PROFILE TUNING SUITE (PTS)

EXTENDED AUTOMATING – USING ETS MANAGER API

Disclaimer and Copyright Notice

Table of Contents

1 PTS Terminology 4
2 Automating PTS 5
2.1 "Lightweight Automated Operation" 5
3 General API usage 7
4 Functions in the ETS Manager API 8
4.1 Initialization 8
4.1.1 InitGetDevInfoWithCallbacks() 8
4.1.2 InitEtsEx () 8
4.1.3 ReinitEtsEx () 9
4.1.4 RegisterProfileWithCallbacks () 9
4.1.5 InitStackEx () 11
4.2 Dongle Address and Devices 11
4.2.1 VerifyDongleEx() 11
4.2.2 GetDongleBDAddress () 11
4.2.3 StartDeviceSearchEx () 12
4.2.4 StopDeviceSearchEx () 12
4.2.5 GetDongleDeviceInformation () 12
4.2.6 GetDeviceList () 13
4.2.7 SetPTSDevice () 13
4.3 Setting ICS and IXIT 13
4.3.1 SetParameterEx () 13
4.4 Running Test Cases 14
4.4.1 SetPostLoggingEx() 14
4.4.2 StartTestCaseEx() 14
4.4.3 StopTestCase() 15
4.4.4 TestCaseFinishedEx() 15
4.5 Clean-up APIs 16
4.5.1 ExitStackEx() 16
4.5.2 UnregisterProfileEx() 16
4.5.3 UnRegisterGetDevInfoEx() 16
4.6 Working with Bluetooth Protocol Viewer 17
4.6.1 SnifferInitializeEx() 17
4.6.2 SnifferRegisterNotificationEx() 17
4.6.3 SnifferClearEx() 17
4.6.4 SnifferIsRunningEx() 18
4.6.5 SnifferCanSaveEx() 18
4.6.6 SnifferCanSaveAndClearEx() 18
4.6.7 SnifferSaveEx() 18
4.6.8 SnifferSaveAndClearEx() 18
4.6.9 SnifferLogVerdictDescriptionEx () 19
4.6.10 SnifferIsConnectedEx() 19
4.6.11 SnifferCanClearEx () 19
4.6.12 SnifferTerminateEx () 19
4.6.13 InitSniffer () 20

1 PTS Terminology

- **IUT (Implementation Under Test)**: The device, component or subsystem to be tested.
- **Workspace**: A group of profile and protocol test suites to be tested against the Implementation Under Test. A workspace may be thought of as representing a particular device, component or subsystem.
- **Project**: A profile or protocol test suite and its associated data files. One or more projects may be present in a workspace. Each project represents a profile or protocol supported by the IUT.
- **ICS (Implementation Conformance Statement)**: Official declaration of the profile or protocol features and functions that are supported by the IUT. Each item in the ICS selects one or more tests that must be executed in order to demonstrate proper implementation.
- **IXIT (Implementation Extra Information for Testing)**: Data items, such as the *Bluetooth* Device Address (BD_ADDR), that are specific to a particular IUT. In general, IXIT items represent data that cannot be specified in advance by the programmer who created a test case or test suite.
- **ETS (Executable Test Suite)**: Each profile or protocol specified for use in *Bluetooth* wireless technology has an accompanying test specification. An ETS is a programmatic representation of the test purposes found in a particular test specification. Test cases in an ETS are executed under the control of the Profile Tuning Suite.
- **Test Purposes vs. Test Cases**: A test specification defines a number of test purposes which describe the environment that must be present to perform a test of a particular feature or function, the proper procedure to perform a test, and the expected outcome of a test.

  A test case is specific implementation of a test purpose, for example, a test case found in a PTS Executable Test Suite.

- **Test Case Naming**: Each test purpose defined in a test specification is identified by a name which is created according to a standard policy. The name identifies which groups of tests a particular test belongs to along with the nature of the test. Test purpose names are in a format similar to

2 Automating PTS

The Profile Tuning Suite (PTS) offers four features which can be used in automated testing:

1. "Operator-less Operation" allows the interactive prompts that appear during the execution of a test case to be processed by user written software which can inspect each message and take appropriate action.

   This feature can be used with either of the following program control features and is described in the "Automating – Using Implicit Send" reference document.

2. "Scripted Operation" where a set of test cases can be selected and run as a group. The group can be executed as needed, or scheduled for execution at a later time.

   The "Scripting" reference document describes this feature.

3. "Fully Automated Operation" – PTS provides an Application Programming Interface (API) which allows complete control of the software. User written programs can take advantage of the API in order to open Workspaces, select Projects, execute Test Cases, and many other functions.

   "Extended Automating – Using PTSControl API" document describes "Fully Automated Operation".

4. "Lightweight Automated Operation" – ETS Manager API provides another way of executing PTS test cases in a "fully automated" manner. It exposes test-implementing DLLs to automation clients in the same way as they are exposed to the PTS User Interface.

This document describes "Lightweight Automated Operation".

2.1 "Lightweight Automated Operation"

Completely unattended execution of PTS tests can be achieved by using the ETS Manager API. There are three parts to "Lightweight Automated Operation":

1. ETSManager.dll: Defines generic interface to interact with test-implementing DLLs, exposes test-implementing DLL functionality to client applications including PTS.exe;

2. Client Application: An application program written by a PTS user that makes use of the ETS Manager API in order to run PTS test cases and collect test execution results. The ETS Manager API portion of a Client Application may be part of a larger program that interfaces to a test system platform that has the capability of also controlling the *Bluetooth* device that is being tested;

3. ETS DLLs: Collection of DLLs that implement Bluetooth qualification tests.

The ETSManager.dll exposes standard C DLL interface that can be consumed by any Windows-based programming language including scripting languages such as Python.

This document contains a description of all ETSManager.dll API methods that are exposed to the clients.

In addition, a sample client script – ETSPythonClient.py – is also provided as part of the PTS installation. This script, written in Python 3.5, exercises many of the functions available in the API. It can be a valuable reference when developing your own client scripts.

ETSPythonClient.py is located in the

> C:\ Program Files [(x86)]\Bluetooth SIG\Bluetooth PTS\SampleCode\ETSManager Script Client folder in the PTS installation.

3 General API usage

The following points should be kept in mind while using the ETS Manager API:

- Since ETS Manager exposes C DLL interface, all API call parameters must be passed using C compatible data types.
- Before ETSManager.dll is loaded, it is helpful to set current working directory to <Your PTS installation directory>\ Bluetooth PTS \bin to avoid DLL dependency issues.
- String values must be passed to API calls as byte literals.

4 Functions in the ETS Manager API

4.1 Initialization

The first step in using the ETS Manger API is to call various initialization APIs to set required callbacks and prepare the profile and Host Stack DLLs.

4.1.1 InitGetDevInfoWithCallbacks()

Declaration: __declspec(dllexport) bool InitGetDevInfoWithCallbacks(const char * pchExeInstallDir, LPDEVICESEARCH devSearchCallback, LPDONGLEMSG dongleMsgCallback);

Parameters: pchExeInstallDir: ASCII character array specifying PTS.exe installation directory.

devSearchCallback: Device search call back function pointer. The call back function will be executed every time new Bluetooth device data becomes available.

The device search call back function must have the following signature:
bool (_stdcall *LPDEVICESEARCH)(const char* pchAddr, const char* pchName, const char* pchCod);

Please note that the device search call back may be called multiple times for the same device. Although every call back invocation will contain device address data, the device name and cod may be absent.

dongleMsgCallback: Dongle message call back function pointer. The call back function will be executed whenever a new message from the PTS dongle is received. The dongle message call back function must have the following signature:
bool (_stdcall *LPDONGLEMSG)(const char* pchMsg);

It is expected that the call back will be called with the following message:
'SNIFFER/Save and clear complete'
Whenever Bluetooth Protocol Viewer completes Save or Clear Data requests.

Return values: true if successful, false otherwise.

4.1.2 InitEtsEx ()

Declaration: __declspec(dllexport) bool InitEtsEx(const char* pchProfile, const char* pchWorkspacePath, const char* pchImplicitSendDllPath, const char* pchPtsDongleAddress);

Parameters: pchProfile: ASCII character array specifying selected profile name.

It is necessary to call this API before running any selected profile tests.

pchWorkspacePath: ASCII character array specifying selected workspace directory.

Although having a PTS workspace is not required to use ETS Manager APIs, some of the profiles require supplementary files to execute tests. It is expected that these files are located inside of the workspace directory

pchImplicitSendDllPath: ASCII character array specifying path to Implicit Send DLL.

Even if Implicit Send DLL is not used in association with automated test execution, it is recommended to provide path to a valid Implicit Send DLL. The path to the default Implicit Send DLL located in PTS's bin directory is acceptable.

pchPtsDongleAddress: ASCII character array specifying PTS Dongle BD address.

The PTS Dongle BD address can be found using GetDongleBDAddress() API (see description below).

Return values: true if successful, false otherwise.

4.1.3 ReinitEtsEx ()

Declaration: __declspec(dllexport) bool ReinitEtsEx (const char* pchProfile);

Parameters: pchProfile: ASCII character array specifying selected profile name.

It is necessary to call this API after modifying an ICS or IXIT value of a selected profile if InitEtsEx() has already been called for this profile.

Return values: true if successful, false otherwise.

4.1.4 RegisterProfileWithCallbacks ()

Declaration: __declspec(dllexport) bool RegisterProfileWithCallbacks(const char* pchProfileName,
LPUSEAUTOIMPLICITSEND useAutoImplicitSendCallback,
LPAUTOIMPLICITSEND autoImplicitSendCallback, LPLOG logCallback,
LPDEVICESEARCH devSearchCallback, LPDONGLEMSG dongleMsgCallback);

Parameters: pchProfileName: ASCII character array specifying selected profile.

It is necessary to call this API before running any selected profile tests.

useAutoImplicitSendCallback: UseAutoImplicitSend call back function pointer.

The UseAutoImplicitSend function must have the following signature:
bool (_cdecl *LPUSEAUTOIMPLICITSEND)(void);

This function must return true if the automation client implements ImplicitSend call back. If false is returned, it is expected that Implicit Send calls will be handled by the DLL specified by InitEtsEx() API.

autoImplicitSendCallback: ImplicitSend call back function pointer. The call back function will be executed every time a new MMI message is available.

This call back function must have the following signature:
char *(_cdecl *LPAUTOIMPLICITSEND)(char * description, UINT style);

If UseAutoImplicitSend returns true this function must be implemented. The expected return value is defined by the ImplicitSend message style:

Style name	Value	Message Type	Expected return values
MMI_Style_Ok_Cancel1	0x11041	Simple prompt	OK, Cancel
MMI_Style_Ok_Cancel2*	0x11141	Simple prompt	OK, Cancel
MMI_Style_Ok	0x11040	Simple prompt	OK
MMI_Style_Yes_No1	0x11044	Simple prompt	Yes, No
MMI_Style_Yes_No_Cancel1	0x11043	Simple prompt	Yes, No, Cancel
MMI_Style_Abort_Retry1	0x11042	Simple prompt	Abort, Retry, Ignore
MMI_Style_Edit1	0x12040	Request for data input	String that would be entered into corresponding edit box during manual test execution
MMI_Style_Edit2	0x12140	Select item from a list	String that would be selected in a list during manual test execution

The description parameter has the following format: {MMI_ID,Test Name,Layer Name}MMI Action\n\nDescription: MMI Description.

logCallback: Log call back function pointer. The call back function will be executed every time new log data becomes available.

The log call back function must have the following signature:
bool (_stdcall *LPLOG)(const char* pchLogTime, const char* pchLogDescription, const char* pchLogMessage, int nLogType, void* pProject);

The following is the list of potential log types:
LOG_TYPE_GENERAL_TEXT = 0
LOG_TYPE_START_TEST_CASE = 1
LOG_TYPE_TEST_CASE_ENDED = 2
LOG_TYPE_START_DEFAULT = 3
LOG_TYPE_DEFAULT_ENDED = 4
LOG_TYPE_FINAL_VERDICT = 5
LOG_TYPE_PRELIMINARY_VERDICT = 6
LOG_TYPE_TIMEOUT = 7
LOG_TYPE_ASSIGNMENT = 8
LOG_TYPE_START_TIMER = 9
LOG_TYPE_STOP_TIMER = 10
LOG_TYPE_CANCEL_TIMER = 11
LOG_TYPE_READ_TIMER = 12
LOG_TYPE_ATTACH = 13
LOG_TYPE_IMPLICIT_SEND = 14
LOG_TYPE_GOTO = 15
LOG_TYPE_TIMED_OUT_TIMER = 16
LOG_TYPE_ERROR = 17
LOG_TYPE_CREATE = 18
LOG_TYPE_DONE = 19
LOG_TYPE_ACTIVATE = 20
LOG_TYPE_MESSAGE = 21
LOG_TYPE_LINE_MATCHED = 22
LOG_TYPE_LINE_NOT_MATCHED = 23
LOG_TYPE_SEND_EVENT = 24
LOG_TYPE_RECEIVE_EVENT = 25
LOG_TYPE_OTHERWISE_EVENT = 26
LOG_TYPE_RECEIVED_ON_PCO = 27
LOG_TYPE_MATCH_FAILED = 28
LOG_TYPE_COORDINATION_MESSAGE = 29.

LOG_TYPE_FINAL_VERDICT message should be used to obtain the executed test verdict.
Call back should return true in case of success, false otherwise.

devSearchCallback: Device search call back function pointer. The call back function will be executed every time new Bluetooth device data becomes available.

The device search call back function must have the following signature:
bool (_stdcall *LPDEVICESEARCH)(const char* pchAddr, const char* pchName, const char* pchCod);

Please note that the device search call back may be called multiple times for the same device. Although every call back invocation will contain device address data, the device name and cod may be absent.

dongleMsgCallback: Dongle message call back function pointer. The call back function will be executed whenever a new message from the PTS dongle is received.
Dongle message call back function must have the following signature:
bool (_stdcall *LPDONGLEMSG)(const char* pchMsg);

It is expected that the call back will be called with the following message:
'SNIFFER/Save and clear complete'
Whenever Bluetooth Protocol Viewer completes Save or Clear Data requests.

Return values: true if successful, false otherwise.

4.1.5 InitStackEx ()

Declaration: __declspec(dllexport) bool InitStackEx(const char* pchProfileName);

Parameters: pchProfileName: ASCII character array specifying selected profile name.

It is necessary to call this API every time before running a group of tests from the selected profile.

Return values: true if successful, false otherwise.

4.2 Dongle Address and Devices

This group of API calls facilitates PTS dongle BD address discovery and detection of nearby Bluetooth devices.

4.2.1 VerifyDongleEx()

Declaration: __declspec(dllexport) bool VerifyDongleEx(void);

Parameters: none.

Return values: true if PTS dongle is available and ready to communicate, false otherwise.

4.2.2 GetDongleBDAddress ()

Declaration: __declspec(dllexport) long GetDongleBDAddress(void);

Parameters: none.

Return values: PTS dongle BD address as hexadecimal value.

4.2.3 StartDeviceSearchEx ()

Declaration: __declspec(dllexport) bool StartDeviceSearchEx(const char* pchFilter, const char* pchMask, const char* pchProfileName);

Parameters: pchFilter: The following table contains possible filter values:

Device class	Filter Value
Any	"0"
Audio/Video	"4"
Computer	"1"
Health	"9"
Imaging	"6"
LAN	"3"
Peripheral	"5"
Phone	"2"
Toy	"8"
Wearable	"7"

pchMask: The following table contains mask values:

Service class	Filter Value
None	"0"
Transfer	"16"
Audio	"32"
Capturing	"8"
Information	"128"
Limited Discoverability	"32"
Networking	"2"
Positioning	"1"
Rendering	"4"
Telephony	"64"

pchProfileName: ASCII character array, should be set to 'GetDevInfo'.

Return values: true if successful, false otherwise.

4.2.4 StopDeviceSearchEx ()

Declaration: __declspec(dllexport) bool StopDeviceSearchEx(const char* pchProfileName);

Parameters: pchProfileName: ASCII character array, should be set to 'GetDevInfo'.

Return values: true if successful, false otherwise.

4.2.5 GetDongleDeviceInformation ()

Declaration: __declspec(dllexport) void GetDongleDeviceInformation (void);

Parameters: None.

Return values: void. Dongle device information such as firmware version will be received through dongleMsgCallback (see Section 4.1.4) and identified by the "GetDeviceInformation:" prefix.

4.2.6 GetDeviceList ()

Declaration: __declspec(dllexport) char* GetDeviceList (void);

Parameters: none.

Return values: ASCII character array that contains a ';'-separated list of names of radio devices that can be used as PTS endpoint devices. The names can be in one of the following formats:

- USB:InUse:<Unique ID> - BR/EDR/LE PTS dongle that is free and available for connection
- USB:Free:<Unique ID> - BR/EDR/LE PTS dongle that is in use by another instance of PTS and not available for connection
- COM<Unique ID> - LE-only PTS dongle that is available for connection

4.2.7 SetPTSDevice ()

Declaration: __declspec(dllexport) void SetPTSDevice (const char* pchDeviceName);

Parameters: pchDeviceName: ASCII character array that contains the name of a radio device to be used as the PTS endpoint device. When connecting to a 'USB' device the name should correspond to last ':'-separated section of the string returned by GetDeviceList() call (see <Unique ID> in Section 4.2.6). Use the string returned by GetDeviceList() as-is to connect to a 'COM' device.

Return values: void.

4.3 Setting ICS and IXIT

The following API call facilitates setting required test parameters before execution.

4.3.1 SetParameterEx ()

Declaration: __declspec(dllexport) bool SetParameterEx(const char* pchParameterName, const char* pchParameterType, const char* pchParameterValue, const char* pchProfileName);

Parameters: pchParameterName: ASCII character array that contains the name of an ICS/IXIT item defined in the selected profile that is to be updated.

The ICS/IXIT item name is case sensitive and must match the name shown in the ICS/IXIT editor tool window of the PTS User Interface.

pchParameterType: ASCII character array that contains the type of an ICS/IXIT item defined in the selected profile that is to be updated. The following are the supported types and their corresponding value sets:

Data Type	**Legal Characters For The New Value**
"BITSTRING"	ASCII character arrays consisting of 0 and 1
"BOOLEAN"	"TRUE" or "FALSE" (case sensitive)
"IA5STRING"	ASCII character arrays

"INTEGER"	ASCII character arrays consisting of digits 0 through 9
"OCTETSTRING"	ASCII character arrays consisting of digits "digits" 0 to 9 and characters A through F ("A" to "F" must be upper case)

The ICS/IXIT type is case sensitive and must match the type name shown in the IXIT editor tool window of the PTS User Interface.

pchParameterValue: ASCII character array that contains the new value to be assigned to the specified ICS/IXIT item.

See the table above for restrictions on the contents of this string.

pchProfileName: ASCII character array specifying selected profile name.

Return values: true if successful, false otherwise.

4.4 Running Test Cases

This group of APIs facilitates PTS test execution.

4.4.1 SetPostLoggingEx()

Declaration: __declspec(dllexport) void SetPostLoggingEx(bool bPostLogging, const char* pchProfileName);

Parameters: bPostLogging: A Boolean specifying whether post-login should be turned off or on.

If post-logging is turned on, all callbacks to LPLOG-type function will be executed after test execution is complete. It is recommended that post-logging is turned on to avoid simultaneous invocations of LPLOG and LPAUTOIMPLICITSEND callbacks.

pchProfileName: ASCII character array specifying selected profile name.

Return values: void.

4.4.2 StartTestCaseEx()

Declaration: __declspec(dllexport) bool StartTestCaseEx(const char* pchTestCaseName, const char* pchProfileName, bool bEnableMaxLog);

Parameters: pchTestCaseName: ASCII character array specifying selected test case to execute.

Supported test case names must correspond to the current TCRL version and should be the same as presented in the PTSUser Interface.

pchProfileName: ASCII character array specifying selected profile name.

bEnableMaxLog: A Boolean specifying whether PTS should maximize the amount of produced log information.

Turning specific elements of logging on and off is normally done through pts.ini file located in C:\Users\<user_name>\AppData\Roaming\Bluetooth_SIG\ProfileTuningSuite_6. Setting bEnableMaxLog to true would be similar to enabling all selections in pts.ini or executing PTS test cases using Run (Debug Logs) GUI command.

Return values: true if successful, false otherwise.

StartTestCaseEx () API returns immediately after test case execution has been started.

4.4.3 StopTestCase()

Declaration: __declspec(dllexport) bool StopTestCaseEx(const char* pchTestCaseName, const char* pchProfileName);

Parameters: pchTestCaseName: ASCII character array specifying selected test case to execute.

pchProfileName: ASCII character array specifying selected profile name.

Return values: true if successful, false otherwise.

Use this API to stop the currently executing test case if it does not complete within the expected time period.

4.4.4 TestCaseFinishedEx()

Declaration: __declspec(dllexport) bool TestCaseFinishedEx(const char* pchTestCaseName, const char* pchProfileName);

Parameters: pchTestCaseName: ASCII character array specifying selected test case to execute.

pchProfileName: ASCII character array specifying selected profile name.

Return values: true if successful, false otherwise.

This API must be executed after the running test case completes. It performs required “after test case” clean up.

4.5 Clean-up APIs

The APIs in this group should be executed before completing the automated test session to release allocated resources.

4.5.1 ExitStackEx()

Declaration: __declspec(dllexport) bool ExitStackEx(char* pchProfileName);

Parameters: pchProfileName: ASCII character array specifying selected profile name.

Return values: true if successful, false otherwise.

This API releases resources allocated by InitStackEx() API call.

4.5.2 UnregisterProfileEx()

Declaration: __declspec(dllexport) bool UnregisterProfileEx(const char* pchProfileName);

Parameters: pchProfileName: ASCII character array specifying selected profile name.

Return values: true if successful, false otherwise.

This API releases resources allocated by RegisterProfileWithCallbacks () API call.

4.5.3 UnRegisterGetDevInfoEx()

Declaration: __declspec(dllexport) bool UnRegisterGetDevInfoEx(void);

Parameters: void.

Return values: true if successful, false otherwise.

This API releases resources allocated by InitGetDevInfoWithCallbacks () API call.

4.6 Working with Bluetooth Protocol Viewer

API calls in this group configure and control Bluetooth Protocol Viewer operation.

Bluetooth Protocol Viewer is a desktop application. In order to take advantage of Bluetooth Protocol Viewer during automated test session it must be started using Windows API facilities provided by the operating system. If properly initialized ETSManager will find the running instance of Bluetooth Protocol Viewer and direct Bluetooth traffic into this application for capturing and parsing.

4.6.1 SnifferInitializeEx()

Declaration: __declspec(dllexport) bool SnifferInitializeEx(void);

Parameters: void.

Return values: true if successful, false otherwise.

This API releases the initialized subsystem responsible for communicating with the Bluetooth Protocol Viewer application.

4.6.2 SnifferRegisterNotificationEx()

Declaration: __declspec(dllexport) int SnifferRegisterNotificationEx(void);

Parameters: void.

Return values: integer identifying Bluetooth Protocol Viewer status:

Value	Numeric Value	Comments
SNIFFER_STATUS_OK	0	No error
SNIFFER_STATUS_NOT_AVAILABLE	1	Requested functionality is not available
SNIFFER_STATUS_NOT_READY	2	Requested functionality is available, but the API is not in the proper state to execute it
SNIFFER_STATUS_INSUFFICIENT_BUFFER	3	Size of buffer provided for data to be returned in is too small
SNIFFER_STATUS_INVALID_PARAM	4	A function parameter value is invalid
SNIFFER_STATUS_INTERNAL_ERROR	5	Error in the sniffer code
SNIFFER_STATUS_INITIALIZED	6	Sniffer has been initialized

This API must be invoked in order for LPDONGLEMSG to be called whenever Bluetooth Protocol Viewer sends status messages to its clients.

4.6.3 SnifferClearEx()

Declaration: __declspec(dllexport) int SnifferClearEx(void);

Parameters: void.

Return values: integer identifying Bluetooth Protocol Viewer status (see table above).

This API deletes previously collected data and re-initializes Bluetooth Protocol Viewer.

4.6.4 SnifferIsRunningEx()

Declaration: __declspec(dllexport) bool SnifferIsRunningEx(void);

Parameters: void.

Return values: true if Bluetooth Protocol Viewer is active and running, false otherwise.

4.6.5 SnifferCanSaveEx()

Declaration: __declspec(dllexport) bool SnifferCanSaveEx (void);

Parameters: void.

Return values: true if installed Bluetooth Protocol Viewer version supports SnifferSaveEx() API, false otherwise.

4.6.6 SnifferCanSaveAndClearEx()

Declaration: __declspec(dllexport) bool SnifferCanSaveAndClearEx (void);

Parameters: void.

Return values: true if installed Bluetooth Protocol Viewer version supports SnifferSaveAndClearEx() API, false otherwise.

4.6.7 SnifferSaveEx()

Declaration: __declspec(dllexport) int SnifferSaveEx(const char* pchSavePath);

Parameters: pchSavePath: ASCII character array specifying path to a file where data collected by Bluetooth Protocol Viewer should be saved. The standard file extension for Bluetooth Protocol Viewer file is .cfa.

Return values: integer identifying Bluetooth Protocol Viewer status (see table above).

Call this API to save collected data to a file.

4.6.8 SnifferSaveAndClearEx()

Declaration: __declspec(dllexport) int SnifferSaveAndClearEx(const char* pchSavePath);

Parameters: pchSavePath: ASCII character array specifying path to a file where data collected by Bluetooth Protocol Viewer should be saved.

Return values: integer identifying Bluetooth Protocol Viewer status (see table above).

Call this API to save collected data to a file and then clear collected data.

4.6.9 SnifferLogVerdictDescriptionEx ()

Declaration: __declspec(dllexport) int SnifferLogVerdictDescriptionEx(const char* pchLogString, int nVerdictType, DWORD msSinceTestCaseStart);

Parameters: pchLogString: ASCII character array specifying string to add to Bluetooth Protocol Viewer output.

nVerdictType: integer specifying the type of the string to be added:

Value	Numeric Value	Comments
SNIFFER_VERDICT_PASS	0	Pass
SNIFFER_VERDICT_INCONCLUSIVE	1	Inconclusive
SNIFFER_VERDICT_FAIL	2	Fail
SNIFFER_VERDICT_NONE	3	None
SNIFFER_VERDICT_COMMENT	4	Commentary about the test case execution. This is not a verdict, but can be used to provide additional information about a verdict decision.

msSinceTestCaseStart: double word integer specifying the number of milliseconds elapsed since the start of test case:

Return values: integer identifying Bluetooth Protocol Viewer status (see table above).

Use this API to add descriptive test execution status information when necessary.

4.6.10 SnifferIsConnectedEx()

Declaration: __declspec(dllexport) bool SnifferIsConnectedEx(void);

Parameters: void.

Return values: true if running Bluetooth Protocol Viewer has been found and connected to.

4.6.11 SnifferCanClearEx ()

Declaration: __declspec(dllexport) bool SnifferCanClearEx (void);

Parameters: void.

Return values: true if Bluetooth Protocol Viewer is running, connected and supports SnifferClearEx() API, false otherwise.

4.6.12 SnifferTerminateEx ()

Declaration: __declspec(dllexport) int SnifferTerminateEx (void);

Parameters: void.

Return values: integer identifying Bluetooth Protocol Viewer status (see table above).

Call this API to terminate the connection to the running instance of Bluetooth Protocol Viewer. Note that Bluetooth Protocol Viewer supports only one connection at any given time.

4.6.13 InitSniffer ()

Declaration: __declspec(dllexport) void InitSniffer (void);

Parameters: void.

Return values: none.

Call this API to initialize the running instance of Bluetooth Protocol Viewer to prepare for connecting PTS to a radio device. The API should be called after InitGetDevInfoWithCallbacks() (see Section 4.1.1)