PROFILE TUNING SUITE (PTS)

AUTOMATING - USING IMPLICIT SEND

Disclaimer and Copyright Notice

Table of Contents

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 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 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 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 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 many 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 test case belongs to along with the nature of the test. Test purpose names are in a format like

  <PROFILE>/<ROLE>/<FEATURE>/BV-01-I (example: A2DP/SNK/AS/BV-01-I)

  In the PTS, the naming format matches the test specification identification convention exactly.

2 Automating PTS

The Profile Tuning Suite (PTS) offers three 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 this document and in "Automating" section of the PTS Help.

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 later.

The "Scripting" section of the PTS Help 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.

   "Fully Automated Operation" is described in the "Extended Automating" section of the PTS Help and in …\Bluetooth SIG\Bluetooth PTS\SampleCode\PTSControlClient\Extended_Automating.pdf document.

2.1 "Operator-less Operation"

Many, if not most, of the *Bluetooth* qualification tests are designed around the idea that a human test operator is part of test environment – operating the tester and performing manual operations on the Implementation Under Test (IUT). This can be seen in the various test specifications in comments like

- *Expected Outcome*

  Pass verdict:

  The Object Push operation is processed correctly and completed corresponding to the settings and user actions.

  Client:

  - The Object Push function is initiated by user action and not automatically.[1]

Part of the reason for this is that the *Bluetooth* Special Interest Group (SIG) does not specify the ways in which users are to interact with *Bluetooth* enabled devices. This can also be seen in the test specifications

- *Test Procedure*

  …

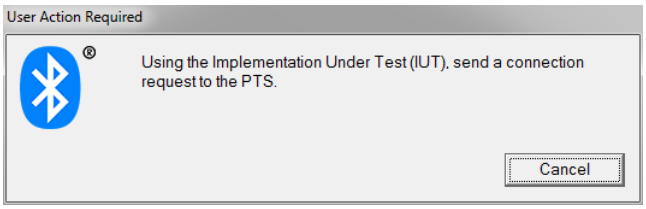
  Server:

  …

  - (Depending on the architecture that is to use the object push feature the steps how an item is pushed may vary).[1]

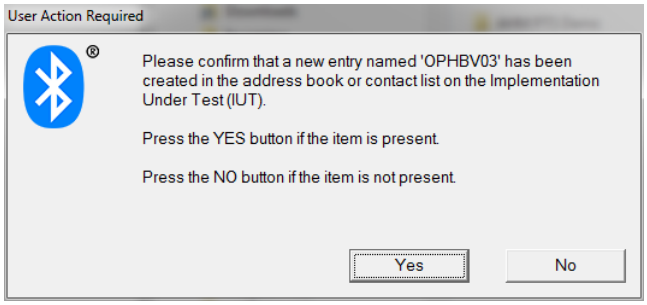
In its default configuration, the Profile Tuning Suite (PTS) presents the test operator with various popup dialogs during the execution of a test case. These dialogs may be used to

- Ask the test operator to perform a function on the IUT;

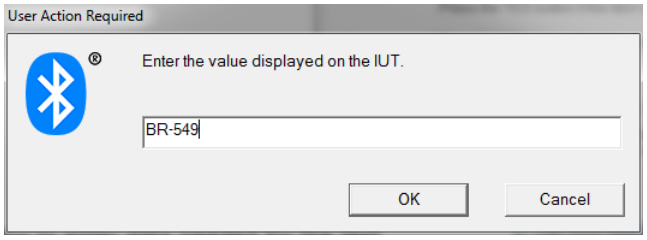


[1] Description of Test Purpose OPP/SR/OPH/BV-03-I, Object Push Profile Specification Test Suite Structure (TSS) and Test Purposes (TP), section 4.2.3. Document Number OPP.TS/1.2.1.

- Ask the test operator to confirm that a file transfer or other operation completed successfully;



- Ask the test operator to enter data needed for the test.



2.2 Automation test platforms

Having a test operator involved in the testing process can be very time consuming. Additionally, regression testing becomes somewhat difficult since a test operator needs to “babysit” the testing process. For this reason, many organizations create automation test platforms to be used in the testing of their devices. These platforms may have the ability to press buttons, recognize prompts and messages on the display, or access the storage on the device for contact items, pictures, or other files. These operations are controlled by software running on a computer that is connected to the test platform.

PTS can work with automation test platforms by providing a user defined mechanism that can be used to replace the popup dialogs mentioned above. Instead of sending the various messages to the display, PTS can be configured to send them to user written replacement functions that in turn can interact directly with the device being tested.

2.3 PTS test case operation

As mentioned above, there are various points during test case execution where PTS needs to interact with the test operator or an automation platform. When this occurs

1. The part of the test case that needs outside assistance sends a request to the "MMI Handler". PTS test cases are implemented using a Main Test Component (MTC) and some number of Parallel Test Components (PTCs). The PTCs provide various support functions and operate concurrently with the main body of the test. PTCs are often used to implement a protocol layer in the Bluetooth stack or to serve as the "knowledgeable authority" for the details of a Bluetooth profile.

   The MTC and the various PTCs interact with one another to process commands, responses and the transfer of data between themselves and the Implementation Under Test (IUT).

   The MMI PTC handles the interaction with the outside environment. Since it operates in parallel with the other parts of the test, test case execution is not held up while waiting for a response from the test operator.

2. After receiving a request, the MMI PTC passes it on to a support library known as an "Implicit Send DLL".

3. The Implicit Send DLL performs whatever steps are needed to execute the request and waits for a response.

4. The response is sent back up the chain to the MMI PTC, and from there to whatever Test Component is expecting it.

3 Implicit Send DLLs

PTS provides a default version of the Implicit Send DLL. It is this DLL that provides the popup dialogs that one normally sees when using the PTS.

To integrate PTS with an automation test platform, a custom Implicit Send DLL needs to be developed.

3.1 Basic information

- Implicit Send DLLs are standard Windows Dynamic Link Libraries.

- Implicit Send DLLs are written in C++.

  The interface between the PTS and an Implicit Send DLL uses the std::string class from the Standard Template Library (STL), and the "bool" data type. No other C++ features are used, so someone knowledgeable in the C programming language should not have too much trouble.

- An Implicit Send DLL provides five functions:

  - InitImplicitSend()
  - ImplicitSendStyle() / ImplicitSendStyleEx()
  - ImplicitSendPinCode() / ImplicitSendPinCodeEx()
  - ImplicitStartTestCase()
  - ImplicitTestCaseFinished()

  All five functions must be provided. If any one of the functions is missing, or has an incorrect name, PTS will be unable to load the DLL. The ImplicitSendStyleEx() and ImplicitSendPinCodeEx() take an extra parameter of Bluetooth address of PTS. If the Ex version of both functions is provided, PTS will ignore the non-Ex functions. Otherwise, both non-Ex functions must be provided. The functions are described in the following section (3.2, "Implicit Send functions").

- Implicit Send DLLs are loaded dynamically. They are standalone entities that do not need special names or folder locations for PTS to locate them. (A configuration setting tells PTS where to find the DLL.)

3.2 Implicit Send functions

3.2.1 Conventions

Each of the function definitions contains the following two declarations. These declarations must be used for the interface between PTS and an Implicit Send DLL to work correctly.

- *extern "C"* – This declaration tells the C++ compiler that the symbol name for a function is not to be decorated in any way. The C++ language allows multiple functions with the same name, if they have different parameter lists and/or return types. This is accomplished by changing – or "decorating" – the function names behind the scenes, resulting in each function having a different name.

  PTS expects the functions in an Implicit Send DLL to have plain, undecorated names.

- *WINAPI* – This declaration tells the C++ compiler that the function calling convention is the same as functions defined in the Windows API. This primarily has an impact on parameter handling during the calls from PTS to the Implicit Send functions.

3.2.2 InitImplicitSend()

Declaration: extern "C" bool WINAPI InitImplicitSend(void);

Parameters: None

Return values: "true" if successful
"false" if not successful

This function is called during the initialization of an Executable Test Suite (ETS), just after the Implicit Send DLL has been loaded into memory. It can be used to perform any initialization that might be needed before executing test cases.

If no initialization is required, the function can simply return a value of "true".

If the initializations failed, which would lead to the DLL not being usable, a "false" value should be returned. In this case, the ETS will be disabled.

3.2.3 **ImplicitStartTestCase()**

Declaration: extern "C" void WINAPI ImplicitStartTestCase(std::string& strTestCaseName);

Parameters: A character string containing the name of the current test case

Return values: None

ImplicitStartTestCase() is called at the start of each test case execution. It provides the name of the test case that is starting.

This function can be used to perform initializations that are needed at the start of every test case. The test case name allows the initialization process to be customized to specific test cases.

3.2.4 **ImplicitSendStyle() / ImplicitSendStyleEx()**

Declaration: extern "C" char* WINAPI ImplicitSendStyle(std::string& strMmiText,UINT mmiStyle);

Parameters: strMmiText – Information about the MMI (Implicit Send) request being made
mmiStyle – A value describing the type of request and the expected values
strBdAddr – (ImplicitSendStyleEx only) Bluetooth address of PTS

Return values: A pointer to a character string containing the information to be returned
A NULL pointer if the request cannot be processed or no information is to be returned

This is the main routine for handling Implicit Send requests; most of interactions with the test operator or automated test environment will be handled by this function.

strMmiText will consist of two pieces

- A message "tag" that uniquely identifies the message (see section 3.3, "Message tags").
- Message text that would normally be displayed to the test operator.

In the default, Implicit Send DLL used by PTS, the mmiStyle parameter is used to select the style of dialog box that is to be displayed. Custom DLLs can use this information to determine how to process the strMmiText, the type of request that is being made, and the expected return values.

In most cases, the return value will be a pointer to a string containing the word "OK". Some requests, such as those using MMI_Style_Edit1 expect a string of data – for example, a PIN code or a file name – as the return value.

The MMI_Style_Edit2 style provides a list of items in the strMmiText and expects one of those items to be returned.

For more information on the MMI styles see section 3.4, "MMI styles".

3.2.4.1 **Scope of the return value**

It is important to note that the character string that is pointed at by the ImplicitSendStyle() return value must not go "out of scope". For example, std::string values that are created during the execution of a function are likely to be destroyed when the function exits. The returned pointer may continue to point at valid text, but there is a good chance that the memory space used by the string could be reused.

One way to avoid this type of issue is to create the string to be returned in dynamic memory (using malloc() or "new"). The string would then be added to a list or variable sized array (such as a std::vector). All of strings returned during the execution of the test case would remain until the end of the test, at which time they could be destroyed.

For an example of one way to do this, look in sample source code (see section 5.4, "Sample source code") at the PersistentText C++ class (PersistentText.cpp/.h) and how the class is used in ImplicitSend.cpp. In this example, the most recently returned string is "persistent" and is deleted when the Implicit Send DLL is unloaded.

3.2.5 **ImplicitSendPinCode() / ImplicitSendPinCodeEx()**

Declaration: extern "C" char* WINAPI ImplicitSendPinCode(void);

Parameters: strBdAddr – (ImplicitSendPinCodeEx only) Bluetooth address of PTS

Return values: A pointer to a character string containing a PIN code to be returned
A NULL pointer if the request cannot be processed or no PIN code is to be returned

This is a special case function that is only used when a dynamic PIN code is needed.

It is not currently used in PTS, but it might be used in the future. One way to implement this function to be prepared for future use is

```
extern "C" char* WINAPI ImplicitSendPinCode(void)
{
std::string             strPrompt = "Please enter a PIN Code:";
return(ImplicitSendStyle(strPrompt,MMI_Style_Edit1));
}
```

Please refer to the previous section (3.2.4.1, "Scope of the return value") for details regarding the return value from this function.

3.2.6 **ImplicitTestCaseFinished()**

Declaration: extern "C" void WINAPI ImplicitTestCaseFinished(void);

Parameters: None

Return values: None

ImplicitTestCaseFinished() is called at the end of test case execution. It may be used to perform any cleanup that is needed, or to undo operations that were performed during ImplicitStartTestCase().

3.2.7 **Final cleanup**

The Implicit Send API does not provide a final cleanup function. Generally, such a function is not needed because the unloading of the DLL or the termination of the main PTS executable causes resources such as open files to be closed and dynamic memory to be released.

Should some form of final cleanup be required, the following is suggested:

1. Create a C++ class that contains the various objects that need to be cleaned up when the DLL is unloaded.

2. Declare an instance of the class at module level scope and make sure that the class is visible to all functions that need to access the data within. Variables declared at "module level scope" are those that are declared somewhere in a source file, but outside the boundary of all the functions in the file.

3. Place the necessary cleanup code in the class destructor. When a DLL (or executable program) is about to be unloaded from memory, the C++ runtime support invokes the destructor for each instance of a class that is defined at module level scope. Placing the cleanup code in the destructor ensures that it executes at the proper time.

4. Note that the standard C++ "singleton pattern" should not be used here. The standard singleton pattern uses a pointer to an instance of a class. The runtime support cleanup code will NOT call the destructor for an object that is accessed indirectly via a pointer.

As mentioned in section 3.2.4.1 ("Scope of the return value"), the sample source code for the default Implicit Send DLL uses this mechanism to address the ImplicitSendStyle() return value "persistency" issue.

Another possibility is to use a DllMain() function and perform the cleanup work when it is called with a reason code of DLL_PROCESS_DETACH. Note however that the use of DllMain() is no longer recommended by Microsoft. When using current versions of the Microsoft development tools, DllMain() isn't even required – the language runtime support provides its own.

3.3 Message tags

As mentioned in section 3.2.4, the strMmiText parameter passed to ImplicitSendStyle() is a string consisting of two parts. One of those parts is a message tag that uniquely identifies the message.

The purpose of the message tags is that they will always be the same regardless of the informational text that may be displayed to a test operator. This means that custom Implicit Send DLLs do not need to process the informational text – they only need to process the tag to know what request is being made.

The message is at the beginning of the strMmiText string and is in the following format

{<message number>,<test case name>,<test suite name>}

where

- <message number> identifies a message within a given test suite;
- <test case name> identifies the executing test case;
- <test suite name> identifies the Executable Test Suite than contains the currently executing test case.

The combination of <message number> and <test suite name> uniquely identifies a message across all Executable Test Suites. For example

{999,<any test case name>,OPP}

{999,<any test case name>,FTP}

are different messages even though they have the same <message number>.

The <test case name> helps to identify the usage of the message. For example, in the Object Push Profile (OPP) test suite <message number> 47 is used in every test case where the test operator (or automated test platform) needs to confirm that an object transfer occurred successfully. In test case OPP/SR/OPH/BV-03-I the operator needs to confirm that a new contact entry with the name "OPHBV03" is on the IUT. In OPP/SR/OPH/BV-07-I, a new calendar entry titled "OPHBV07" should have been created.

An automated test platform can distinguish between the two uses of <message number> 47 by looking at the <test case name>

{47,OPP/SR/OPH/BV-03-I,OPP}Please check that …

{47, OPP/SR/OPH/BV-07-I,OPP}Please check that …

3.3.1 Finding the tags

The default Implicit Send DLL provided with PTS removes the tags before sending the message text to the popup dialog. In other words, in normal operation PTS does not display the message tags.

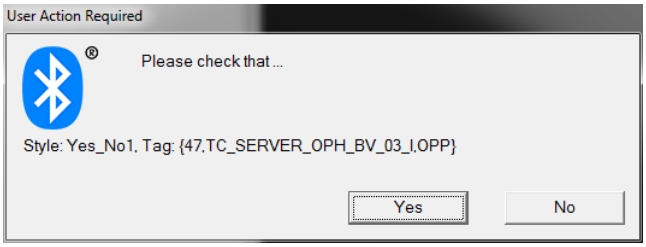
There are three ways to determine the tag associated with a given message:

3.3.1.1 Have the default Implicit Send DLL display the tag

Starting with version 4.5.3 of PTS, the default Implicit Send DLL has the option to display both the style of each message along with its tag. This functionality is enabled by adding the following lines to PTS.ini, normally found in C:\Users\<USER>\AppData\Roaming\Bluetooth_SIG\ProfileTuningSuite_6.

```
[ImplicitSend]
showTag=1
```

For the example above, this setting will cause the dialog to look something like this:



3.3.1.2 Consult the ATS document

Each test suite has an accompanying Abstract Test Suite (ATS) document that describes the details of the suite and its environment. In each ATS document there is a section on Implicit Send that includes a table listing each of the messages including their tags.

For example:

TSC_MMI_Confirm_ Preparation_Template	"{47,%s,OPP}Please check that <what to check for> This test case will <what test does> Press OK when you are ready to continue. Press CANCEL if you want to terminate this test case."

The "%s" in the message tag is replaced with the name of the currently executing test case at runtime.

3.4 MMI styles

The mmiStyle parameter to ImplicitSendStyle() provides direction about the contents of the strMmiText parameter along with an indication of the expected return value. When used with the default PTS Implicit Send DLL, the mmiStyle value selects the type of dialog box that will be displayed along with the buttons that will appear.

mmiStyle name	**Value**	**Message Type**	**Buttons Displayed by the default Implicit Send DLL**
MMI_Style_Ok_Cancel1	0x11041	Simple prompt	OK, Cancel Default: OK
MMI_Style_Ok_Cancel2*	0x11141	Simple prompt	Cancel
MMI_Style_Ok	0x11040	Simple prompt	OK

MMI_Style_Yes_No1	0x11044	Simple prompt	Yes, No Default: Yes
MMI_Style_Yes_No_Cancel1	0x11043	Simple prompt	Yes, No, Cancel Default: Yes
MMI_Style_Abort_Retry1	0x11042	Simple prompt	Abort, Retry, Ignore Default: Abort
MMI_Style_Edit1	0x12040	Request for data input	OK, Cancel Default: OK
MMI_Style_Edit2	0x12140	Select item from a list	OK, Cancel Default:OK

***Note: When ImplicitSendStyle() is called with style MMI_Style_Ok_Cancel2, implementation may signal the IUT the requested action after the message tag is identified but it should not block in the function. Otherwise, it may block PTS from progressing. Implementation should always return "OK".**

3.4.1 "Simple prompt" message type

These MMI styles are used to instruct the test operator or automation test platform to take an action. The action may be to make a connection from the IUT to the PTS, press a button on the IUT, etc.

For these messages, the strMmiText contains instructions about the action that is needed.

If the action can be successfully completed, the return value from ImplicitSendStyle() should be a pointer to a character string such as "OK". Please be sure to look at section 3.2.4.1 ("Scope of the return value") for important information about the "scope" of the string that is returned.

Successful completion is indicated in the default Implicit Send DLL when the user presses the OK, Yes, Retry or Ignore buttons.

If the operation cannot be completed, or the proper response to an action is to indicate that it did not happen, ImplicitSendStyle() should return a NULL pointer.

3.4.2 "Request for data input" message type

This message style is used when information is needed from the test environment, and that information is not available until after the test case begins execution. For example, the Passkey Entry association model in Secure Simple Pairing requires that one device display a six-digit number. The number must be entered on the other device to complete the association process. The number itself is random and is not generated until the Secure Simple Pairing process begins.

strMmiText describes the information that is being requested.

The return value from ImplicitSendStyle() should be a pointer to a character string containing the requested data, if the data is available. If the requested data is not available, or an error occurs, a NULL pointer should be returned.

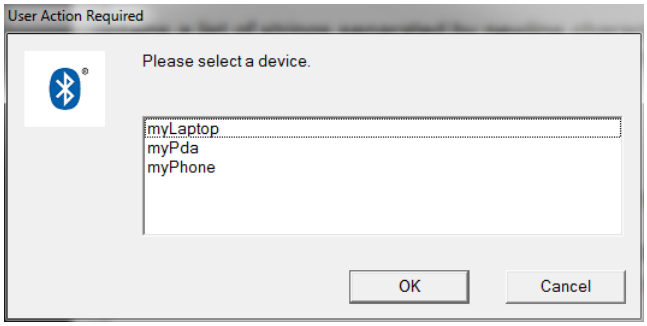
As has been noted above, the string pointed at by the return value should not be allowed to go out of scope. (Section 3.2.4.1, "Scope of the return value".)

3.4.3 "Select item from a list" message type

For this message strMmiText contains a list of strings separated by newline characters. (C/C++: '\n', ASCII code 0x0A.) The first string in the list contains the instructions to the user. The rest of the strings provide a list values for selection. The list is ended by an empty line.

For example, strMmiText may contain the following information

"{<message tag>}Please select a device.\nmyPhone\nmyLaptop\nmyPda\n\n"



The first item in the list ("Please select a device.") indicates that a list of devices follows and that one of the devices should be selected. There are three devices in the list: "myPhone", "myLaptop", and "myPda".

The items in the list are separated by newline characters, indicated as '\n' above. The extra '\n' at the end of the string marks the end of the list.

The expected return value is a pointer to a string containing one of the values from the list. A NULL pointer should be returned in case of error or if none of the values are appropriate at that point in the test.

The string to be returned will need to be a copy of one of the items in the list. Returning a pointer to the first character of one of the items in the list will not work since there is additional information (newlines and other items) following the selection. The copy is subject to the data scoping concerns mentioned earlier. (3.2.4.1, "Scope of the return value".)

3.5 Software build requirements

It was mentioned earlier (section 3.1, “Basic information”) that Implicit Send DLLs must be written in C++. A few additional requirements need to be considered when starting the development of an Implicit Send DLL.

- Microsoft Visual C++ **must** be used. C++ objects such as std::string are not guaranteed to be implemented the same way in every compiler. Mixing definitions is a recipe for trouble.

- Microsoft Visual C++ 2008/Visual Studio 2008 **must** be used for development of a custom Implicit Send DLL. Subtle runtime issues can occur when mixing different versions of the Visual C++ runtime environment.

  In particular, Visual C++ 2010 has been found to produce an Implicit Send DLL that **does not** work with PTS.

  It is possible to build the Implicit Send DLL with Visual Studio 2013 by setting project's Properties >> Configuration Properties >> General >> Platform Toolset to Visual Studio 2008 (v.90).

- The PTS Team **suggests** that custom DLLs be built in the “Release” configuration. Data structures and dynamically allocated memory may be laid out differently between “Debug” and “Release” configurations. (The “Debug” versions may contain extra elements to assist in the debugging process.) It is rarely a good idea to mix “Release” and “Debug”.

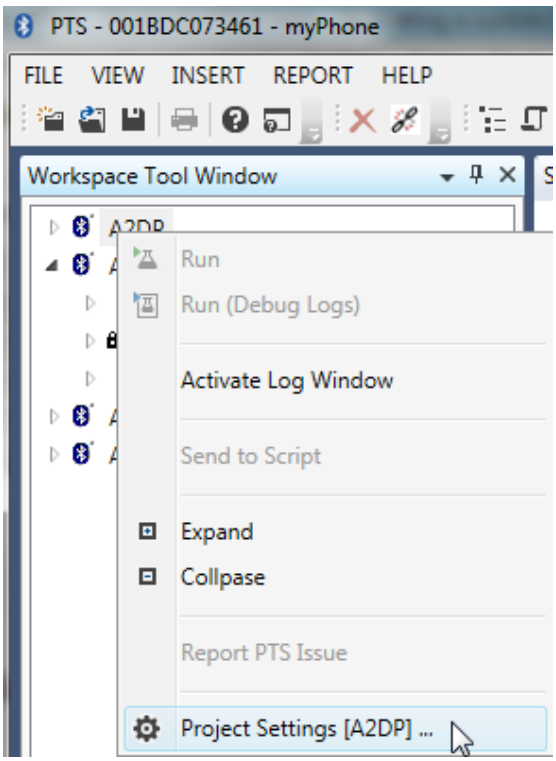
  Note that it is possible to use the Visual Studio Debugger on executables and DLLs that are built in the “Release” configuration. A few small changes may be needed to the Visual Studio project configuration to enable this functionality. Please contact PTS Technical Support for more information.

The requirements above are a result of the development environment used by the PTS Team. The Team uses Microsoft Visual C++ 2013 with the Platform Toolset set to Visual Studio 2008 (v.90). The PTS executables and DLLs are built in the “Release” configuration.

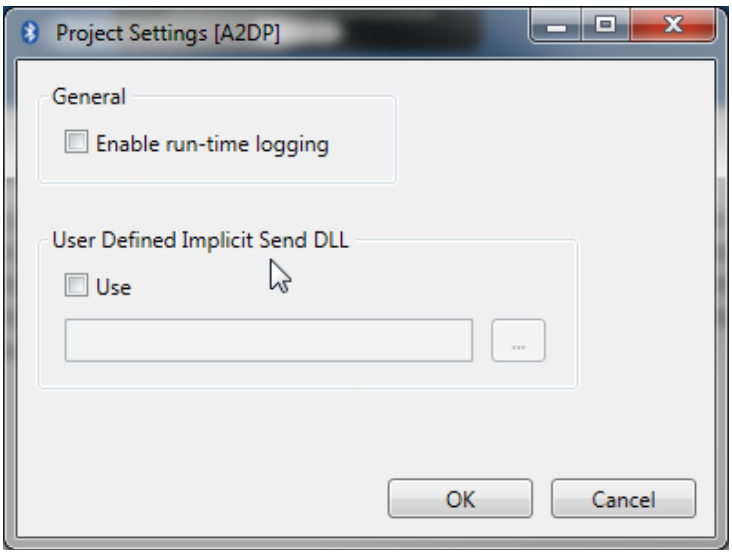
4 Activating a custom Implicit Send DLL

Once a custom Implicit Send DLL has been created, it is a simple matter to start using it with PTS. The DLL may be attached to a test suite selected by a PTS project via the Project Settings dialog.

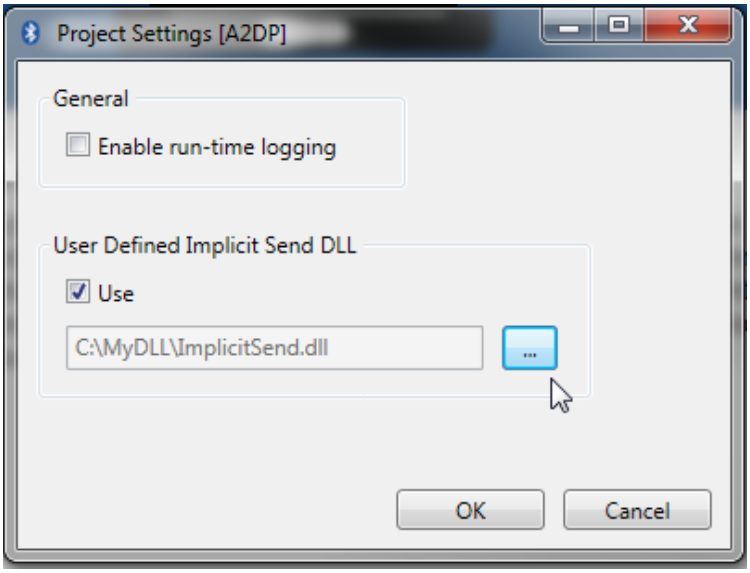
Begin the process by selecting the desired project (test suite) in the Workspace Test Case View window. Right click on the top level node of the project and select “Settings” from the menu that appears.



At the bottom of the Project Settings dialog is a section labeled “User Defined Implicit Send DLL”. Normally the box labeled “Use” is unchecked. When this box is unchecked the test suite will use the default Implicit Send DLL provided by PTS.



Place a check mark in the "Use" box. This will cause the text box and browse button to become active. Use Browse button ("…") to locate the custom Implicit Send DLL. After the DLL has been selected, press the OK button to record the change.



The custom Implicit Send DLL will be used starting with the next test case that is executed in the project.

4.1 Usage notes

- The custom Implicit Send DLL may be disabled at any time by returning to the Project Settings page and removing the checkmark from the box labeled "Use".

- The procedure above attaches a custom DLL to one and only project in a single workspace. The process must be repeated to use the DLL with other projects in the same workspace, or, with the same test suite (project) in a different workspace.

5 Technical tidbits

The following sections are items of general interest that should be reviewed by anyone who wishes to develop their own Implicit Send DLL.

5.1 Automatic dismissal of Implicit Send requests

At various points during the execution of a test, the test case implementation can detect that an action requested via Implicit Send has occurred. When this happens, the test case may attempt to complete the ImplicitSendStyle() or ImplicitSendPinCode() operation that is in progress.

This most commonly occurs with messages using MMI_Style_Ok_Cancel2. These messages tend to be "transient", for example, "Using the IUT, make a connection to the PTS". For these types of actions, there is no need to require operator interaction with PTS. The operator can simply take whatever steps are necessary on the IUT to cause the connection to happen; the test case can then detect the connection and take down the dialog.

The mechanism used to do this is to send simulated button presses to a dialog whose title is “User Action Required”. This works very well with the default Implicit Send DLL because all the dialogs it displays are titled “User Action Required”.

This however may not work very well with custom Implicit Send DLLs – especially ones that have no need to create popup dialogs. The test case will send the simulated button presses, but no one – most specifically the functions in the custom DLL – will be listening.

This situation may not be as bad as it seems. It’s likely, the IUT also knows that the requested action has taken place and would normally notify the user of the device. When the user of the device is replaced with an automation test platform, the platform can detect the situation and notify the custom Implicit Send DLL.

If it turns out that a custom DLL needs to know when the simulated button presses occur, it could create a hidden window whose title is “User Action Requested”. This would allow the delivery of the simulated button presses to the custom Implicit Send DLL.

For more information about the simulated button presses, please contact PTS technical support.

5.2 ImplicitSend() function

Earlier versions of PTS used a function called ImplicitSend(). This function has been replaced by ImplicitSendStyle() and is no longer used by the PTS.

5.3 TSPX_use_implicit_send

Most test suites have a IXIT value named TSPX_use_implicit_send which is used to enable or disable the Implicit Send functionality. Normally the value of this item is TRUE indicating that Implicit Send is to be used.

Setting the value to FALSE will disable Implicit Send for both user developed DLLs and the PTS default DLL.

The value of TSPX_use_implicit_send should be checked whenever it appears that the Implicit Send functionality is not working at all.

5.4 Sample source code

The PTS installation has a folder containing source code that may be used as a reference during development of a custom Implicit Send DLL. The source code itself is the complete source for the default Implicit Send DLL that is normally used by PTS. A Visual Studio project is included making it possible to build and execute the sample.

Of interest is implicit_send.cpp. In addition to the functions it provides, there are notes for an alternate implementation that connects to an automated test platform via TCP/IP.

The sample may be found under custom\implicit_send in the PTS installation folder. The default path to this location is

C:\Program Files [(x86)]\Bluetooth SIG\Bluetooth PTS\SampleCode\implicit_send

5.5 One DLL or many DLLs?

The tag data in the Implicit Send messages makes it possible to use the same DLL for more than one profile since each message is uniquely identified by <message number>, <currently executing test case>, and <currently active test suite>. (Section 3.3, “Message tags”.)

It may be convenient to develop one Implicit Send DLL for all uses and use the test suite name in the message tag to know which profile is requesting assistance. On the other hand, a single DLL may be more complicated to construct and maintain, suggesting that a custom DLL for each test suite might be more appropriate.

The point to keep in mind is that PTS can support either design decision – developers of custom Implicit Send DLLs are not locked into one way or the other.

5.6 Hybrid environments

It may be desirable to replace some, but not all, of PTS's default Implicit Send handling. This is possible by doing the following:

- Create a custom Implicit Send DLL.
- In the custom DLL, InitImplicitSend() could dynamically load the default DLL using the Windows LoadLibrary() and GetProcAddress() API functions.
- When a message arrives at ImplicitSendStyle() in the custom DLL, the function could look at the message tag and decide whether or not it wants to handle the message. If the custom ImplicitSendStyle() does not want to handle the message, it could call ImplicitSendStyle() in the default DLL using the same parameters.

  The return value from the default DLL would then become the return value for the custom DLL.