzers.

7.1.1 IRecOptions::Load

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

Loads recording options from the specified file.

Parameters

ro_file_name	String providing 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.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.Load( CurrentDir & "Input\rec_options.rec" )
```

C++:

7.1.2 IRecOptions::Save

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

Saves recording options into the specified file.

Parameters

ro_file_name	String providing 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 is created. If it exists, it is overwritten.

Example

WSH:

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

C++:

7.1.3 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 the snapshot recording mode.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetRecMode 2 ' Event trigger
```

C++:

7.1.4 IRecOptions::SetBufferSize

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

Sets the size of buffer to record.

Parameters

buffer_size	Buffer size in bytes

Return values

E_INVALIDARG	Invalid buffer size was specified.

Remarks

The default setting is 1 MB.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetBufferSize 2*1024*1024 ' 2 MB
```

C++:

7.1.5 IRecOptions::SetPostTriggerPercentage

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

Sets the post-trigger buffer size.

Parameters

posttrigger_percentage	Size of post trigger buffer in percents of the whole recording buffer (see 7.1.4)

Return values

E_INVALIDARG	Invalid percentage was specified.

Remarks

This method call has no effect if recording mode was set to **RMODE_SNAPSHOT**.
The default setting is 50%.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetPostTriggerPercentage 60 ' 60%
```

C++:

7.1.6 IRecOptions::SetTriggerBeep

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

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

Parameters

`beep`	TRUE to beep when trigger occurs. FALSE to not beep when trigger occurs.

Return values

Remarks

The default state of the beeper is FALSE.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetTriggerBeep TRUE
```

C++:

7.1.7 IRecOptions::SetDataTruncate

```
HRESULT SetDataTruncate (
        [in] long length );
```

Sets the flag indicating that recorded data is to be truncated and sets the length of data to truncate.

Parameters

`length` Length of data in bytes; cannot be less then 108

Return values

Remarks

By default, data is not truncated.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetDataTruncate 345 ' Truncate data that is more then 345 bytes long.
```

C++:

7.1.8 IRecOptions::SetAutoMerge

```
HRESULT SetAutoMerge (
      [in] BOOL auto_merge );
```

Sets the flag indicating that recorded traces on different channels are merged automatically after recording is done.

Parameters

auto_merge	TRUE performs merge automatically. FALSE does not perform merge.

Return values

Remarks

By default, automatic merge is not performed.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetAutoMerge TRUE
```

C++:

7.1.9 IRecOptions::SetSaveExternalSignals

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

Sets the flag indicating to save external signals.

Parameters

save	TRUE saves external signals. FALSE does not save external signals.

Return values

Remarks

By default, external signals are not saved.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetSaveExternalSignals TRUE
```

C++:

7.1.10 IRecOptions::SetTraceFileName

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

Sets the path to the file where the trace is stored after recording.

Parameters

`file_name` String providing the full pathname to the file where recording is stored

Return values

Remarks

If specified file does not exist, it is created. If it exists, it is overwritten.

Example

WSH:

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

C++:

7.1.11 IRecOptions:: SetFilterPolarity

```
HRESULT SetFilterPolarity (
     [in] BOOL filter_out );
```

Sets whether to filter recording events in or out.

Parameters

`filter_out`	TRUE = filter out FALSE = filter in

Return values

`E_INVALIDARG`	Operation code and/or connection was specified.

Remarks

By default, all events are filtered out.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetFilterPolarity 0 ' filter in
```

C++:

7.1.12 IRecOptions::Reset

```
HRESULT Reset ( );
```

Resets recording options to its initial state.

Parameters

Return values

Remarks

For default values of recording options, see the remarks.

Example

WSH:

```
CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
Set Analyzer = WScript.CreateObject("CATC.USBTracer")
Set RecOptions = Analyzer.GetRecordingOptions
RecOptions.SetRecMode 2             ' Event trigger
RecOptions.SetBufferSize 1024*1024  ' 1 MB
RecOptions.SetPostTriggerPercentage 60  ' 60%
. . .
RecOptions.Reset
```

C++:

7.2 IUsbRecOptions2

The **IUsbRecOptions** interface is the second interface for the **RecOption** object. It derives from the **IUSBRecOptions** interface that implements the common functionality for all Teledyne LeCroy Analyzers.

IID: `699CAC9B-E32A-40dd-92E1-8B237057B67F`

7.2.1 IUSBRecOptions2::ChangeUsbRecordingOptionSetting

```
HRESULT ChangeUsbRecordingOptionSetting
    ( [in] int setting, [in] int value, [out, retval] int* result )
```

Method provides capability to change specific setting of loaded recording option before recording start.
NOTE: FOR INTERNAL TELEDYNE LECROY USE ONLY.

Parameters

`setting` specifies desired setting to change [look at remarks for detail]
`value` new value of setting
`result` pointer to integer value which returns setting modify was successful or not

Return values

Method returns ANALYZERCOMERROR_WRONGCALL exception if setting id is invalid.

Remarks

The setting can be one of these values:

setting	Possible Values	Description
1	0 : Off 1 : Host Emulation 2 : Device Emulation	USB3 - Generation Mode
2	0 : Off 1 : Auto 2 : On	USB3 - Rx/Tx Descrambling
3	0 : Off 1 : Auto 2 : On	USB3 - Rx/Tx Polarity
4	0 : Off 1 : On	USB3 - Rx Spread Spectrum Clock
5	0 : Off 1 : On	USB3 - Rx Spread Spectrum Clock
6	0 : Proxied from other port 1 : Always presented	USB3 - Analyzer Ports Termination
7	0 : No 1 : Yes	USB3 - Exerciser Rx Detect Off
8	0 : Off 1 : -5300ppm 2 : +300ppm	USB3 - Tx Clock (Compliance Only)

10	0 : Off 1 : Auto 2 : On	USB3 - Rx Descrambling
11	0 : Off 1 : Auto 2 : On	USB3 - Tx Descrambling
12	0 : Off 1 : Auto 2 : On	USB3 - Rx Polarity
13	0 : Off 1 : Auto 2 : On	USB3 - Tx Polarity
14	0 : No 1 : Yes	General - Disable CATC Sync
15	0 : No 1 : Yes	General - CATC Sync Independent Record Stop
16	0 : No 1 : Yes	General - CATC Sync Independent Trigger
17	0 : No 1 : Yes	Ignore Uploading
18	0 : Hardware Sampling 1 : Legacy Method	USB3 - M3x LFPS Capture Method
19	0 : No 1 : Yes	General - Capture Raw File
20	0 : Gen1 1 : Gen2	USB3 – Generation Speed
23	0 : Auto 1 : Gen1 2 : Gen2	USB3 – Analyzer Speed
24	0 : No 1 : Yes	Enable External Trigger Custom Pulse Width
25	Number less than 4096	External Trigger Pulse Width in nanosecond
26	0 : No 1 : Yes	Power Delivery - Allow Vbus without CC-pin Termination
27	Text	General – Connected Device name (Connector 1)
28	Text	General – Connected Device name (Connector 2)
31	0 : No 1 : Yes	Capture CC Traffic
34	0 : No 1 : Yes	General - Capture Power Measurements
35	Number	General – Recording Buffer size in Megabyte
36	0x0000 : Auto 0x0001 : Source 0x0002 : Sink 0x0004 : DRP 0x0008 : Audio Adapter 0x0010 : DTS 0x0020 : Support Audio Acc 0x0040 : Support Debug Acc 0x0080 : Support Vconn-Pwd	USB Type-C – Right Device Information
37		USB Type-C – Left Device Information

	0x0100 : Try.SRC 0x0200 : Try.SNK	
38	0 : No 1 : Yes	USB3 – Support Legacy Timers (Exerciser)
39	number (1 to 100)	Increase the Recording Buffer size by the input percentage
40	0 : Enabled 1 : Disabled – Start With TERM On 2 : Disabled – Start With TERM Off 3 : Disabled – Keep TERM State	USB3 – Exerciser LTSSM Controlled Termination
41	Number	USB Type-C – Source Vbus On Threshold in millivolt
42	Number	USB Type-C – Sink Vbus On Threshold in millivolt
43	0 : x1 1 : x2	USB3 – Generation Link Width
44	0 : No 1 : Yes	Power Delivery – Allow Vbus > 5V
45	1 : Pulse Low 2 : Pulse High 3 : Pulse Toggle	General – External Trigger Out Type
46	0 : Off 1 : On	General – Recording channel USB3
47	0 : Off 1 : On	General – Recording channel USB4/TBT
48	0 : Off 1 : On	General – Recording channel USB2
49	0 : Off 1 : On	General – Recording channel PD
50	0 : Off 1 : On	General – Recording channel DP AUX

7.3 IusbRecOptions3

The **IUsbRecOptions** interface is the third interface for the **RecOption** object. It derives from the **IUSBRecOptions2** interface.

IID: 689F24F4-1AE5-412D-AA93-F75686E77532

7.3.1 IUSBRecOptions3::SetBufferSizeEx

```
HRESULT SetBufferSizeEx ( [in] DWORD num_megabytes )
```

This method is new alternative to SetBufferSize which parameter accepts recording size in mega bytes. So, API users can assign recording sizes bigger than 4G [depends on analyzer product buffer].

Parameters

Num_megabytes	specifies desired recording size in mega bytes

Return values

Remarks

8 Errors Collection Interface

8.1 IAnalyzerErrors dispinterface

This is a standard collection interface for collection of errors of a specified type (see ITrace::AnalyzerErrors). It has the following standard methods.

8.1.1 IAnalyzerErrors::get_Item

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

Parameters

`index`	Index of error in the collection
`packet_number`	Points to long value where error packet number is retrieved.

8.1.2 IAnalyzerErrors::get_Count

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

Parameters

`number_of_errors` Points to long value where number of elements in the collection is retrieved.

Remarks

Example

```
WSH:
        ' Makes recording, and
        ' saves the portions of the recorded trace where "Bad VCRCs" errors occurred.
        CurrentDir = Left(WScript.ScriptFullName, InstrRev(WScript.ScriptFullName, "\"))
        Set Analyzer = WScript.CreateObject("CATC.USBTracer")
        Set Trace = Analyzer.MakeRecording (CurrentDir & "Input\test_ro.rec")
        Set Errors = Trace.AnalyzerErrors (16) ' Packet Length Error
        For Each ErrorPacketNumber In Errors
              ErrorFile = CurrentDir & "\Output\ PckLen_error_span_" & CStr(ErrorPacketNumber)
        & ".usb"
              Trace.Save ErrorFile, CInt(ErrorPacketNumber)-5, CInt(ErrorPacketNumber)+5
        Next
```

C++:

```
IUsbTrace* usb_trace;

. . .

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

analyser_errors->Release();

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

9 Analyzer Events Callback Interface

9.1 _IAnalyzerEvents dispinterface

In order to retrieve the events from the USB Protocol Suite application, you must implement the **_IAnalyzerEvents** interface.

Since this interface is the default source interface for the **UsbAnalyzer** object, there is a very simple implementation from such languages as Visual Basic, VBA, VBScript, WSH, etc.

C++ implementation used in the examples below implements 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 which 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 establish the connection with the server, you need to advise your implementation of the event interface:

```
hr = CoCreateInstance( CLSID_UsbAdvisor, NULL,
                   CLSCTX_SERVER, IID_IUsbAnalyzer, (LPVOID *)&m_poUsbAnalyzer );

m_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_poUsbAnalyzer, &m_poAnalyzerSink->m_libid,
        &m_poAnalyzerSink->m_iid, &m_poAnalyzerSink->m_wMajorVerNum,
        &m_poAnalyzerSink->m_wMinorVerNum);

 if ( FAILED(hr) )
        return 1;

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

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

9.1.1 _IAnalyzerEvents::OnTraceCreated

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

Fired when trace is created.
This event is a result of an **IAnalyzer::StartRecording** or **IAnalyzer::StopRecording** method call.

Parameters

`trace` Address of a pointer to the **UsbTrace** object primary interface

Return values

Remarks

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

Example

```
VBScript:

        <OBJECT
                ID = Analyzer
                CLASSID = "clsid:0B179BB3-DC61-11d4-9B71-000102566088" >
        </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 )
        {
                IUsbTrace* usb_trace;
                HRESULT hr;
                hr = trace->QueryInterface( IID_IUsbTrace, (void**)&usb_trace );
                if (FAILED(hr))
                {
_com_error er(hr);
if (er.Description().length() > 0)
    ::MessageBox( NULL, er.Description(), _T("UsbAnalyzer client"), MB_OK );
else
    ::MessageBox( NULL, er.ErrorMessage(),_T("UsbAnalyzer client"), MB_OK );
return hr;
                }

                . . .
                return hr;
        }
```

9.1.2 _IAnalyzerEvents::OnStatusReport

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

Fires when there is a change in Analyzer state or there is a change in progress (percent_done) of the Analyzer state.

Parameters

`subsystem` Subsystem sending event has the following values:

`RECORDING_PROGRESS_REPORT ( 1 )` – recording subsystem

`GENERATION_PROGRESS_REPORT ( 2 )` – recording subsystem

`state` Current Analyzer state; has the following values:

If `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
`ANALYZERSTATE_RT_UPLOADING_DATA`	`( 8 )`	Recording and uploading data concurrently

If `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, (valid only 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 or ANALYZERSTATE_RT_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.

When Analyzer state is `ANALYZERSTATE_UPLOADING_DATA_DONE`, this parameter is equal to Uploaded MBytes.

Return values

Remarks

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

Example

VBScript:

```
<OBJECT
        ID = Analyzer
        CLASSID = "clsid:0B179BB3-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 );
```

10 CATCAnalyzerAdapter

Script languages, such as the Visual Basic (VB) and Javascript script languages used in HTML pages and Windows Script Host (WSH), can implement and automate exposed functionalities in client applications. However, such script engines cannot efficiently create automation objects dynamically, handle events, or control operation remotely. For example, an HTML page script language can only handle events from an automation object if the object is created with the <OBJECT> tag when the page is loaded. If the object is created at runtime, events from the object cannot be handled. An automation object can only be launched remotely if the name of the remote server is already known when the script client begins running. If the remote server is not yet known, automation objects cannot be launched.

The Teledyne LeCroy USB Analyzer COM API includes an additional COM server, **CATCAnalyzerAdapter**, to provide full support for automation using script languages. The **CATCAnalyzerAdapter** automation server:

- Allows launching and accessing Teledyne LeCroy analyzer automation servers dynamically at runtime.
- Allows launching and accessing Teledyne LeCroy analyzer automation servers remotely, when the remote server has all necessary DCOM settings and permissions.
- Handles automation events from Teledyne LeCroy analyzer automation servers.
- Overrides limitations imposed by the script engines used in HTML browsers and Windows Script Host.

The following diagram and the examples below show how the **CATCAnalyzerAdapter** automation server is used as an intermediate between an HTML page and the USB Protocol Suite analyzer server to create automation objects dynamically, handle events, and control operation remotely.

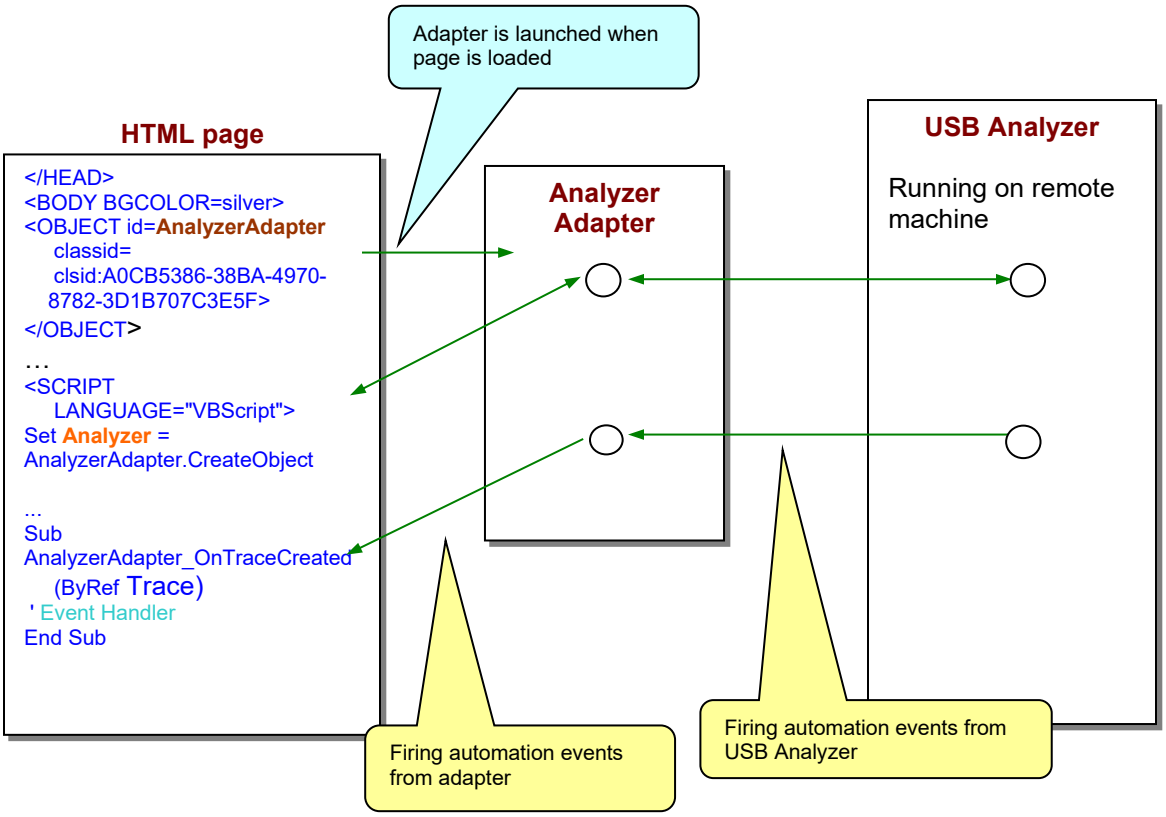


Figure 6-1 Analyzer Adapter

10.1 IAnalyzerAdapter Interface

```
Classid:                        A0CB5386-38BA-4970-8782-3D1B707C3E5F
ProgID:                         CATC.AnalyzerAdapter
COM server:                     CATCAnalyzerAdapter.exe

Primary default interface:      IAnalazerAdapter
```

10.1.1 IAnalyzerAdapter::CreateObject

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

This method instantiates a 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: `0B179BB3-DC61-11d4-9B71-000102566088`" or "`CATC.USBTracer`" for CATC USB Analyzer)
`host_name`	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 **Detach** method (see below) should be called when 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.USBTracer", RemoteServer.value)

                if Not Analyzer Is Nothing Then
                        window.status = "USBTracer connected"
                else
                        msg = "Unable to connect to USBTracer"
                        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.USBTracer", RemoteServer)

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

10.1.2 IAnalyzerAdapter::Attach

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

This method attaches a 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 was previously attached to the adapter, it is 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.USBTracer")
		' VBScript function creates object locally.

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

			window.status = "USBTracer connected"
		else
			msg = "Unable to connect to USBTracer"
			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")
	...
```

10.1.3 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 does not guarantee that all resources associated with the detached object are 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.USBTracer", RemoteServer.value)

        if Not Analyzer Is Nothing Then
                window.status = "USBTracer connected"
        else
                msg = "Unable to connect to USBTracer"
                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 = "USBTracer disconnected"
End Sub
-->
</SCRIPT>
```

WSH:

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

RemoteServer = "EVEREST"
Set Analyzer = AnalyzerAdapter.CreateObject("CATC.USBTracer", 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
```

10.1.4 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.

pObj	Pointer to the object validated
pVal	Pointer to the variable receiving result TRUE if the validated object can be attached FALSE otherwise.

Parameters

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 USBTracer!!!
       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>
```

Appendix A. DCOM Configuration

USB*Tracer*™, USB Advisor™, and Voyager™ USB Protocol Suite Automation can be run locally or remotely. This appendix describes how to configure Automation to run over a network. Automation uses Distributed Component Object Model (DCOM) to manage the transmission of automation commands over a network. When Automation is set up for remote usage, the Host Controller is configured as a DCOM server and the remote PC is configured as a DCOM client.

A.1. Running Automation Locally

If you intend to run Automation on the Host Controller (the PC attached to the Analyzer), you do not need to perform any special configuration. You can simply execute the scripts or programs you have created and they run the Analyzer.

A.2. Setting Up Automation for Remote Use

If you intend to run automation remotely over a network, you must perform DCOM configuration. These steps are described below:

Summary of the Steps

1. Run the Microsoft™ utility **dcomcnfg** or **dcom98** on the Host Controller and configure the Host Controller so that it can be controlled by another device. For Windows™ 2000 and NT 4.0, use **dcomcnfg.** For Windows 98, use **dcom98. Dcom98** is not bundled with the operating system, so you must install it first.

2. Run **dcomcnfg** or **dcom98** on the remote PC.

3. Using **dcomcnfg** or **dcom98**, configure the remote PC so that it activates the Host Controller by default.

4. Test your DCOM configuration by running a script file on the remote PC. Teledyne LeCroy provides some sample script files that you can use for this test.

5. If you write a program in C++, make sure that the Host Controller is registered on both the client and server machine.

A.3. DCOM Configuration for Host Controller

To configure DCOM on the Host Controller, you run a DCOM configuration utility called **dcomcnfg**. On Windows 2000 and Windows NT 4.0, DCOM and the **dcomcnfg** are bundled into the operating systems. On Windows 98, DCOM is not bundled into the operating system. You must first install DCOM before you can use **dcomcnfg** to configure the Analyzer for remote use.

When **dcomcnfg** is run, it presents a list of installed applications that support DCOM. To configure an application for DCOM, select an application from the list, then apply security and launch settings.

This appendix describes the process for configuring security and launch settings to the Analyzer for Windows 2000 and Windows NT 4.0 systems. If you are using Windows 98, you must first install DCOM before you can use **dcomcnfg**. The screens for Windows 98 are a little different from what you see in this appendix.

Summary of DCOM Server Configuration Steps

DCOM server configuration for the Host Controller involves three steps:

1. Configure DCOM Authentication to **Default**.

2. Configure DCOM launch permissions to **Everyone**.

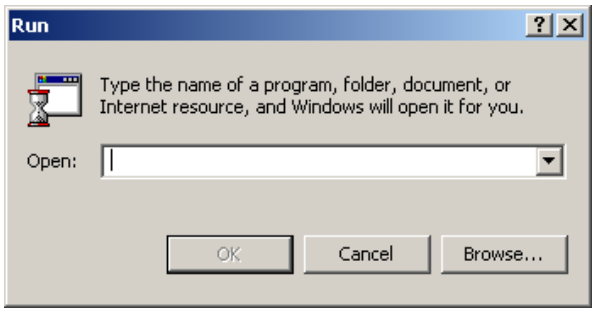
3. Configure Run Identity with your login name.

Configuring When and How Authentication Should Occur

The **dcomcnfg** utility presents a number of security options. In the steps that follow, configure the Analyzer application on the Host Controller to authenticate the user when the user first attempts a connection.

1. Click the Windows **Start** button.

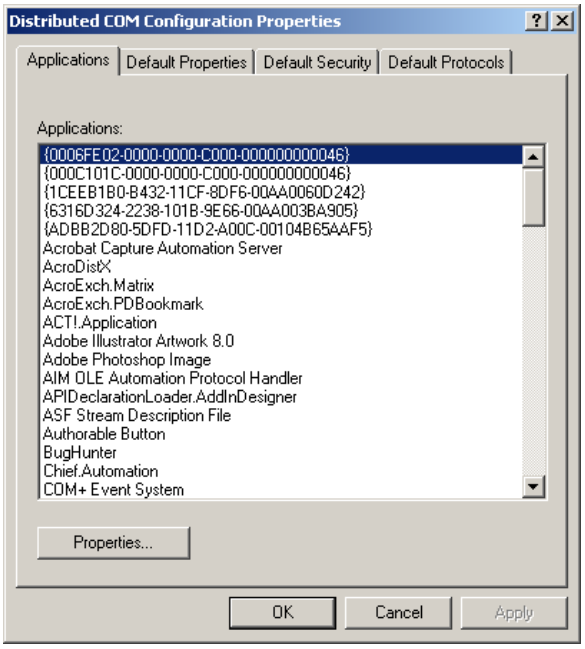
2. Select **Run ...**
   The Run dialog appears.



3. In the Open edit box within the Run dialog, type **dcomcnfg**.

4. Click **OK**.

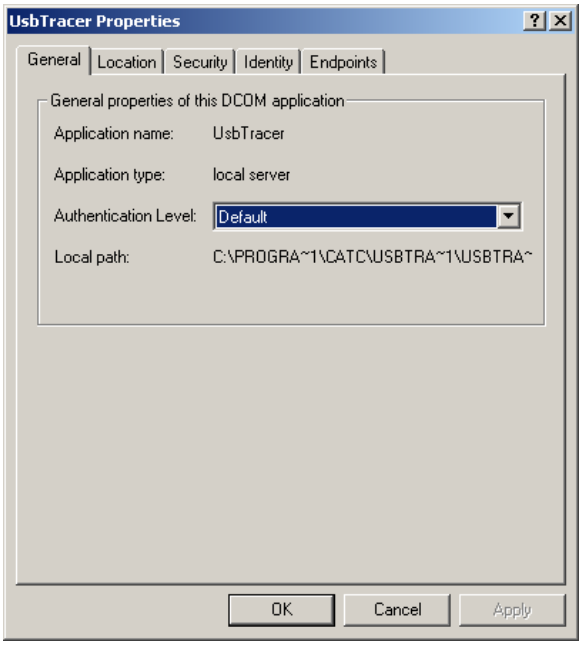
The DCOM configuration dialog box opens.



5. In the **Applications** tab, scroll down the list of applications and select **USBTracer/Trainer, USB Advisor**, or **Voyager (USB Protocol Suite)**.

6. Click the **Properties** button.

The Properties dialog box opens. The options in this dialog box allow you to configure security on the selected application.



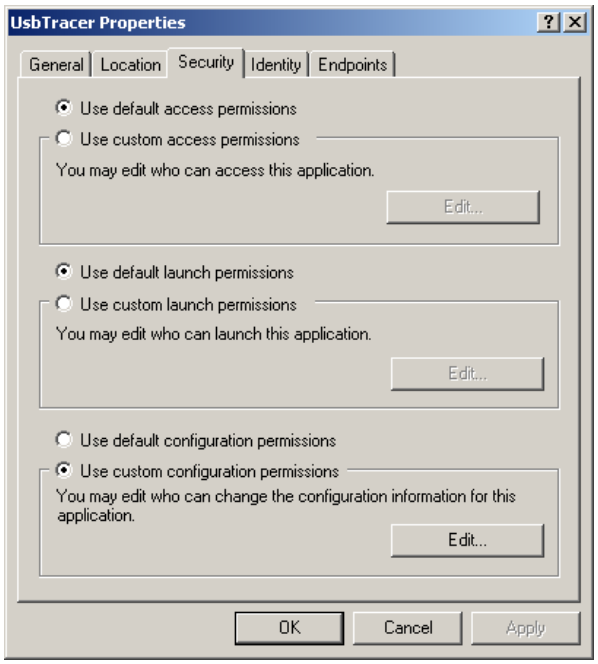
7. From the Authentication Level menu, select **Default**.

The Authentication Level menu lets you set packet-level security on communications for the selected application.

Configuring Access Permissions

Access permission determines who may execute commands on the application once it is running. In the following steps, you give everyone access to the application on the Host Controller.

1. With the Properties dialog box still displayed, select the **Security** tab.
   The following settings display:



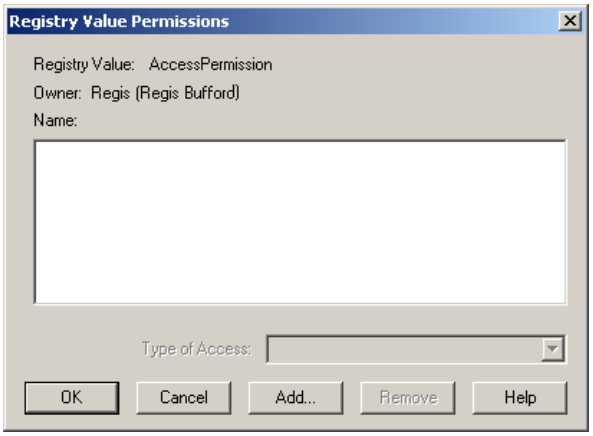
You see three options:

- **Access Permissions**: Determines who may execute commands on the application once it has been started
- **Launch Permissions**: Determines who may launch the application
- **Configuration Permissions**: Determines who may configure permissions for the application

2. Click the **User Custom Access** permissions option.
   You are going to give all users permission to execute commands on the server.

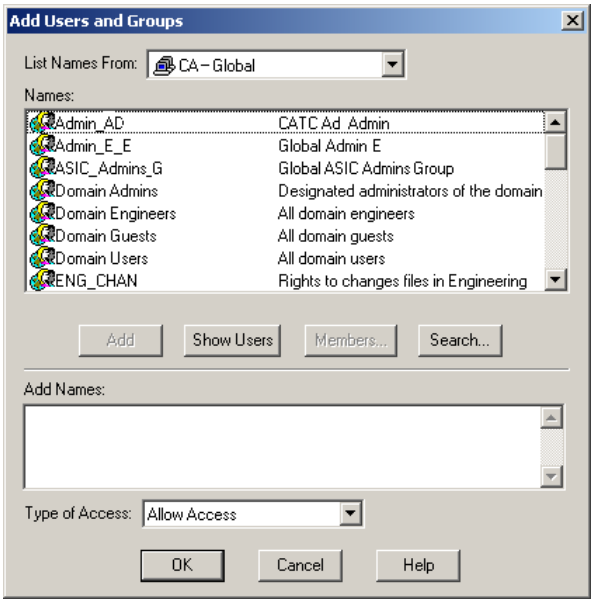
3. Click the **Edit ...** button.

The Registry Value Permissions dialog box appears.



4. Click the **Add ...** button.

The Add Users and Groups dialog box appears.



5. Select the group called **Everyone**.

6. Click the **Add** button.

7. Select the group **System**.

8. Click the **Add** button.

9. Click **OK**.

*The **Add Users** dialog box closes.*

10. Click **OK**.

The Add Users and Groups dialog box closes. The Registry Key Permissions dialog box remains on the screen.

11. Click **OK**.

The Registry Key Permissions dialog box closes. The Properties dialog box remains on the screen.

Configuring Launch Permissions

Launch Permissions control who can start an application. In the following steps, you give everyone permission to launch the application on the Host Controller.

1. In the Security tab of the server Properties dialog box, select the option **Custom Launch Permissions**.

2. Click the **Edit ...** button.
   The Registry Value Permissions dialog box appears.

3. Click the **Add ...** button.
   The Add Users and Groups dialog appears.

4. Select the group called **Everyone**.
   *If you prefer, you can select individual user accounts instead of **Everyone**.*

5. Click the **Add ...** button.

6. Click **OK**.
   The Add Users and Groups dialog box closes. The Registry Key Permissions dialog box remains on the screen.

7. Click **OK**.
   The Registry Key Permissions dialog box closes. The Properties dialog box remains on the screen.

Configuring Configuration Permissions

You are going to give everyone permission to configure permissions for the application on the Host Controller.

1. In the Security tab of the server Properties dialog box, select the option **User Custom Configuration Permissions**.

2. Click the **Edit ...** button.
   The Registry Value Permissions dialog box appears.

3. Click the **Add ...** button.
   The Add Users and Groups dialog appears.

4. Select the group called **Everyone**.
   *If you prefer, you can select individual user accounts instead of **Everyone**.*

5. Click the **Add ...** button.

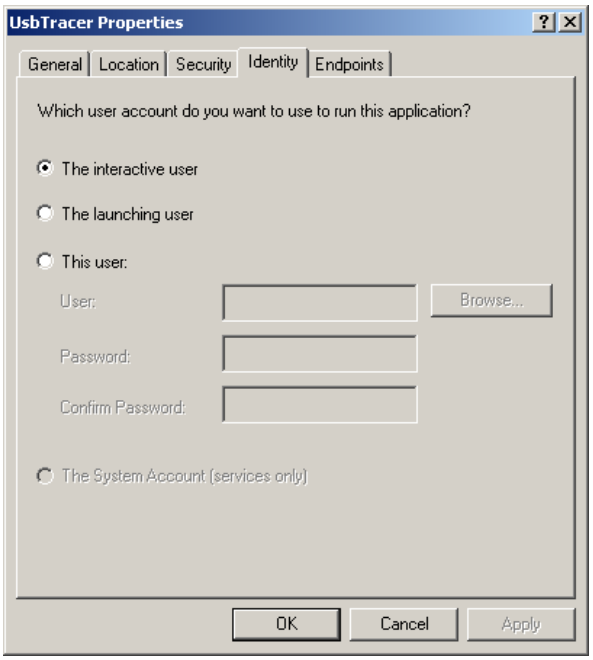
6. Click **OK**.
   The Add Users and Groups dialog box closes. The Registry Key Permissions dialog box remains on the screen.

7. Click **OK**.
   The Registry Key Permissions dialog box closes. The Properties dialog box remains on the screen.

Set User Run Permissions for Host Controller

If you want to create password-based security for individual users, perform the following steps on the Host Controller:

1. From the **Properties** dialog box, select the **Identity** tab.
   The following screen displays:



2. Make sure that the **Interactive User** option is selected.

A.4. DCOM Client Configuration

DCOM client configuration is a four-step process:

1. Set authentication to occur upon connection with the Host Controller.

2. Set the Host name or IP address of the server to which the client will connect.

3. Set commands to run on the Host Controller.

4. Set the network protocol over which DCOM will run.

Setting When and How Authentication Should Occur

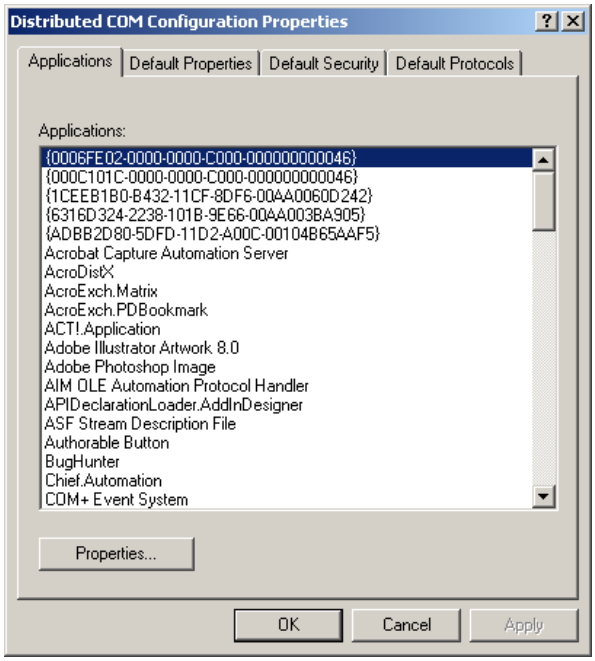
You specify when and how the client should be authenticated by performing the following steps:

1. Click the Windows **Start** button.

2. Select **Run ...**
   The Run dialog appears.

3. In the Open edit box within the Run dialog, type **dcomcnfg.**

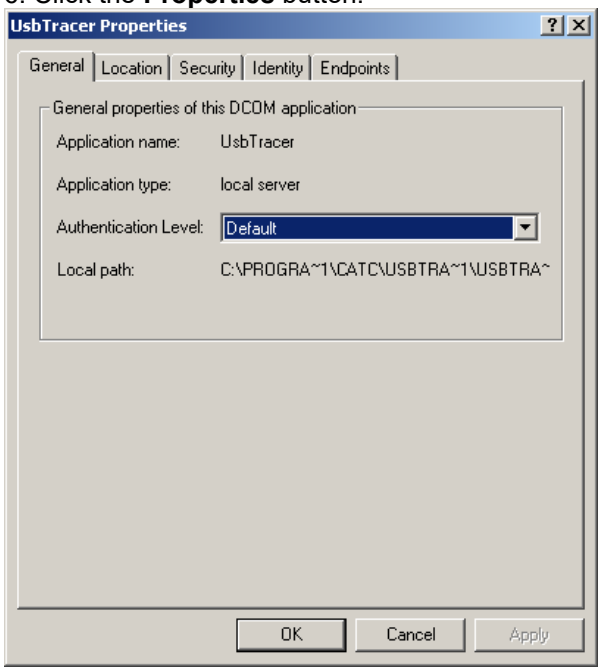
4. Click **OK**.
   The following dialog box appears:



This dialog box presents a list of applications on the PC that support DCOM.

5. Select **USBTracer/Trainer, USB Advisor**, or **Voyager (USB Protocol Suite)** from the list of applications.

6. Click the **Properties** button.



7. Click the **Authentication** drop-down menu and select **Default**.

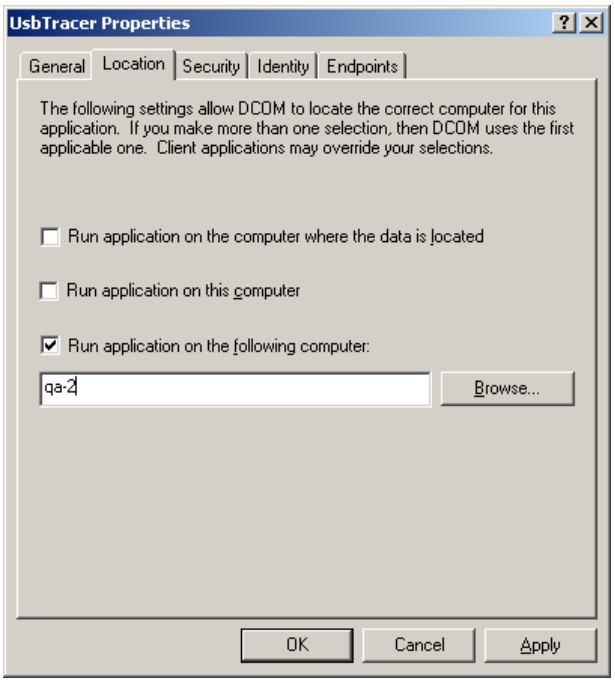
Default *tells the client to connect to the Host Controller using the Host Controller's Authentication Level.*

Set the Host Name or IP Address of the Host Controller

You must identify the device upon which the client will execute its commands.

Note: If you are writing a C/C++ program for automation, you can skip the following steps because you will specify the remote machines programmatically.

1. With the **Properties** dialog box still open, click the **Location** tab.
   The Location window displays.



2. Select the option **Run application on the following computer**.

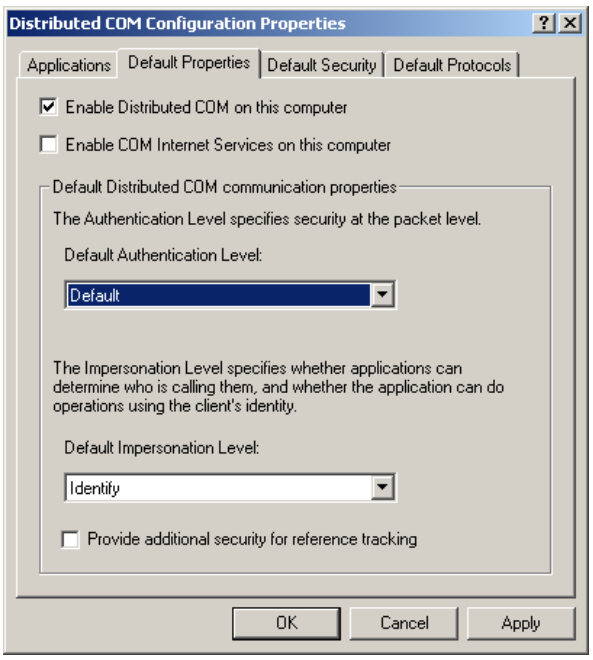
3. In the text box below the selected option, enter the **Host Name or IP address** of the Host Controller.
   *If you do not know the name of the Host Controller, you can browse to it using the **Browse ...** button.*

4. Click **OK**.
   The Properties dialog box closes. You see the Distributed COM Configuration Properties dialog box.

Set The Analyzer Commands to Execute on the Host Controller

The last configuration step is to set the "launch" location to **Server** for commands. This setting tells the client to execute commands remotely on the DCOM server rather than on itself.

1. On the Distributed Configuration Properties dialog box, click the tab marked **Default Properties**.
   You should see the following settings. If the settings are not the same as those shown, change them to match the screenshot.



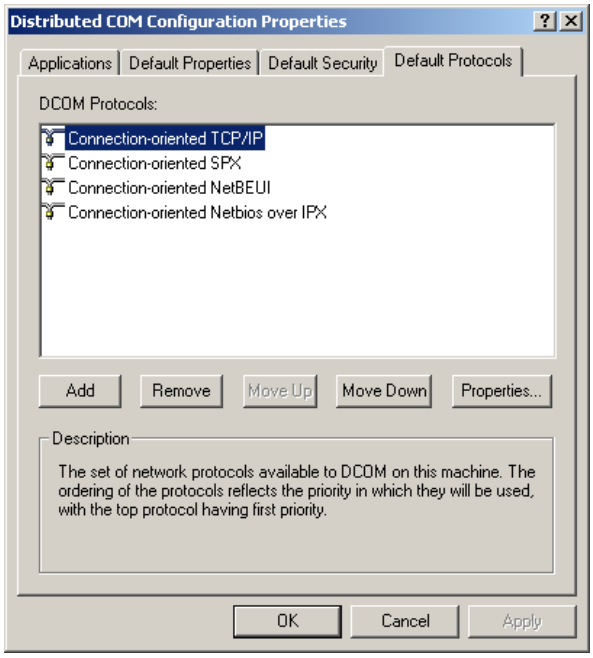
2. Select the option **Enable Distributed COM on this computer.**

You must perform this step on both computers - the DCOM server and DCOM client.

Set the Default Network Protocol

1. In the Distributed COM Configuration Properties dialog box, click the **Default Protocols** tab.

   *The following screen displays. TCP/IP should appear at the top of the list. If it does not, use the **Move Up** button to move it to the top of the list. If the protocol does not appear at all, you must add it using the **Add** button.*



2. Click **OK** to close the Distributed COM Configuration Properties dialog box.

   Your PC is now configured to run USBTracer/Trainer, USB Advisor, and Voyager USB Protocol Suite Automation.

A.5. Samples for Automation

You can test your Automation setup by running a sample script file or executing one of the simple client applications included in the software installation in the following path (typical installation):

C:\Users\Public\Documents\LeCroy\USB Protocol Suite\Automation

. The folder contains 4 directories:

- **cpp** - Contains C++ source code files. This directory includes the source code for an application called **analyzerclient.exe**. **Analyzerclient.exe** is a simple interface to the Automation API. **Analyzerclient.exe** lets you manage the Analyzer.

- **cpp_command_line** – Similar to above, but implements from the command line instead of a GUI.

- **html** - Contains a directory with an HTML-based client application called **analyzer1.html** that is a simple interface to the Automation API.

- **wsh** - This is a directory containing Visual Basic script files. You can run these script files by double-clicking on them.

To test your Automation setup, execute **AnalyzerClient.exe** (after compiling it first) or **analzyer1.html**, or double-click one of the **wsh** files. If your setup is correct, execute a Teledyne LeCroy sample script on the client PC that connects to the Host Controller. Sample script files can be found in the Automation folder mentioned above.

Testing Your Automation Setup with AnalyzerClient.exe

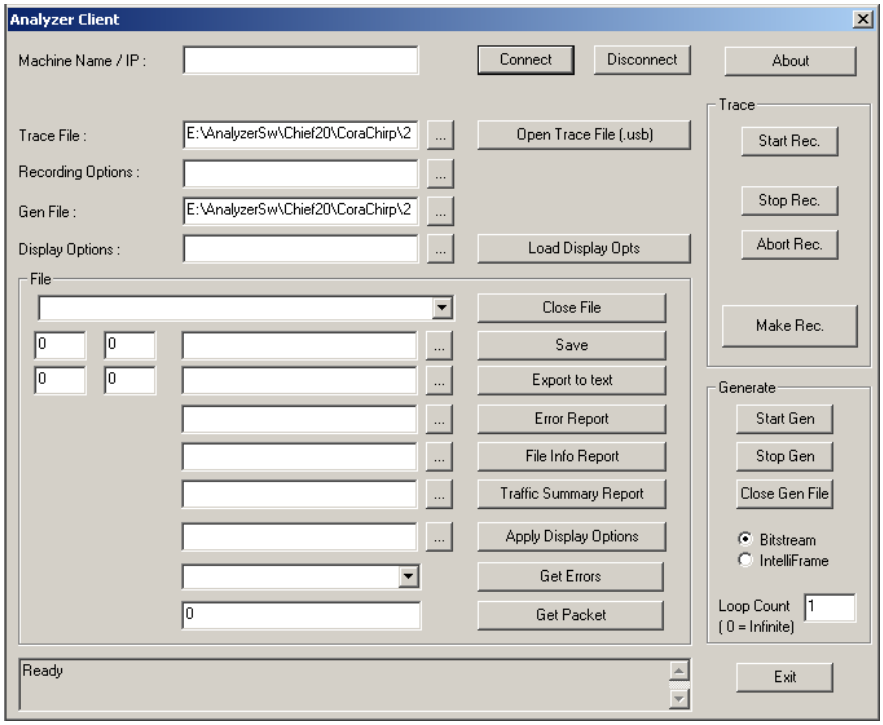
The following steps show you how to test your automation setup using **AnalyzerClient.exe**. As mentioned above, you must compile this first from the source code in the **cpp** directory.

In the steps below, you execute **analyzerclient.exe** on a remote PC and establish a connection with the Host Controller PC. Then, you remotely load a trace file on the Host Controller.

1. Browse to the directory in which you compiled **AnalyzerClient.exe**.

2. Double-click **AnalyzerClient.exe**.

The application starts. If you prefer, you can copy the executable to your hard drive and launch it from there.



3. Enter the **IP address or Host Name** of the Host Controller.

4. Press **Connect**.

The status message at the bottom of the window should change from **Ready** to a message that reads something like **USBTracer: Version 1.70**. This change indicates that a connection has been established to the Host Controller.

When you have completed this step, the Interface software automatically launches on the Host Controller.

If the message "The RPC server is unavailable" appears, check the network connections between the two PCs. Also try using the IP address in place of the Host Name (assuming that you had originally used the Host Name).

If the message "Access denied" appears, make sure that the account that you are using on the Host Controller is the same as the one you are using on the client PC and that both have Administrator privileges.

5. Click the  button next to the **Trace File** text box. You are going to browse for a trace file. If you are going to run the commands over a network, you must supply the network path.

6. Browse for a trace file of your choice. If you do not have a trace file on your PC, you can select one from an Installation diskette. When you have selected a file, its name should appear in the **Trace File** text box.

7. Press the button marked **Open File**. A message should appear at the bottom of the window listing the trace name, file size (in packets) and trigger position.

Appendix B.
How to Contact Teledyne LeCroy

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