Programmer's Guide

BCM1000-BTW

BCM1000-BTW Software Development Kit

1000-BTW-PG108-R

5300 California Avenue • Irvine, CA 92617 • Phone: 949-926-5000 • Fax: 949-926-5203

12/3/08

REVISION HISTORY

Revision	*Date*	*Change Description*
1000-BTW-PG108-R		**Updated:** • "Bond ( )" on page 21 • "BondQuery ( )" on page 22 • "CreateCOMPortAssociation ( )" on page 30 **Added:** • "BondEx( )" on page 19 • "BondReply( )" on page 21

Broadcom Corporation
5300 California Avenue
Irvine, CA 92617

Revision	*Date*	*Change Description*
1000-BTW-PG107-R	7/12/07	**Updated:** • Section 1: "Introduction" on page 1 • Figure 1: "Block Diagram of the BTW SDK," on page 2 • "Development Kit Components" on page 4 • Section : "Build Environments" on page 5 • "Compatibility" on page 7 • "Compatibility with Broadcom's Bluetooth Software for Windows" on page 7 • "Compatibility with Broadcom's Vista Profile Pack" on page 7 • "Compatibility and Visual Studio 2005 Support" on page 8 • "Compatibility and New Features" on page 8 • "CBtIf" on page 13 • "GetLastDiscoveryResult ( )" on page 18 • "GetLocalDeviceVersionInfo ( )" on page 24 • "ReadLinkMode ( )" on page 31 • "RegisterForLinkStatusChanges ( )" on page 31 • "Connection Statistics — tBT_CONN_STATS" on page 32 • "Reconfigure ( )" on page 44 • "GetConnectionStats ( )" on page 46 • "AddAttribute ( )" on page 57 • "CSdpDiscoveryRec" on page 61 • "FindProtocolListElem ( )" on page 62 • "FindAdditionalProtocolListElem ( )" on page 62 • "GetConnectionStats ( )" on page 71 • "RegOppMultiCloseCB ( )" on page 98 • "RegOppMultiPushCB ( )" on page 99 • "GetConnectionStats ( )" on page 102 • "Pure virtual OnStateChange ( )" on page 104 • "Pure virtual OnStateChange ( )" on page 104 • "GetConnectionStats ( )" on page 106 • "Configuration Notes" on page 109 • "GetConnectionStats ( )" on page 110 • "GetConnectionStats ( )" on page 114 **Removed:** - Bulletted text "Evaluation or retail version of the WIDCOMM BCM1000-BTW software from Broadcom Corporation" in Section 2: "System Requirements" on page 3

Revision	*Date*	*Change Description*
1000-BTW-PG106-R	11/17/06	**Updated:** • Section 1: "Introduction" on page 1 • Section 2: "System Requirements" on page 3 • Section 3: Implementation: - "Development Kit Components" on page 4 - "BCM1000-BTW Stack and SDK Compatibility" on page 7 • Section 4: Development Kit Classes: "Derived Functions Run on Separate Threads" on page 11 • Section 5: Class Descriptions and Usage: - "GetLastDiscoveryResult ( )" on page 18 - "GetLocalDeviceVersionInfo ( )" on page 23 **Added:** • To Section 3: Implementation:"Build Environments" on page 5 • Figure 3: "Project Settings for MS VS 2005," on page 6
1000-BTW-PG105-R	07/27/06	**Updated:** • "Development Kit Components" on page 4 • "Global Object Instantiation" on page 10 • "CBtIf" on page 15 • The prototype of "GetLastError ( ) " on page 182 **Clarified:** • "virtual OnDeviceResponded ( )" on page 17 • "GetLocalServiceName ( )" on page 29 • "SetEscoMode ( )" on page 39 • "Connect ( )" on page 48 • "AssignScnValue ( )" on page 71 • "virtual OnEventReceived ( )" on page 78 • "tOBEX_ERRORS" on page 163 • "CreateAudioConnection ( )" on page 173 **Added:** • Values to Table 29: "Headphone Return Codes," on page 183 • Values to Table 30: "Headphone Status Values," on page 184 **Replaced:** • Parameter "LP**T**STR" with "LPSTR" throughout the document.
1000-BTW-PG104-R	09/12/05	**Added:** • To "Section 5: Class Descriptions and Usage" (pg 11):" - New Function, "ReadLinkMode ( )" on page 30 **Modified:** • "SendVendorSpecific_HCICMD ( )" on page 29, Return value, SENDVENDOR_HCICMD_BUFFER_TOO_BIG • "SetSniffMode ( )" on page 30, multiple text edits • "CancelSniffMode ( )" on page 30, Returns TRUE, text edit.

Revision	***Date***	***Change Description***
1000-BTW-PG103-R	08/30/05	**Added:** • A2DP to Section 1: "Introduction" on page 1. • "CPrintClient" and "CHeadphoneClient" to Table 1: "Classes Offered in the SDK," on page 5. • To the "CBtIf" chapter: - Return value "REPEATED_ATTEMPTS" to "Bond ( ) ." on page 19 - Additional return values to "SendVendorSpecific_HCICMD ( )" on page 29 - New function, "IsRemoteDeviceConnected ( )" on page 30 - New function, "RegisterForLinkStatusChanges ( )" on page 31 - New RSSI range to "Connection Statistics — tBT_CONN_STATS" on page 32 - Additional return codes to Table 6: "Audio Return Codes," on page 33 - New function "SetEscoMode ( )" on page 36 - New function "RegForEScoEvts ( )" on page 37 - New function "ReadEScoLinkData ( )" on page 38 - New function "ChangeEscoLinkParms ( )" on page 39 - New function "EScoConnRsp ( )" on page 39 ***Note:*** The five immediately preceeding new functions are replicated in these chapters: - "CL2CapConn" beginning on page-52 - "CRfCommPort" beginning on page-79 - "CPrintClient" beginning on page-155 - "CObexServer" beginning on page -172 • New chapter "CHeadphoneClient" on page 180 • "CreateConnection ( )" on page 117, of the CSppClient chapter, added to the introduction and appended a note. **Modified:** • "AddSupportedFormatsList ( )" on page 63 - modified the pDataTypeValue parameter description and the accompanying note.
1000-BTW-PG102-R	03/10/05	Audio Connections update, minor corrections and cleanup.
1000-BTW-PG101-R	11/23/04	Updated "Development Kit Components" on page 4.
1000-BTW-PG100-R	11/1/04	Initial release.

TABLE OF CONTENTS

Section 1: Introduction 1

Section 2: System Requirements 3

Section 3: Implementation 4

Development Kit Components 4

Build Environments 5

Compatibility 7

Compatibility with Broadcom's Bluetooth Software for Windows 7

Compatibility with Broadcom's Vista Profile Pack 7

Compatibility and Visual Studio 2005 Support 8

Compatibility and New Features 8

Section 4: Development Kit Classes 9

Virtual Functions 9

Constructors and Destructors for Development Kit Classes 9

Global Object Instantiation 10

Use of GUID to Represent UUID 10

Use of Maximum Transmission Unit (MTU) 11

Derived Functions Run on Separate Threads 11

Examples (RFCOMM) 12

Client 12

Server 12

Section 5: Class Descriptions and Usage 13

CBtIf 13

StartInquiry ( ) 14

StopInquiry ( ) 14

virtual OnDeviceResponded ( ) 15

virtual OnInquiryComplete ( ) 15

StartDiscovery ( ) 16

SwitchRole ( ) 16

virtual OnDiscoveryComplete ( ) 17

GetLastDiscoveryResult ( ) 18

ReadDiscoveryRecords ( ) 19

BondEx( ) 19
BondReply( ) 21
Bond ( ) 21
BondQuery ( ) 22
UnBond ( ) 22
IsDeviceReady ( ) 22
IsStackServerUp ( ) 23
virtual OnStackStatusChange ( ) 23
GetExtendedError ( ) 23
SetExtendedError ( ) 24
GetLocalDeviceInfo ( ) 24
GetLocalDeviceName ( ) 24
SetLocalDeviceName ( ) 24
GetRemoteDeviceInfo ( ) 25
GetNextRemoteDeviceInfo ( ) 25
GetLocalDeviceVersionInfo ( ) 26
GetRemoteDeviceVersionInfo ( ) 27
GetDKVersionInfo ( ) 28
GetBTWVersionInfo ( ) 28
GetLocalServiceName ( ) 28
GetNextLocalServiceName ( ) 29
Add_Printer ( ) 29
Remove_Printer ( ) 29
CreateCOMPortAssociation ( ) 30
RemoveCOMPortAssociation ( ) 31
ReadCOMPortAssociation ( ) 31
SetLinkSupervisionTimeOut ( ) 31
SendVendorSpecific_HCICMD ( ) 32
SetSniffMode ( ) 32
CancelSniffMode ( ) 32
ReadLinkMode ( ) 33
IsRemoteDeviceConnected ( ) 33
RegisterForLinkStatusChanges ( ) 33
Connection Statistics — tBT_CONN_STATS 34
GetConnectionStats ( ) 34

Audio Connections 35
CreateAudioConnection ( ) 35
RemoveAudioConnection ( ) 36
virtual OnAudioConnected ( ) 36
virtual OnAudioDisconnect ( ) 37
SetEscoMode ( ) 37
RegForEScoEvts ( ) 38
ReadEScoLinkData ( ) 39
ChangeEscoLinkParms ( ) 40
EScoConnRsp ( ) 40
CL2CapIf 41
AssignPsmValue ( ) 41
Register ( ) 42
Deregister ( ) 42
GetPsm ( ) 42
SetSecurityLevel ( ) 43
CL2CapConn 44
Listen ( ) 44
Accept ( ) 45
Reject ( ) 45
Connect ( ) 46
Reconfigure ( ) 46
Disconnect ( ) 47
Write ( ) 47
GetConnectionStats ( ) 48
SwitchRole ( ) 48
virtual OnIncomingConnection ( ) 48
virtual OnConnectPendingReceived ( ) 48
virtual OnConnected ( ) 49
virtual OnDataReceived ( ) 49
virtual OnCongestionStatus ( ) 49
virtual OnRemoteDisconnected ( ) 49
SetLinkSupervisionTimeOut ( ) 50
Audio Connections 50
CreateAudioConnection ( ) 50

Basic Version 50
Enhanced Version 51
RemoveAudioConnection ( ) 51
virtual OnAudioConnected ( ) 51
virtual OnAudioDisconnect ( ) 51
SetEscoMode ( ) 52
RegForEScoEvts ( ) 52
ReadEScoLinkData ( ) 54
ChangeEscoLinkParms ( ) 55
EScoConnRsp ( ) 55
CSdpService 56
AddServiceClassIdList ( ) 56
AddServiceName ( ) 56
AddProfileDescriptorList ( ) 57
AddL2CapProtocolDescriptor ( ) 57
AddRFCommProtocolDescriptor ( ) 57
AddProtocolList ( ) 57
AddAdditionProtoLists ( ) 58
AddLanguageBaseAttrIDList ( ) 59
MakePublicBrowseable ( ) 59
SetAvailability ( ) 59
AddAttribute ( ) 59
DeleteAttribute ( ) 62
AddSupportedFormatsList ( ) 62
CommitRecord() 63
CSdpDiscoveryRec 63
FindRFCommScn ( ) 63
FindL2CapPsm ( ) 63
FindProtocolListElem ( ) 64
FindAdditionalProtocolListElem ( ) 64
FindProfileVersion ( ) 65
FindAttribute ( ) 65
CRfCommIf 67
AssignScnValue ( ) 67
GetScn ( ) 67

SetSecurityLevel ( ) 68
CRfCommPort 69
PORT_RETURN_CODE 69
OpenServer ( ) 69
OpenClient ( ) 70
Close ( ) 70
IsConnected ( ) 70
SetFlowEnabled ( ) 70
SetModemSignal ( ) 71
GetModemStatus ( ) 71
SendError ( ) 72
Purge ( ) 72
Write ( ) 72
GetConnectionStats ( ) 73
virtual OnDataReceived ( ) 73
virtual OnEventReceived ( ) 73
virtual OnFlowEnabled ( ) 74
SwitchRole ( ) 74
SetLinkSupervisionTimeOut ( ) 75
Audio Connections 75
CreateAudioConnection ( ) 75
Basic Version 76
Enhanced Version 76
RemoveAudioConnection ( ) 76
virtual OnAudioConnected ( ) 76
virtual OnAudioDisconnect ( ) 77
SetEscoMode ( ) 77
RegForEScoEvts ( ) 78
ReadEScoLinkData ( ) 79
ChangeEscoLinkParms ( ) 80
EScoConnRsp ( ) 80
CFtpClient 81
Special Registry Settings 81
FTP_RETURN_CODE 82
OpenConnection ( ) 82

CloseConnection ( ) 82
PutFile ( ) 83
GetFile ( ) 83
FolderListing ( ) 83
ChangeFolder ( ) 83
DeleteFile ( ) 84
Abort ( ) 84
Parent ( ) 84
Root ( ) 84
CreateEmpty ( ) 85
CreateFolder ( ) 85
SetSecurity ( ) 85
GetExtendedError ( ) 86
SetExtendedError ( ) 86
FTP_RESULT_CODE 86
virtual OnProgress ( ) 87
virtual OnOpenResponse ( ) 87
virtual OnCloseResponse ( ) 88
virtual OnPutResponse ( ) 88
virtual OnGetResponse ( ) 88
virtual OnCreateResponse ( ) 88
virtual OnDeleteResponse ( ) 89
virtual OnChangeFolderResponse ( ) 89
virtual OnFolderListingResponse ( ) 89
virtual OnXmlFolderListingResponse ( ) 91
virtual OnAbortResponse ( ) 91

COppClient 92

OPP_RETURN_CODE 93
Push ( ) 93
Pull ( ) 93
Exchange ( ) 94
Abort ( ) 94
SetSecurity ( ) 95
GetExtendedError ( ) 95
SetExtendedError ( ) 96

OPP_RESULT_CODE 96
virtual OnProgress ( ) 97
virtual OnPushResponse ( ) 97
virtual OnPullResponse ( ) 97
virtual OnExchangeResponse ( ) 98
Basic Version 98
Enhanced Version 98
virtual OnAbortResponse ( ) 99
COppMultiPush 99
OpenOppConnection ( ) 99
CloseOppConnection ( ) 99
MultiPush ( ) 100
RegOppMultiOpenCB ( ) 100
RegOppMultiCloseCB ( ) 100
RegOppMultiPushCB ( ) 101
CLapClient 102
LAP_RETURN_CODE 102
CreateConnection ( ) 103
PANU/NAP/GN Version 103
PPP Version 103
RemoveConnection ( ) 103
GetConnectionStats ( ) 104
SetSecurity ( ) 104
GetExtendedError ( ) 105
SetExtendedError ( ) 105
Pure virtual OnStateChange ( ) 106
CDunClient 107
DUN_RETURN_CODE 107
CreateConnection ( ) 107
RemoveConnection ( ) 108
GetConnectionStats ( ) 108
SetSecurity ( ) 108
GetExtendedError ( ) 109
SetExtendedError ( ) 109
Pure virtual OnStateChange ( ) 110

CSppClient 111
Configuration Notes 111
SPP_CLIENT_RETURN_CODE........ 111
CreateConnection ( ) 112
RemoveConnection ( ) 112
GetConnectionStats ( ) 112
GetExtendedError ( ) 113
SetExtendedError ( ) 113
Pure virtual OnClientStateChange ( ) 114
CreateCOMPort ( ) 114
CSppServer 115
Configuration Notes 115
SPP_SERVER_RETURN_CODE........ 115
CreateConnection ( ) 116
RemoveConnection ( ) 116
GetConnectionStats ( ) 116
GetExtendedError ( ) 117
SetExtendedError ( ) 117
Pure virtual OnServerStateChange ( ) 118
CreateCOMPort ( ) 118
CObexHeaders 118
SetCount ( ) 119
DeleteCount ( ) 119
GetCount ( ) 119
SetName ( ) 119
DeleteName ( ) 119
GetNameLength ( ) 120
GetName ( ) 120
SetType ( ) 120
DeleteType ( ) 120
GetTypeLength ( ) 121
GetType ( ) 121
SetLength ( ) 121
DeleteLength ( ) 121
GetLength ( ) 121

SetTime ( ) 122
DeleteTime ( ) 122
GetTime ( ) 122
SetDescription ( ) 122
DeleteDescription ( ) 123
GetDescriptionLength ( ) 123
GetDescription ( ) 123
AddTarget ( ) 124
GetTargetCnt ( ) 124
DeleteTarget ( ) 124
GetTargetLength ( ) 125
GetTarget ( ) 125
AddHttp ( ) 125
GetHttpCnt ( ) 126
DeleteHttp ( ) 126
GetHttpLength ( ) 126
GetHttp ( ) 126
SetBody ( ) 127
DeleteBody ( ) 127
GetBodyLength ( ) 127
GetBody ( ) 128
SetWho ( ) 128
DeleteWho ( ) 128
GetWhoLength ( ) 129
GetWho ( ) 129
AddAppParam ( ) 129
GetAppParamCnt ( ) 130
DeleteAppParam ( ) 130
GetAppParamLength ( ) 130
GetAppParam ( ) 130
AddAuthChallenge ( ) 131
GetAuthChallengeCnt ( ) 131
DeleteAuthChallenge ( ) 131
GetAuthChallengeLength ( ) 132
GetAuthChallenge ( ) 132

AddAuthResponse ( ) 133
GetAuthResponseCnt ( ) 133
DeleteAuthResponse ( ) 133
GetAuthResponseLength ( ) 134
GetAuthResponse ( ) 134
SetObjectClass ( ) 134
DeleteObjectClass ( ) 135
GetObjectClassLength ( ) 135
GetObjectClass ( ) 135
AddUserDefined ( ) 135
GetUserDefinedCnt ( ) 136
DeleteUserDefined ( ) 136
GetUserDefinedLength ( ) 136
GetUserDefined ( ) 136
CObexUserDefined 137
SetHeader ( ) 137
Single Byte Header 137
Four Byte Header 137
UNICODE Text Header 137
Octet Array Header 137
GetUserType ( ) 138
GetByte ( ) 138
GetFourByte ( ) 138
GetLength ( ) 138
GetText ( ) 138
GetOctets ( ) 138
CObexClient 139
tOBEX_ERRORS 139
Open ( ) 140
SetPath ( ) 140
Put ( ) 141
Get ( ) 141
Abort ( ) 141
Close ( ) 142
Pure virtual OnOpen ( ) 142

Pure virtual OnClose ( ) 143
virtual OnAbort ( ) 143
virtual OnPut ( ) 144
virtual OnGet ( ) 144
virtual OnSetPath ( ) 145
SetLinkSupervisionTimeOut ( ) 145
SetWriteTimeOut ( ) 145
Audio Connections 146
CreateAudioConnection ( ) 146
RemoveAudioConnection ( ) 146
virtual OnAudioConnected ( ) 146
virtual OnAudioDisconnect ( ) 147
SetEscoMode ( ) 147
RegForEScoEvts ( ) 148
ReadEScoLinkData ( ) 149
ChangeEscoLinkParms ( ) 150
EScoConnRsp ( ) 150
CObexServer 151
tOBEX_ERRORS 151
Register ( ) 151
RegisterBinaryTarget ( ) 152
Unregister ( ) 152
OpenCnf ( ) 153
SetPathCnf ( ) 153
PutCnf ( ) 154
PutCreateCnf ( ) 154
PutDeleteCnf ( ) 155
GetCnf ( ) 155
AbortCnf ( ) 156
CloseCnf ( ) 156
GetRemoteBDAddr ( ) 156
SwitchRole ( ) 157
Pure virtual OnOpenInd ( ) 157
Pure virtual OnSetPathInd ( ) 157
Pure virtual OnPutInd ( ) 158

Pure virtual OnPutCreateInd ( ) 158
Pure virtual OnPutDeleteInd ( ) 158
Pure virtual OnGetInd ( ) 159
Pure virtual OnAbortInd ( ) 159
Pure virtual OnCloseInd ( ) 159
SetLinkSupervisionTimeOut ( ) 159
Audio Connections 160
CreateAudioConnection ( ) 160
Basic Version 160
Enhanced Version 160
RemoveAudioConnection ( ) 161
virtual OnAudioConnected ( ) 161
virtual OnAudioDisconnect ( ) 161
SetEscoMode ( ) 161
RegForEScoEvts ( ) 162
ReadEScoLinkData ( ) 163
ChangeEscoLinkParms ( ) 164
EScoConnRsp ( ) 165
CPrintClient 165
ePRINT_PROFILE 165
ePRINT_STATE 166
ePRINT_ERROR 166
BTPRINTSTRUCT 167
Start ( ) 167
Cancel ( ) 167
virtual OnStateChange ( ) 167
GetState ( ) 168
GetBytesSent ( ) 168
GetBytesTotal ( ) 168
GetLastError ( ) 168
CHeadphoneClient 169
HEADPHONE_RETURN_CODE 169
HEADPHONE_STATUS 170
ConnectHeadphone ( ) 170
DisconnectHeadphone ( ) 170

SetSecurity ( )....171
GetExtendedError ( )....171
SetExtendedError ( )....172
GetConnectionStats ( )....172
RegStatusChangeCB ( )....172

Section 6: References.... 174

LIST OF FIGURES

Figure 1: Block Diagram of the BTW SDK 2
Figure 2: Project Settings for MS VC++ 6.0 6
Figure 3: Project Settings for MS VS 2005 6

List of Tables

Table 1: Classes Offered in the SDK 4
Table 2: SDK Library Usage 5
Table 3: DISCOVERY_RESULT 18
Table 4: Supported Pairing Processes 20
Table 5: STACK_STATUS 23
Table 6: REM_DEV_INFO_RETURN_CODE 25
Table 7: Bluetooth Hardware Manufacturer's List 27
Table 8: Audio Return Codes 35
Table 9: SDP_RETURN_CODE 56
Table 10: Service Record Attribute IDs 61
Table 11: Service Record Attribute Types 62
Table 12: PORT_RETURN_CODE 69
Table 13: Modem Signals 71
Table 14: Modem Status Codes 71
Table 15: Error Code Flags 72
Table 16: Port Events 74
Table 17: FTP_RETURN_CODE 82
Table 18: FTP_RESULT_CODE 86
Table 19: OPP_RETURN_CODE 93
Table 20: OPP_RESULT_CODE 96
Table 21: LAP_RETURN_CODE 102
Table 22: DUN_RETURN_CODE 107
Table 23: SPP_CLIENT_RETURN_CODE 111
Table 24: SPP_SERVER_RETURN_CODE 115
Table 25: tOBEX_ERRORS 139
Table 26: ePRINT_PROFILE 165
Table 27: ePRINT_STATE 166
Table 28: ePRINT_ERROR 166
Table 29: BTPRINTSTRUCT 167
Table 30: Headphone Return Codes 169
Table 31: Headphone Status Values 170

Section 1: Introduction

This document describes Broadcom's BCM1000-BTW Software Development Kit (SDK). The BCM1000-BTW SDK is designed to support custom Bluetooth application developers with protocol-layer access to the following:

- GAP
- L2CAP
- RFCOMM
- OPP
- FTP
- SDP
- SPP
- LAP [1]
- OBEX [2]
- DUN
- BPP
- HCRP
- A2DP

For more information on protocol specifications, go to the Bluetooth Special Interest Group website at http://www.bluetooth.org/spec.

The SDK consists of:

- A Static Link Library for each supported Build Configuration
- A Dynamic Link Library for redeployment in certain scenarios
- C++ header files
- Source and project files for sample applications
- This document, which provides usage and interface details for the library

The operational context for applications produced with the SDK is a standard Bluetooth PC platform on which Broadcom's WIDCOMM BTW software is installed. Broadcom's two major WIDCOMM BTW products are supported:

- BTW 5 Broadcom Bluetooth Software for Windows (Broadcom's Bluetooth stack)
- BTW 6 Broadcom Vista Profile Pack (for Microsoft's Bluetooth stack)

Note: Some SDK components may not be fully supported in the BTW 6 operational context. For detailed information, see "Compatibility with Broadcom's Bluetooth Software for Windows" on page 7 and "Compatibility with Broadcom's Vista Profile Pack" on page 7.

For detailed information on supported versions and specific API and class support for BTW 6 platforms, refer to Broadcom's latest SDK release notes available at http://www.broadcom.com/products/bluetooth_sdk.php).

For convenience, a function has been added to the SDK named IsBroadcomStack(). This function returns TRUE if the application is running on the Bluetooth Software for Windows platform and returns FALSE if it is running on the Vista Profile Pack platform.

Custom applications developed using the SDK, such as BTNeighborhood and BTTray, can run concurrently with the standard BTW applications and profiles.

Functionally, the SDK presents a C++ interface to the inquiry and discovery functions of the GAP and SDP layers. SDK functions allow access to the discovery database and the attribute values it contains. Custom applications can add to and delete from the SDP service database. For the OPP, FTP, L2CAP, OBEX, and RFCOMM layers, C++ classes are provided for all detailed functions exposed by the BTW stack implementation. For the SPP, LAP, DUN, BPP, A2DP, and HCRP profiles, C++ classes are also provided to allow applications to create and manage connections to these service profiles.

Figure 1 provides a block diagram of the SDK in the BTW context.

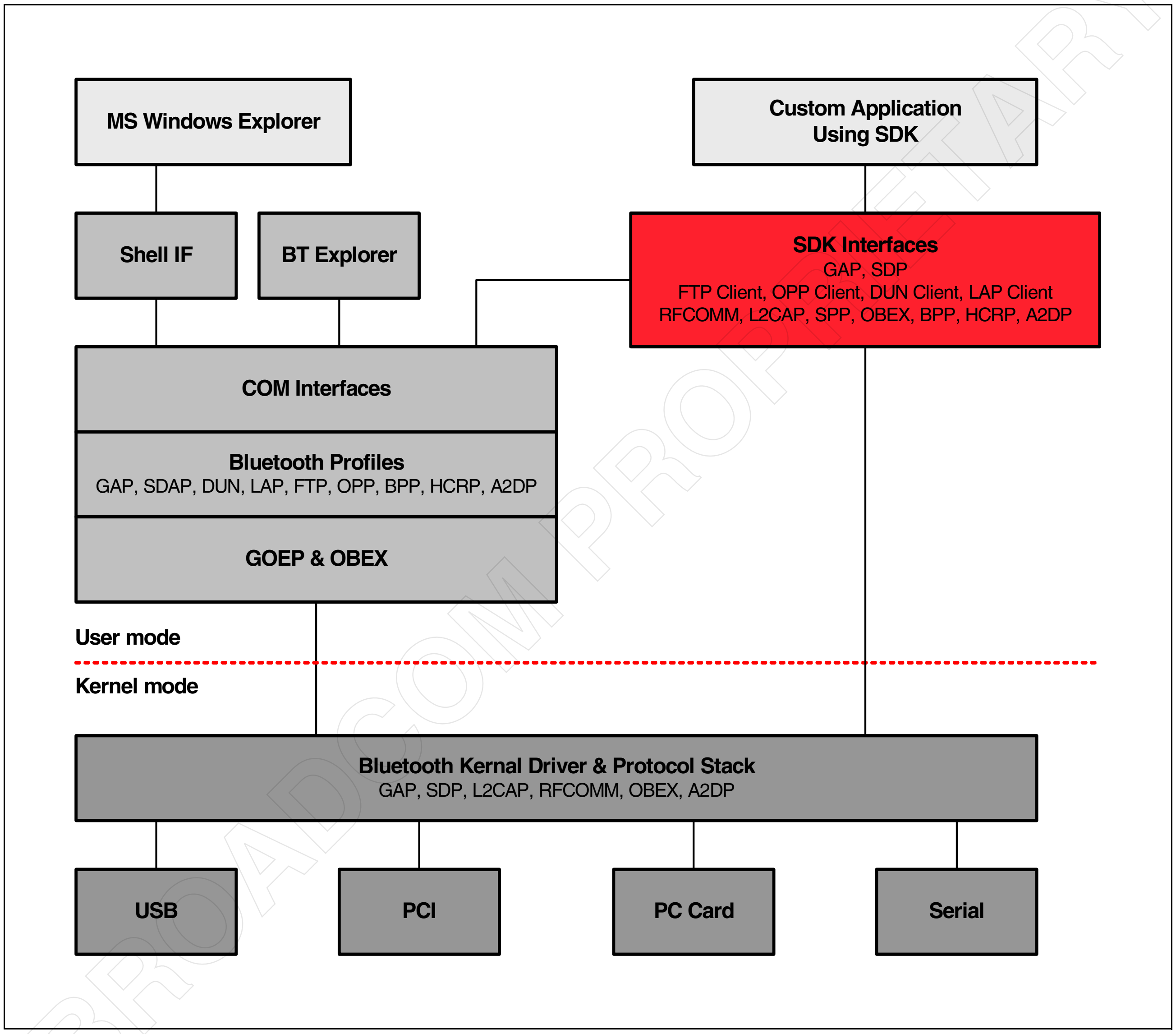

MS Windows Explorer
Custom Application Using SDK
Shell IF
BT Explorer
SDK Interfaces
GAP, SDP
FTP Client, OPP Client, DUN Client, LAP Client
RFCOMM, L2CAP, SPP, OBEX, BPP, HCRP, A2DP
COM Interfaces
Bluetooth Profiles
GAP, SDAP, DUN, LAP, FTP, OPP, BPP, HCRP, A2DP
GOEP & OBEX
User mode
Kernel mode
Bluetooth Kernal Driver & Protocol Stack
GAP, SDP, L2CAP, RFCOMM, OBEX, A2DP
USB
PCI
PC Card
Serial

Figure 1: Block Diagram of the BTW SDK

Section 2: System Requirements

The following hardware and software are required to develop software using the SDK:

- Windows® 98 SE / Windows Me / Windows 2000 / Windows XP / Windows Vista (system administrator privileges required when running Windows 2000 and Windows Vista)
- Microsoft Visual C++ 6.0 or Visual Studio® 6.0 with service pack 5 installed or Microsoft Visual Studio 2005
- 64 MB memory
- At least 50 MB of available hard disk space (after Visual C++ or Visual Studio is installed). Up to 1 GB hard disk space may be required to build every sample application under every build configuration provided.

To deploy software built using the SDK requires a PC with the following hardware and software (can be the same as development PC):

- Windows® 98 SE / Windows Me / Windows 2000 / Windows XP / Windows Vista (system administrator privileges required when running Windows 2000 and Windows Vista)
- Evaluation or retail version of the WIDCOMM BCM1000-BTW software from Broadcom Corporation
- Bluetooth radio (integrated or USB)

Section 3: Implementation

Development Kit Components

The SDK consists of several versions of a library file interface to the BTW software, and several header files.

Multiple versions of the library files (either `WidcommSdkLib.lib` or `BtWdSdkLib.lib`) are supplied for use in various Build Configurations. See "Build Environments" on page 5 for more information.

BtIfDefinitions.h is a header file that defines constants and structures used by the SDK and applications.

BtIfClasses.h is a header file that defines the classes offered by the SDK (see Table 1).

BtIfObexHeaders.h is a header file that defines the classes for OBEX.

BtwLib.h is the header file that includes *BtIfDefinitions.h, BtIfClasses.h* and facilitates linking the correct interface library automatically, based on the Build Configuration. The application should include this header file, instead of *BtIfDefinitions.h* and *BtIfClasses.h.* If it is an OBEX application, then *BtIfObexHeaders.h* should be included as well.

com_error.h is a header file that contains an enumerated list of error codes common to the BTW stack and the SDK.

Table 1: Classes Offered in the SDK

SDK Class	*Function*
CBtIf	Provides interface level management functions, e.g., methods for doing inquiry and service discovery
CL2CapIf	Interfaces with L2CAP for Protocol/Service Multiplexor (PSM) allocation & registration, and security settings
CL2CapConn	Controls L2CAP connections
CSdpService	Manages an SDP service record
CSdpDiscoveryRec	Contains an SDP discovery record and methods to query it
CRfCommIf	Interfaces with RFCOMM for Service Channel Number (SCN) allocation and security settings
CRfCommPort	Controls RFCOMM connections
CFtpClient	Provides the client-side interface for FTP
COppClient	Provides the client-side interface for OPP
COppMultiPush	Provides an extended OPP client interface supporting multiple push items
CLapClient	Provides the client-side interface for LAN access using PPP
CDunClient	Provides the client side interface for DUN
CSppClient	Provides the client-side interface for SPP COM port connections
CSppServer	Provides the server-side interface for SPP COM port connections
CObexServer	Provides the server-side interface for OBEX
CObexClient	Provides the client-side interface for OBEX
CObexHeaders	Container class for all OBEX header structures
CObexUserDefined	Container class for the user-defined type of OBEX header
CPrintClient	Provides the client side interface to HCRP, BPP, and SPP printing.
CHeadphoneClient	Provides the client side interface for A2DP.

BUILD ENVIRONMENTS

The SDK can be used with either the Microsoft Visual C++ 6.0 compiler or Microsoft Visual Studio 2005 C++ compiler. Support is provided for either 32-bit or AMD64 target systems. There are four versions of the interface library file provided with the SDK, one for each combination. Applications must link with only one of the versions of the static library interface files (`WidcommSdkLib.lib` or `BtWdSdkLib.lib`) supplied in the `{installation_base_directory}\SDK\Release` directory tree, depending on the Build Configuration (target platform + compiler).

Table 2: SDK Library Usage

Interface Library:	*Build Configuration*
Release\WidcommSdkLib.lib	- 32 bit VC++ 6.0
Release\BtWdSdkLib.lib	- 32 bit VS2005
Release\amd64\WidcommSdkLib.lib	- 64 bit VC++ 6.0
Release\x64\BtWdSdkLib.lib	- 64 bit VS2005

For applications built using VC++ 6.0, targeting AMD64 systems, the October 2003 update for the February 2003 Microsoft Platform SDK is required to use the VC++ 6.0 compiler to build 64-bit applications. In that Build Configration, the compiler must be launched from a Windows Server 2003 AMD64 Build Environment window. Refer to that Platform SDK documentation for information on opening a Build Environment window. In the VC++ 6.0 IDE, the Release AMD64 target is used to link the Release\amd64\WidcommSdkLib.lib file.

For applications built using VS2005 targeting AMD64-bit systems, the Microsoft Platform SDK shipped with VS2005 contains the correct native support for the x64 target.

Note: Installation of the Microsoft Platform SDK does NOT install the compiler or library support for the x64 targets by default.

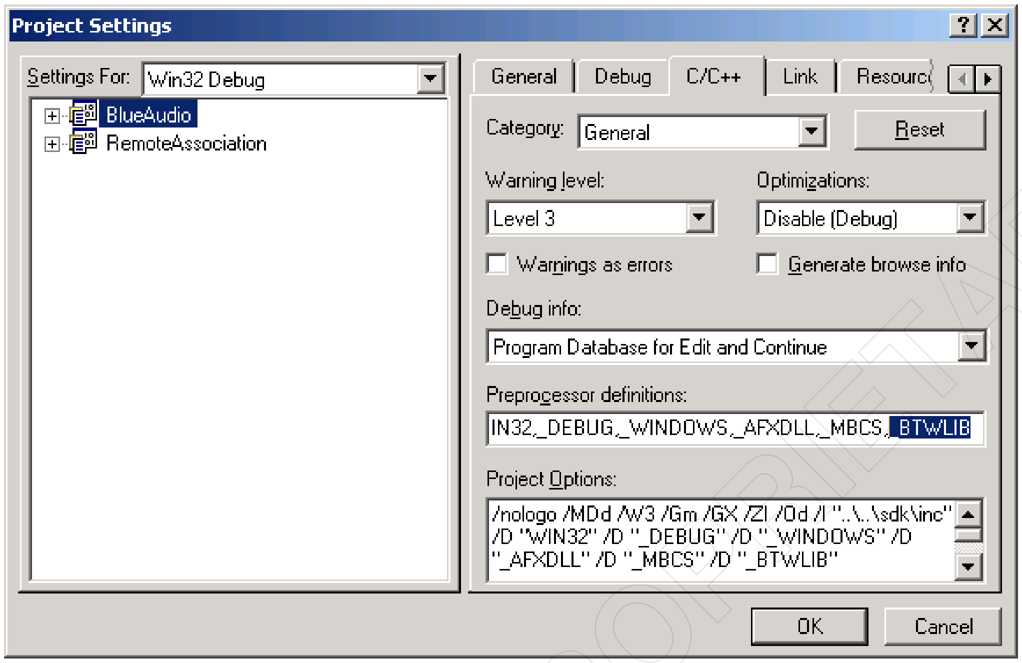
For applications built using VS2005, there are two other considerations:

1. VS2005 uses MFC and standard runtime version 8 libraries. Target platforms may not contain these libraries. Therefore, application developers must be prepared to redistribute the applicable Microsoft libraries through the Microsoft Visual C++ 2005 Redistributable packages vcredist_x86.exe and vcredist_x64.exe (supplied with VS2005) or using Merge Modules, as documented on MSDN http://msdn2.microsoft.com. These libraries are not included in the Broadcom SDK.
2. The selected `BtWdSdkLib.lib` file that must be linked uses the VS2005 definition of the `wchar` data type. BTW 5 stack software began supporting the interface to this library with this `wchar` definition at certain points in various releases, but older BTW 5 stack software will not have this support by default (all BTW 6 versions support the interface natively). A supplemental DLL has been supplied with the SDK to meet this need, `btwapi.dll`. There are two copies of this DLL, accompanying the 32 and 64-bit `BtWdSdkLib.lib` files under the appropriate directories as specified above. The application developer must be prepared to redistribute the appropriate `btwapi.dll` to target systems in `%systemroot%\system32`.

Note: Only redistribute `btwapi.dll` if it is not already present on the target system. Overwriting an existing `btwapi.dll` will cause unpredictable results.

All of the library files supplied with the SDK are built in Release configuration. To avoid potential problems in conflicting Release vs. Debug run time libraries, applications should use the following Linker setting when building debug versions: `/NODEFAULTLIB:"msvcrt.lib"`

As shown in Figure 2 and Figure 3, `BTWLIB` should be added to the preprocessor definitions in the project settings.



Category: General

Warning level: Level 3

Optimizations: Disable (Debug)

Warnings as errors

Generate browse info

Debug info: Program Database for Edit and Continue

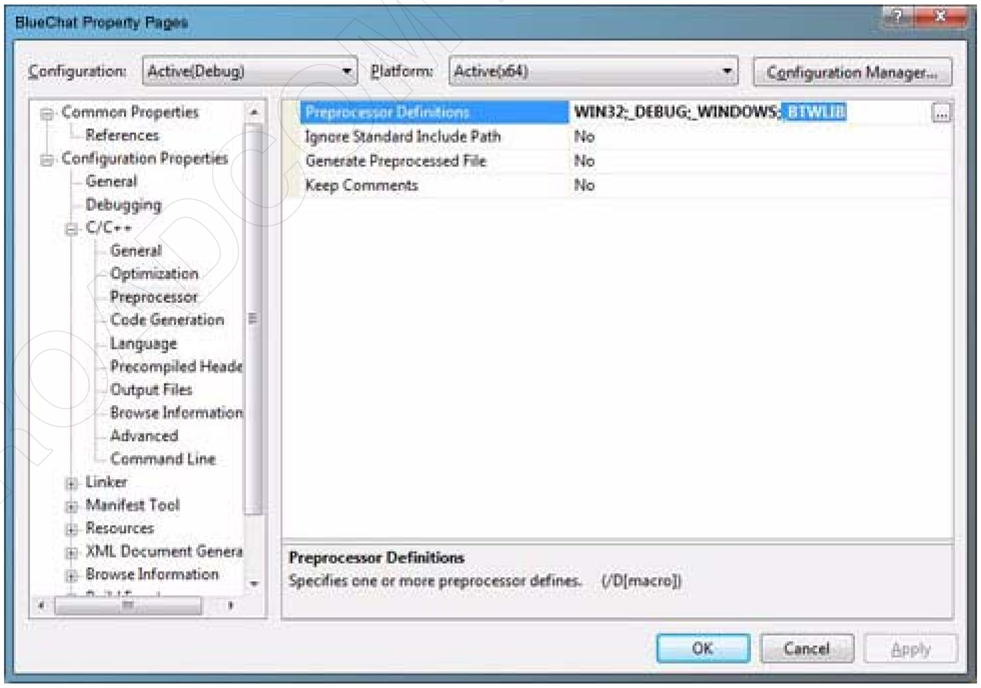
Preprocessor definitions:

IN32,_DEBUG,_WINDOWS,_AFXDLL,_MBCS,_BTWLIB

Project Options:

```
/nologo /MDd /W3 /Gm /GX /ZI /Od /I "..\..\sdk\inc"
/D "WIN32" /D "_DEBUG" /D "_WINDOWS" /D
"_AFXDLL" /D "_MBCS" /D "_BTWLIB"
```

Figure 2: Project Settings for MS VC++ 6.0



Preprocessor Definitions	WIN32;_DEBUG;_WINDOWS;_BTWLIB
Ignore Standard Include Path	No
Generate Preprocessed File	No
Keep Comments	No

Preprocessor Definitions
Specifies one or more preprocessor defines. (/D[macro])

Figure 3: Project Settings for MS VS 2005

COMPATIBILITY

This section discusses the compatibility of applications produced using the SDK when deployed against Broadcom's various WIDCOMM BTW Bluetooth software products. BTW refers to Broadcom's WIDCOMM Bluetooth software products for Windows PCs. Currently, there are two major BTW products—Broadcom's Bluetooth Software for Windows and Broadcom's Vista Profile Pack. The latest SDK release is compatible with both BTW products.

Broadcom's Bluetooth Software for Windows refers to all versions of the Broadcom/WIDCOMM Bluetooth Stack software. Previous releases are identified by version as BTW 1.x, 3.x, and 4.x. Current versions are referred to as BTW 5 and are identified by version as 5.0.x.x and 5.1.x.x. Broadcom's Vista Profile Pack is a version of the Broadcom/WIDCOMM Bluetooth software that runs on the Microsoft Bluetooth stack. It is referred to as BTW 6 and is identified by version as 6.x.x.x.

Additionally, there is Broadcom's Vista Audio Pack. This product is identified by version as 5.2.x.x. SDK compatibility with BTW version 5.2.x.x is limited, as this version only supports audio profiles. Contact Broadcom Technical Support directly at http://www.broadcom.com/products/bluetooth_support.php for detailed information on SDK and Vista Audio Pack compatibility.

SDK versions loosely follow BTW version numbers. As of the release of SDK version 6.1.0.1501, the SDK produces applications that can be deployed for Vista Profile Pack and Bluetooth Software for Windows platforms.

COMPATIBILITY WITH BROADCOM'S BLUETOOTH SOFTWARE FOR WINDOWS

BTW and SDK software versions 1.4.2.10 SP5 through the latest BTW 5 versions are designed to be forward- and backward-compatible with all combinations of SDK and BTW software, where both components are version 1.4.2.10 SP5 or greater. With the release of SDK version 6.1.0.1501, SDK software compatibility includes BTW 6 versions as well.

BTW and SDK software versions prior to 1.4.2.10 SP5 are not forward- or backward-compatible with other software versions and are deprecated. For applications built with SDK versions previous to 1.4.2.10 SP5 that now need to run on PCs with BTW 1.4.2.10 SP5 and later, Broadcom recommends rebuilding the application using the latest SDK release to avoid interface discrepancies in the profile APIs.

All SDK APIs and classes are fully supported in Bluetooth Software for Windows deployments, subject to documented deprecations and with the exceptions described in "Compatibility and New Features" on page 8.

COMPATIBILITY WITH BROADCOM'S VISTA PROFILE PACK

SDK version 6.1.0.1501 is the first SDK release compatible with the Vista Profile Pack (BTW 6). Applications built with SDK versions prior to 6.1.0.1501 are only compatible with BTW versions prior to BTW 6, whereas applications built with SDK version 6.1.0.1501 or greater are fully forward- and backward-compatible with all supported BTW versions, including BTW 6. Furthermore, the binary applications produced by SDK versions 6.1.0.1501 are capable of executing under any supported BTW version without the need for deployment-specific libraries or build configurations.

The Vista Profile Pack runs on the Microsoft Bluetooth stack. As a result, some SDK features, APIs and classes may not be fully supported by a particular BTW 6 version. Consult the latest SDK Release Notes for up to date SDK and BTW 6 API and class support details, available at http://www.broadcom.com/products/bluetooth_support.php.

Compatibility and Visual Studio 2005 Support

If an SDK application is built using Microsoft Visual Studio 2005, the application must be prepared to redistribute the Broadcom and the Microsoft runtime and interface library files to target platforms when appropriate. See "Build Environments" on page 5 for information on redistribution requirements.

Compatibility and New Features

When compatible BTW and SDK versions (as discussed earlier) are used together, the following apply:

- Applications built using an SDK version older than the BTW software version on which the application runs will run properly but may not be able to take advantage of newer features added in the more recent BTW software version.
- Applications built using an SDK version newer than the BTW software version on which the application runs will not run properly if it depends on newer features added in the more recent SDK version.

Changes to the SDK software generally consist of additions, such as new functions or new code appended to enumerated constant lists. Such changes are documented in the SDK release notes for the version to which the change applies. In addition, concise comments in the affected SDK header file identify changes and specify the BTW and SDK versions in which the change(s) occur.

Section 4: Development Kit Classes

VIRTUAL FUNCTIONS

The SDK classes provide virtual functions, where appropriate, for applications to react to Bluetooth protocol events.

For example, an application requiring an RFCOMM connection defines an application class that is derived from the CRfCommPort base class. The derived class may define derived functions that substitute for the virtual functions in CRfCommPort.

For instance, the CRfCommPort class *OnDataReceived ( )* function is called to pass an incoming data packet to the application. Applications should implement a derived version to actually receive the data packet.

Similarly, the CRfCommPort class *OnEventReceived ( )* function is called when a significant event is detected, such as a connect or disconnect.

Some virtual functions are pure – no default implementation exists in the base class, so the application *must* provide a derived function. Examples are *CSppClient::OnClientStateChange ( )* and *CLapClient::OnStateChange ( )*.

Virtual functions that are not pure have a default implementation that does nothing in the SDK base class. These functions may be of use to one application but not another. Applications can take the default if the function is not useful. Examples are *CRfCommPort::OnModemSignalChanged ( )* and *CRfCommPort::OnFlowEnabled ( )*.

CONSTRUCTORS AND DESTRUCTORS FOR DEVELOPMENT KIT CLASSES

All of the SDK classes have default constructors that accept no arguments, and return void. Any internal initializations and allocations are performed automatically.

All of the SDK classes have nontrivial destructors. Applications which instantiate SDK classes from the heap (or application classes derived from SDK classes) must delete these objects to prevent the application from leaking memory.

Note: Previous versions of the SDK required applications to call *WIDCOMMSDK_Init ( )* and *WIDCOMMSDK_Shutdown ( )* on startup and exit. This is no longer necessary, these calls are now stub functions for compatibility. Their functionality is now handled internally on CBtIf object construction and destruction.

Global Object Instantiation

Global object instantiation of any of the SDK classes is not supported in developer DLLs. Because of the way constructor code is executed during Microsoft library initializations, processes may block infinitely if SDK class objects are instantiated globally in a DLL (this includes static objects).

If global SDK class objects must be used, they must be instantiated in an application, not in a DLL. If an SDK object must reside in a DLL, then the instantiation of the object must be dynamic.

In addition, an explicit call to destroy any global SDK objects must be performed before the developer DLL is unloaded. See the MSDN documentation for the Microsoft function DllMain for information on why it is NOT appropriate to make such a call from the DllMain detach context.

Use of GUID to Represent UUID

The Bluetooth specification defines UUID values for service attributes, service classes, and protocols.

These are all 16-byte fields, for which the last 12 bytes have the same value, the Bluetooth base UUID represented in hexadecimal bytes as 0x00, 0x00, 0x10, 0x00, 0x80, 0x00, 0x00, 0x80, 0x5F, 0x9B, 0x34, 0xFB.

To save storage space and transmission bandwidth, abbreviated versions (2-byte or 4-byte format) are used, where possible, at the lower-stack layers and for over-the-air transmissions.

At the application level, the full 16-byte version is used. The SDK provides 16-byte definitions for the supported standard UUIDs.

If an application defines a new service, a new 16-byte UUID must be generated outside the SDK. The new value may then be passed to the SDK functions on the new application's server and client sides to establish a connection based on the new service.

Microsoft Visual Studio systems provide a utility program, guidgen.exe, to generate unique GUID values. The program can be executed from the Windows Explorer. It offers a choice of formats for the generated GUID, which can then be cut and pasted into your application. For example, a sequence like the one below can be cut and pasted from the guidgen.exe output window. You just substitute your variable name for *<<name>>*.

```
// {CE37CA6E-288A-409a-9796-191882EE44FC}
static GUID <<name>> =
{0xCE37CA6E, 0x288A, 0x409A, {0x97, 0x96, 0x19, 0x18, 0x82, 0xEE, 0x44, 0xFC}};
```

The SDK implements 16-byte UUIDs as the Microsoft GUID data type. These two definitions are compatible because the Bluetooth base UUID was defined as a form of GUID value.

The definition for class CBtIf, in file BtIfClasses.h, contains a list of standard GUID, with names guid_SERVCLASS_*.

For non-Microsoft contexts, the GUID may be defined as a C or C++ structure:

```
typedef struct _GUID {
   unsigned long    Data1;
   unsigned short   Data2;
   unsigned short   Data3;
   unsigned char    Data4[8];
}  GUID;
```

USE OF MAXIMUM TRANSMISSION UNIT (MTU)

Bluetooth protocol layers, such as L2CAP, RFCOMM, or OBEX, have a default MTU value. See BtIfDefinitions.h for L2CAP_DEFAULT_MTU, RFCOMM_DEFAULT_MTU, and OBEX_DEFAULT_MTU.

The MTU sets an upper limit for a single packet at the Bluetooth protocol layer being used. For instance, when side A sets an MTU value to side B, side A is saying it has an input buffer MTU bytes long. Side B must react by segmenting its transmissions, if necessary, so that a single packet is no larger than MTU bytes. The receiving side reconstructs the segmented packets into the original message size. The segmenting and reconstruction occur below the application level.

Applications can take the default or set a nondefault value at connection setup time or for L2CAP also by supporting a Reconfigure command that can be sent after connection. One reason for setting a nondefault MTU might be that the application writer knows that the hardware platform has very limited memory.

DERIVED FUNCTIONS RUN ON SEPARATE THREADS

Derived functions run in a different context than the application. Operations performed in the derived functions must be made thread-safe for the application.

In particular, if the application is using MFC, the derived functions cannot use MFC classes. The safe way to process information received in derived functions is to copy the information to a new heap-based object that can be sent to the application main thread by posting a user-defined message.

The sample applications show how to do this.

An additional implication of this guideline is that derived functions must not call back into the stack with another SDK API. Any SDK API that must be called as a result of a callback from the stack must be executed from the application main thread, not the callback execution context.

EXAMPLES (RFCOMM)

CLIENT

Common steps for a typical client application:

1. Instantiate an object derived from CBtIf and provide CBtIf functions for the virtual functions. The derived class may be a simple extension of CBtIf or a dialog class derived from both CDialog and CBtIf.
2. Use method *CBtIf::StartInquiry ( )* to obtain a list of devices in the Bluetooth neighborhood.
3. Use the derived method *CBtIf::OnDeviceResponded ( )* to build a list of responding devices.

   A derived method, *CBtIf::OnInquiryComplete ( )*, may optionally be used to determine when the inquiry process is complete.
4. Use *CBtIf::StartDiscovery ( )* to determine the services each device offers.
5. Call derived method *CBtIf::OnDiscoveryComplete ( )* when the discovery process is complete, and then call *CBtIf::ReadDiscoveryRecords ( )* to obtain a list of the services.
6. Using application-dependent criteria, select a server that provides the desired service.
7. Processing now depends on the protocol used by the service.

   For RFCOMM:

   a. An object of class CRfCommIf is needed to establish a service channel number (SCN) and security settings. The SCN is obtained from the discovery record for the selected server.

   b. A connection is established with the server, using an application object derived from class CRfCommPort. *CRfCommPort::OpenClient ( )* begins the connection process. The derived *OnEventReceived ( )* function informs the application that the connection is established.

   c. Processing is application-dependent; data is sent using the *CRfCommPort::Write ( )* function, and the derived function *CRfCommPort::OnDataReceived ( )* is called when data is received.

   d. The connection remains open until the client calls *CRfCommPort::Close ( )*. The close can be initiated by the client or can be called in response to a CONNECT_ERR event from the server.

SERVER

Common steps for a typical server application:

1. Instantiate an object of class CRfCommIf and call function *CRfCommIf::AssignScnValue ( )* to get an SCN assigned.
2. Instantiate an object of class CSdpService and call the functions *AddServiceClassIdList ( )*, *AddServiceName ( )*, *AddRFCommProtocolDescriptor ( )*, and *MakePublicBrowseable ( )* to set up the service in the local Bluetooth device.
3. Call *CRfCommIf::SetSecurityLevel ( )*.
4. *CRfCommPort::OpenServer ( )* starts the server, which then waits for a client to attempt a connection. The derived function *CRfCommPort::OnEventReceived ( )* is called when a connection is established.
5. Processing now depends on application logic. For RFCOMM:

   a. Data is sent using the *CRfCommPort::Write ( )* function. The derived function *CRfCommPort::OnDataReceived ( )* is called to receive incoming data.

   b. The connection remains open until the server calls *CRfCommPort::Close ( )*. The close can be initiated by the server or can be called in response to a CONNECT_ERR event from the client.

Section 5: Class Descriptions and Usage

CBtIf

This class provides a stack interface for device inquiry and service discovery.

An object of this class must be instantiated before any other SDK classes are used (typically at application startup). An object of this class should not be deleted until the application has finished all interactions with the stack (typically at application exit). Do not instantiate multiple CBtIf objects concurrently from the same application.

Note: Previous versions of the SDK required applications to call WIDCOMMSDK_Init and WIDCOMMSDK_Shutdown on startup and exit. This is no longer necessary, these calls are now stub functions for compatibility. Their functionality is now handled internally on CBtIf object construction and destruction.

In addition to the methods described in this section, there are two public member variables that an application may read, if desired:

- *BD_ADDR m_BdAddr* — Bluetooth device address of the local device. Call *GetLocalDeviceInfo ( )* to initialize this member variable before access. Use of this member variable is deprecated, however. Applications that have used this variable should use *CBtIf::GetLocalDeviceVersionInfo ( )* to retrieve the device address, with no need for extra initializations.
- *BD_NAME m_BdName* — Bluetooth name of the local device. Call *GetLocalDeviceInfo ( )* to initialize this member variable before access. Use of this member variable is deprecated, however. Applications that have used this variable should use *CBtIf::GetLocalDeviceName ( )* to retrieve the device name, with no need for extra initializations.

A set of standard GUID values for Bluetooth service classes is provided as public data members for this class, as listed below:

guid_SERVCLASS_SERVICE_DISCOVERY_SERVER

guid_SERVCLASS_BROWSE_GROUP_DESCRIPTOR

guid_SERVCLASS_PUBLIC_BROWSE_GROUP

guid_SERVCLASS_SERIAL_PORT

guid_SERVCLASS_LAN_ACCESS_USING_PPP

guid_SERVCLASS_PANU

guid_SERVCLASS_NAP

guid_SERVCLASS_GN

guid_SERVCLASS_DIALUP_NETWORKING

guid_SERVCLASS_IRMC_SYNC

guid_SERVCLASS_OBEX_OBJECT_PUSH

guid_SERVCLASS_OBEX_FILE_TRANSFER

guid_SERVCLASS_IRMC_SYNC_COMMAND

guid_SERVCLASS_HEADSET

guid_SERVCLASS_CORDLESS_TELEPHONY

guid_SERVCLASS_INTERCOM

guid_SERVCLASS_FAX

guid_SERVCLASS_HEADSET_AUDIO_GATEWAY

guid_SERVCLASS_PNP_INFORMATION

guid_SERVCLASS_GENERIC_NETWORKING

guid_SERVCLASS_GENERIC_FILETRANSFER

guid_SERVCLASS_GENERIC_AUDIO

guid_SERVCLASS_GENERIC_TELEPHONY

guid_SERVCLASS_BPP_PRINTING

guid_SERVCLASS_HCRP_PRINTING

guid_SERVCLASS_SPP_PRINTING

guid_SERVCLASS_HUMAN_INTERFACE

StartInquiry ( )

This function starts the Bluetooth device inquiry procedure.

Because the Bluetooth stack is multiuser, an inquiry may not start immediately when this function is called; the stack may be busy with another operation.

Until the application calls *StopInquiry ( ),* it will receive notification of all new devices found, even though the inquiry that found them was originated by a different process.

While the inquiry is in progress, the derived *OnDeviceResponded ( )* function is called each time a device responds. Typically, an application will use this function to accumulate a list of responding devices.

The application may receive more than one *OnDeviceResponded ( )* call for the same device and should discard duplicate BD addresses.

Prototype:	`BOOL StartInquiry ( );`
Parameters:	None
Returns:	TRUE if the inquiry was started. FALSE, otherwise.

StopInquiry ( )

This function stops a running inquiry.

Prototype:	`void StopInquiry ( );`
Parameters:	None
Returns:	void

virtual OnDeviceResponded ( )

This function is called for each inquiry response from each device in the Bluetooth neighborhood.

This function may trigger multiple times per inquiry – even multiple times per device – once for the address alone, and once for the address and the user-friendly name.

Prototype:

```
virtual void OnDeviceResponded (
                           BD_ADDR   bda,
                           DEV_CLASS devClass,
                           BD_NAME   bdName,
                           BOOL      bConnected);
```

Parameters:	bda	The address of the responding device.
	devClass	The class of the responding device, see BtIfDefinitions.h. devClass[0] along with highest 3 bits of devClass[1] - service class devClass[1] lowest 5 bits - major device class devClass[2] highest 6 bits - minor device class
	bdName	The user-friendly name of the responding device – this is a null-terminated string that will have length 0 when the device is reporting only its address.
	bConnected	TRUE if the responding device is currently connected to the local device.
Returns:	void	

virtual OnInquiryComplete ( )

This optional function may be called when all inquiries are complete, including obtaining the user-friendly names of the devices in the Bluetooth neighborhood.

This function supplements, but does not replace, the *OnDeviceResponded ( )* virtual method.

Prototype:

```
virtual void OnInquiryComplete (
                    BOOL   success,
                    short  num_responses);
```

Parameters:	success	TRUE if the inquiry is successful; otherwise, there is a device error.
	num_responses	The number of devices responding to the inquiry.
Returns:	void	

StartDiscovery ( )

This function requests a service discovery for a specific device. When the discovery is complete, the derived function *OnDiscoveryComplete ( )* is called.

In BTW, the discovery database is cumulative. It contains the results of all this application's previous discoveries. It also contains the discovery results of any other applications that are running or that have run.

Discovery results for a device are not removed until the device fails to respond to an inquiry.

An application can minimize the chance of unwanted discovery results by specifying a specific GUID when calling *StartDiscovery ( ).*

Prototype:

```
BOOL StartDiscovery (   BD_ADDR    p_bda,
                        GUID       *p_service_guid);
```

Parameters:

p_bda	The Bluetooth device address of the device on which the service discovery is to be performed.
p_service_guid	The GUID of the service being looked for. If this parameter is NULL, all public browseable services for the device will be reported.

Returns: TRUE if discovery started.
FALSE, otherwise.

SwitchRole ( )

An application uses this method to request that the device switch role to Master or Slave. If the application wants to accept multiple simultaneous connections, it must execute this command to request that the device switch role to Master.

For example, if a service must support multiple L2CAP connections, the server starts by allocating a PSM using an L2CapIf object. Then if *n* concurrent connections are required, the server instantiates *n* L2Cap connection objects, and calls the *Listen ( )* function for each. These objects may share a PSM value. Now the server is prepared to receive connections from *n* clients.

The *SwitchRole ( )* function is required when the client connections start arriving.

When the first client successfully connects with the server, the Bluetooth stack automatically puts the server into Bluetooth slave mode. The connection is fine; but in slave mode the server can no longer respond to service discovery requests from other clients. So the server must call *SwitchRole ( )* to restore the server to Bluetooth master mode. The connection is not disturbed; but the server can now accept additional connections.

Each time a client connection request is accepted, the server must call *SwitchRole ( ).*

Prototype:

```
BOOL SwitchRole (   BD_ADDR               bd_addr,
                    MASTER_SLAVE_ROLE     new_role);
```

Parameters:

bd_addr	The BD Address of the remote device associated with the connection to be switched.
new_role	The role to which the device should switch. Valid values are NEW_MASTER or NEW_SLAVE.

Returns: TRUE if successful.

virtual OnDiscoveryComplete ( )

This derived function is called when discovery is complete. The application can then call *ReadDiscoveryRecords ( )* to retrieve the records found.

In BTW the discovery database is cumulative. It contains the results of all this application's previous discoveries. It also contains the discovery results of any other applications that are running or that have run.

Discovery results for a device are not removed until the device fails to respond to an inquiry.

An application can minimize the chance of unwanted discovery results by specifying a specific GUID when calling *StartDiscovery ( ).*

There are two functions for *OnDiscoveryComplete ( )*, one without any parameters, the other one will return number of services found by the device and the result code from discovery.

Prototype:
```
virtual void OnDiscoveryComplete ( );
```

Parameters: None

Returns: void

Prototype:
```
virtual void OnDiscoveryComplete (
                              UINT16    nRecs,
                              long      lResultCode);
```

Parameters:	nRecs	The number of services discovered.
	lResultCode	Result code for the discovery operation: WBT_SUCCESS if discovery completed, else one of the error codes in com_error.h
Returns:	void	

GetLastDiscoveryResult ( )

The application can use this method after notification from the *OnDiscoveryComplete ( )* callback to find out the discovery results.

Prototype:

```
CBtIf::DISCOVERY_RESULT GetLastDiscoveryResult (
                                        BD_ADDR p_bda,
                                        UINT16  *p_num_recs);
```

Parameters:

p_bda	Address of a 6-byte array to receive the Bluetooth device address of the device for which the service discovery has been performed.
p_num_recs	Address of a variable to receive the number of service records obtained during the service discovery.

Returns: One of the CBtIf::DISCOVERY_RESULT codes described in Table 3.

Table 3: DISCOVERY_RESULT

DISCOVERY_RESULT Value	*Meaning*
DISCOVERY_RESULT_SUCCESS	Discovery has completed successfully.
DISCOVERY_RESULT_CONNECT_ERR	Could not connect to remote device.
DISCOVERY_RESULT_CONNECT_REJ	Remote device rejected the connection.
DISCOVERY_RESULT_SECURITY	Security (authentication) failed.
DISCOVERY_RESULT_BAD_RECORD	Remote Service Record Error.
DISCOVERY_RESULT_OTHER_ERROR	Other error.

ReadDiscoveryRecords ( )

This function is called when discovery is complete to retrieve the records received from the remote device.

In BTW the discovery database is cumulative. It contains the results of all previous discoveries of this application. It also contains the discovery results of any other applications that are running or that have run.

Discovery results for a device are not removed until the device fails to respond to an inquiry.

An application can minimize the chance of unwanted discovery results by specifying a specific GUID when calling *StartDiscovery ( ).*

Discovery records are of class CSdpDiscoveryRec, described later in this document.

Prototype:

```
int ReadDiscoveryRecords (
                        BD_ADDR             p_bda,
                        int                 max_size,
                        CSdpDiscoveryRec    *p_list,
                        GUID                *p_guid_filter = NULL);
```

Parameters:	p_bda	The BD address of the device for which records are to be read.
	max_size	The maximum number of discovery records to read.
	p_list	The place to store the records. The list must have the capacity for *max_size* entries of class CSdpDiscoveryRec.
	p_guid_filter	An optional pointer to a GUID filter. If this is set, only record(s) that have a service class ID matching this GUID are returned (typically only one record will match).
Returns:	The number of discovery records read.	

BondEx( )

BondEx() initiates the pairing procedure with the specified device. This function blocks for up to 1 minute while the security functions on the lower levels perform the pairing procedure. It returns failure if called while another call to BondEx() is in progress.

The purpose of pairing is to create a relation between two Bluetooth devices based on a common link key (a bond). The link key is created and exchanged during the pairing procedure and is expected to be stored by both Bluetooth devices for future authentication use.

A temporary connection is set up by the stack server to authenticate the remote device. If the authentication is successful, the devices are paired. Once paired, future connections for services are automatically authenticated by the stack server security logic, using the link key without intervention by the application.

BondEx() is a replacement for the deprecated Bond() function. BondEx() supports Legacy Bluetooth 2.0 pin code pairing, as well as the following Bluetooth 2.1 Secure Simple Pairing methods:

- Numeric Comparison
- Passkey

Out-of-Band (OOB) pairing is not supported at this time.

BondEx() takes an optional user-supplied callback function as an parameter. The callback function is executed when the pairing process requires information from the application or when it wants to provide information to the application, depending on the pairing method (see Table 4) in use. The stack may require a response to the callback through the BondReply() function, again depending on the pairing method in use.

Table 4: Supported Pairing Processes

Pairing Process	*Description*
Legacy pin code pairing	The stack will call the user callback in order to notify the application of Legacy pin code pairing. The application must respond through BondReply() to supply the pin code to use.
Numeric Comparison pairing	The stack will call the user callback in order to notify the application of Bluetooth 2.1 SSP Numeric Comparison pairing. The callback provides the number to display to the user to confirm both pairing entities display the same number. The application must respond through BondReply() to confirm or deny the operation.
Passkey pairing	The stack will call the user callback in order to notify the application of the passkey to display to the user to instruct the user to type the passkey on the remote device. This method is used to pair devices that have input capabilities but no display capabilities. This is notification only, the pairing process will proceed when the remote devices sends the passkey digits. The application does not respond through BondReply() for Passkey pairing.

It is important to note that the SDK application does not have any control over which pairing method is in use. The choice of pairing method is determined by the host stack, after conducting capability negotiations with the remote device.

If the application does not supply a callback function, the native stack user interface pairing process is started. This process does not interact with the calling application until the call returns with the status from the native stack user interface pairing process.

Prototype:

```
BOND_RETURN_CODE BondEx( BD_ADDR   bda,
                         tBond_CB  *BondCb,
                         void      *user_data);
```

Parameters:	bda	The BD address of the device to pair with.
	BondCb	An optional pointer to a user callback function. If supplied, it will be called to identify the pairing method in use and whether the application needs to respond through BondReply(). If no callback supplied, the native stack user interface will be invoked for pairing.
	user_data	An optional pointer to user supplied data. If present, it will be returned to the application in any calls to BondCb.
Returns:	SUCCESS	Everything is OK.
	BAD_PARAMETER	A parameter was invalid.
	NO_BT_SERVER	Cannot access the local Bluetooth COM server.
	FAIL	If timeout or rejection by other device.
	REPEATED_ATTEMPTS	Device reports too many failed attempts to bond, will continue until after device security timeout.

The tBond_CB callback function is defined in BtIfClasses.h as:

Prototype:

```
typedef void (tBond_CB)   (eBOND_CB_EVENT   eEvent,
                           void             *user_data,
                           UINT32           event_data);
```

Parameters:

eEvent	Identification of the pairing event as one of the following:
	• BOND_EVT_PIN_CODE_REQ — Legacy pin code pairing notification.
	• BOND_EVT_CONFIRM_REQ — Numeric Comparison pairing notification.
	• BOND_EVT_PASSKEY_REQ — Passkey pairing notification.
user_data	Pointer to user data optionally supplied in the BondEx() call.
event_data	For Numeric Comparison and Passkey pairing, this is the number to display to the user. Not used for Legacy pairing.

Returns: None

BondReply( )

BondReply() allows the user application to send pairing response data. It should only be called in response to a tBond_CB notification for the Numeric Comparison or Passkey pairing methods.

Prototype:

```
void BondReply (eBOND_REPLY   reply,
                UINT32        nPinLength=0,
                UCHAR         *szPin=NULL);
```

Parameters:

reply	Enumerated reply type as one of the following:
	• BOND_CONFIRM_ALLOW — Confirm Numeric Comparison validated.
	• BOND_CONFIRM_DISALLOW — Reject Numeric Comparison request.
	• BOND_PIN_ALLOW — Legacy pairing allow, pin code sent.
	• BOND_PIN_DISALLOW — Reject Legacy pin code request.
nPinLength	For BOND_PIN_ALLOW reply, length in characters of pin code supplied in szPin. Valid range: 0-16
szPin	For BOND_PIN_ALLOW reply, pin code to use for legacy pairing. This is an array of BT_CHAR, null terminated.

Returns: None

Bond ( )

Note: The Bond() function is deprecated. It is being maintained for compatibility with existing applications, but it is strongly recommended that SDK applications use the new BondEx() API (see "BondEx( )" on page 19).

BondEx() addresses incompatibilities with the deprecated Bond() method when running on Bluetooth 2.1 capable host platforms, adds support for Bluetooth 2.1 Secure Simple Pairing (SSP) methods, and adds functionality to allow SDK applications to directly invoke the native stack user interface pairing process, if desired.

The deprecated Bond() method functions when either one or both pairing entities cannot support Bluetooth 2.1 SSP methods Numeric Comparison or Passkey. But when both pairing entities can support either Numeric Comparison or Passkey SSP methods, Bond() cannot function correctly.

This function initiates the pairing procedure with the specified device. This function will block for up to 1 minute while the security functions at the lower levels perform the pairing procedure.

Prototype:

```
BOND_RETURN_CODE Bond (
                        BD_ADDR  bda,
                        BT_CHAR  *pin_code);
```

Parameters:	bda	The BD address of the device to which to pair.
	pin_code	The PIN code to use for the pairing. This is an array of BT_CHAR, null terminated. A code with the length >= PIN_CODE_LEN (= 16) is invalid.
Returns:	SUCCESS	Everything is OK.
	BAD_PARAMETER	PIN code is NULL or too long.
	NO_BT_SERVER	Cannot access the local Bluetooth COM server.
	FAIL	If timeout or rejection by other device.
	REPEATED_ATTEMPTS	Device reports too many failed attempts to pair, will continue until after device security timeout.

BondQuery ( )

This function tests if the indicated device is already paired with the local device.

Prototype: `BOOL BondQuery (BD_ADDR bda);`

Parameters:	bda	The BD address of the device to query.

Returns: TRUE if the devices are already paired; FALSE, otherwise.

UnBond ( )

This function deletes the bond between the devices.

Prototype: `BOOL UnBond (BD_ADDR bda);`

Parameters:	bda	The BD address of the device with which the bond is being severed.

Returns: TRUE if the devices are now unbonded; FALSE, otherwise.

IsDeviceReady ( )

This function verifies that the application is connected to the stack server, and that the stack server and Bluetooth device are ready to accept commands.

Prototype: `BOOL IsDeviceReady ( );`

Parameters: None

Returns: TRUE, if the device is ready; FALSE, otherwise.

IsStackServerUp ( )

This function checks to see if the Bluetooth stack server is up.

Prototype:	`BOOL IsStackServerUp ( );`
Parameters:	None
Returns:	TRUE, if the stack is up; FALSE, otherwise.

virtual OnStackStatusChange ( )

This is a virtual function. It is used to indicate that the status of the stack server has changed. Applications may provide a derived method to process the status change notice.

Prototype:	`virtual void OnStackStatusChange (CBtIf::STACK_STATUS new_status);`	
Parameters:	new_status	One of the values described in Table 5.
Returns:	void	

Table 5: STACK_STATUS

STACK_STATUS Value	*Meaning*
DEVST_DOWN	Device is present, but down (maybe being removed).
DEVST_UP	Device is present and up.
DEVST_ERROR	Device is in error.
DEVST_UNLOADED	Stack is being unloaded.
DEVST_RELOADED	Stack reloaded after being unloaded.

GetExtendedError ( )

This function returns the object's last-error code. The last-error code is maintained on a per-object basis. Multiple objects do not overwrite each other's last-error code.

This function is intended for obtaining extended error information when the SDK return/result codes are not sufficient. This is useful when the UNKNOWN_RETURN_ERROR code is returned from an SDK API function, or the UNKNOWN_RESULT_ERROR code is returned in an SDK callback function.

To retrieve an extended error code of a method of this class in the case where the method has failed, call *GetExtendedError ( )* directly after this method has been called. *GetExtendedError ( )* can be called in callback functions of this class as well.

Prototype:	`WBtRc GetExtendedError ( );`
Parameters:	None
Returns:	See definition of the WBtRc enumeration in com_error.h

SetExtendedError ( )

This function sets the last-error code for the object of this class. *SetExtendedError ( )* is intended to localize the place where exactly an error occurs in the code. It can be called to reset the last-error code for an object of this class to WBT_SUCCESS (see com_error.h) before any other class method is called. After a class method has been called, *GetExtendedError ( )* can be invoked to read the error information. If *GetExtendedError ( )* reports WBT_SUCCESS, then the error occurs somewhere else in the program.

The last-error code is kept in a local variable of the object so that multiple objects of this class do not overwrite each other's values.

Prototype:	`void SetExtendedError (WBtRc code);`	
Parameters:	code	One of the values from the WBtRc enumeration defined in com_error.h
Returns:	void	

GetLocalDeviceInfo ( )

This method obtains the Bluetooth device address and name of the local device. The address is stored in public variable m_BdAddr, and the name in m_BdName. Use of these member variables is deprecated – see "GetLocalDeviceName ( )" on page 24 and "GetLocalDeviceVersionInfo ( )" on page 26 instead.

Prototype:	`BOOL GetLocalDeviceInfo ( );`
Parameters:	None
Returns:	TRUE if the device information was obtained successfully; FALSE, otherwise.

GetLocalDeviceName ( )

This function retrieves the local device name to the caller's buffer. The name is obtained from the Host Computer Interface the first time this function is called. Subsequent calls retrieve the name from an internal buffer.

Prototype:	`BOOL GetLocalDeviceName (BD_NAME *bdName);`	
Parameters:	bdName	Pointer to caller's buffer
Returns:	TRUE if the name was successfully retrieved; FALSE, otherwise.	

SetLocalDeviceName ( )

This function has been disabled.

GetRemoteDeviceInfo ( )

This function queries the registry for information relating to previously discovered devices. You will have to call *GetNextRemoteDeviceInfo ( )* to get next device info.

Prototype:
```
REM_DEV_INFO_RETURN_CODE GetRemoteDeviceInfo (
                                        DEV_CLASS     filter_dev_class,
                                        REM_DEV_INFO  *p_rem_dev_info);
```

Parameters:	filter_dev_class	A 3-byte array input parameter. It determines which device classes are to be returned. See BtifDefinitions.h for all defined device classes. filter_dev_class[0] is currently not used filter_dev_class[1] is for MAJOR_DEV_CLASS (lowest 5 bits only) filter_dev_class[2] is for MINOR_DEV_CLASS (highest 6 bits only)
	p_rem_dev_info	A pointer to a caller-supplied buffer where the remote device info is to be placed. See the REM_DEV_INFO definition below.
Returns:	See REM_DEV_INFO_RETURN_CODE definitions in Table 6 on page 25.	

```
typedef struct
{
   BD_ADDR       bda;
   DEV_CLASS     dev_class;
   BD_NAME       bd_name;
   BOOL          b_paired;
   BOOL          b_connected;
} REM_DEV_INFO;
```

Table 6: REM_DEV_INFO_RETURN_CODE

REM_DEV_INFO_RETURN_CODE Value	***Meaning***
REM_DEV_INFO_SUCCESS	Discovery has completed successfully.
REM_DEV_INFO_EOF	No more devices found.
REM_DEV_INFO_ERROR	Cannot find existing entry, or *GetNextRemoteDeviceInfo ( )* called without calling *GetRemoteDeviceInfo ( )* first.
REM_DEV_INFO_MEM_ERROR	Cannot allocate memory for temp data.

GetNextRemoteDeviceInfo ( )

This is called after calling *GetRemoteDeviceInfo ( )* to read the next device in the registry.

Prototype:
```
REM_DEV_INFO_RETURN_CODE GetNextRemoteDeviceInfo (REM_DEV_INFO *p_dev_info);
```

Parameters:	p_dev_info	A pointer to a caller-supplied buffer where the remote device info is to be placed. See the definition for REM_DEV_INFO above.
Returns:	See the definition for REM_DEV_INFO_RETURN_CODE in Table 6 on page 25.	

GetLocalDeviceVersionInfo ( )

This function reads the values for the version information for the local Bluetooth device.

Prototype:	`BOOL GetLocalDeviceVersionInfo (DEV_VER_INFO *p_Dev_Ver_Info);`
Parameters:	p_Dev_Ver_Info See DEV_VER_INFO definition below.
Returns:	TRUE, if the information is returned; FALSE, otherwise.

The device version information record is defined by the DEV_VER_INFO structure found in the BtIfClasses.h header file:

```
typedef struct
{
    BD_ADDR   bd_addr;
    UINT8     hci_version;
    UINT16    hci_revision;
    UINT8     lmp_version;
    UINT16    manufacturer;
    UINT16    lmp_sub_version;
} DEV_VER_INFO;
```

bd_addr – The bluetooth device address.
hci_version – HCI version.

0x00 Bluetooth HCI Specification 1.0b
0x01 Bluetooth HCI Specification 1.1
0x02 Bluetooth HCI Specification 1.2
0x03 Bluetooth HCI Specification 2.0
0x04-0xFFF Reserved for future use

hci_revision – Revision of the current HCI in the Bluetooth hardware
lmp_version – Version of the current LMP in the Bluetooth hardware
manufacturer – Manufacturer name of the Bluetooth hardware (see Table 7 on page 27 for list of manufacturers)
lmp_subversion – Sub-version of the current LMP in the Bluetooth hardware

Note: Table 7 on page 27 only lists the first few Bluetooth hardware manufacturers. For the complete list, go to the Bluetooth Special Interest Group website at: https://www.bluetooth.org/foundry/assignnumb/document/company_identifiers.

Table 7: Bluetooth Hardware Manufacturer’s List

0	Ericsson®
1	Nokia®
2	Intel®
3	IBM®
4	Toshiba®
5	3Com®
6	Microsoft
7	Lucent®
8	Motorola®
9	Infineon®
10	CSR
11	Silicon Wave™
12	DigiAnswer
13	Texas Instruments®
14	Parthus
15	Broadcom

GetRemoteDeviceVersionInfo ( )

This function will read the values for the version information of the remote Bluetooth device.

Prototype:

```
BOOL GetRemoteDeviceVersionInfo (
                                    BD_ADDR         bd_addr_remote,
                                    DEV_VER_INFO    *p_Dev_Ver_Info);
```

Parameters:

bd_addr_remote	The Bluetooth address of the remote device.
p_Dev_Ver_Info	A pointer to a buffer where the remote device information will be placed. See DEV_VER_INFO definition in “GetLocalDeviceVersionInfo ( )” on page 26. This function only fills in the following fields of the structure: • lmp_version • manufacturer • lmp_sub_version

Returns: TRUE, if the information is returned; FALSE, otherwise.

Note: The connection has to be made before calling this function.

GetDKVersionInfo ( )

This method is used to get SDK version information.

Prototype:

```
BOOL GetDKVersionInfo (
                          BT_CHAR *p_version_info,
                          int     nSize);
```

Parameters:	p_version_info	The buffer to hold version string.
	nSize	The size of p_version_info. The maximum buffer size used by this API is MAX_PATH.

Returns: Values are:

FALSE – cannot get version info.

TRUE – version info returned.

GetBTWVersionInfo ( )

This method is used to get BTW version information.

Prototype:

```
BOOL GetBTWVersionInfo (
                          BT_CHAR     *p_version_info,
                          int         nSize);
```

Parameters:	p_version_info	The buffer to hold version string.
	nSize	The size of p_version_info. The maximum buffer size used by this API is MAX_PATH.

Returns: Values are:

FALSE – cannot get version info.

TRUE – version info returned.

GetLocalServiceName ( )

This function is used to query to the local Bluetooth Stack for registered services. Call *GetNextLocalSerivceName ( )* to get the next registered service.

Note: This only returns native stack service records, not records created from SDK applications.

Prototype:

```
BOOL GetLocalServiceName (
                           BT_CHAR   *p_ServiceName,
                           int       iBuffLen);
```

Parameters:	p_ServiceName	The buffer to hold local service name.
	iBuffLen	Size of the p_ServiceName buffer.

Returns: TRUE, if successful; FALSE, otherwise.

GetNextLocalServiceName ( )

This function is used to get the next registered service. *GetLocalServiceName ( )* must be called before calling this function.

Prototype:	`BOOL GetNextLocalServiceName ( BT_CHAR *p_ServiceName, int iBuffLen);`	
Parameters:	p_ServiceName	The buffer to hold local service name.
	iBuffLen	Size of the p_ServiceName buffer.
Returns:	TRUE, if successful; FALSE, otherwise.	

Add_Printer ( )

This function creates the HCRP port for a Bluetooth printer device and installs the printer attached to it.

Prototype:	`BOOL Add_Printer ( LPSTR PortName, BD_ADDR bd_Addr);`	
Parameters:	PortName	The HCRP port name to be created.
	bd_Addr	Printer Bluetooth Address.
Returns:	TRUE if successful; FALSE, otherwise.	

Remove_Printer ( )

This function removes all of the printers attached to this Bluetooth device.

Prototype:	`BOOL Remove_Printer (BD_ADDR bd_Addr);`	
Parameters:	bd_Addr	Printer Bluetooth Address.
Returns:	TRUE if successful; FALSE, otherwise.	

CreateCOMPortAssociation ( )

This function requests that an association between a COM port and a service on the remote device be created. This method will find an unused COM port or will create a new COM port, then associate that COM port with the indicated service on the indicated remote device. This association will cause the Bluetooth stack to automatically establish a connection to the indicated service on the remote device whenever an application opens the associated COM port. This association will be in effect for all users that log on to the PC where the association is created.

Prototype:

```
BOOL CreateCOMPortAssociation (
                                    BD_ADDR   bda,
                                    GUID      *p_guid,
                                    LPCSTR    szServiceName,
                                    USHORT    mtu,
                                    BYTE      SecurityID,
                                    BYTE      SecurityLevel,
                                    USHORT    uuid,
                                    USHORT    *p_com_port);
```

Parameters:

bda	The server's Bluetooth device address.
p_guid	(Optional). GUID of the service. If NULL is specified BTW will search for the Bluetooth Serial Port Profile {0x1101, 0, 0x1000, 0x80, 0x00, 0x00, 0x80, 0x5F, 0x9B, 0x34, 0xFB} on the remote device.
szServiceName	The service name from the discovery record obtained for the SPP server.
mtu	(Optional). Maximum Transmission unit to be used in the RFCOMM session. If 0 is specified, value of 660 is used.
SecurityID	(Optional). Security identifier used for the connection. Default value is #define BTM_SEC_SERVICE_SERIAL_PORT 1.
SecurityLevel	Defines security procedures required during the connection. The value is a bit-wise combination of: #define BTM_SEC_OUT_AUTHENTICATE 0x0010 /* Outbound call requires authentication */ #define BTM_SEC_OUT_ENCRYPT 0x0020 /* Outbound call requires encryption */ ***Note:*** BTM_SEC_OUT_ENCRYPT cannot be set alone; have to have BTM_SEC_OUT_AUTHENTICATE set.
uuid	(Optional). UUID of the service to be connected. If 0 is specified, BTW will use properties of the Bluetooth Serial Port Profile (0x1101).
p_com_port	Pointer to buffer to store COM port assigned. If the value is zero, no empty COM port is available. BTW limits maximum of 127 ports.

Returns: TRUE – Everything is OK.
FALSE – No COM port is available.

Associations should be created only for the connections that require Bluetooth search and RFCOMM connection establishment. They cannot be used for the server type connections when a COM port is opened to accept incoming connections.

If authentication or encryption is required for the connection, or may be required by the peer device, Bluetooth pairing should be performed before association is done.

After association is performed, every time an application opens the COM port, the Bluetooth stack will attempt to discover the specified service on the device and establish the RFCOMM connection.

RemoveCOMPortAssociation ( )

This function requests the the existing association between a COM port and a remote device be removed.

Prototype:	`BOOL RemoveCOMPortAssociation (USHORT com_port);`	
Parameters:	com_port	COM port to be removed.
Returns:	TRUE – Everything is OK. FALSE – COM port could not be removed.	

ReadCOMPortAssociation ( )

This method returns a list of Remote Association records that were created using the *CreateCOMPortAssociation ( )* method.

Prototype:

```
BOOL ReadCOMPortAssociation (
                          tBT_REM_ASSOC_REC *pBuffList,
                          DWORD             dwBuffSize,
                          DWORD             *pdwRequiredSize);
```

Parameters:	pBuffList	Pointer to a buffer that will receive the list of Remote Association records.
	dwBuffSize	Supplies the length, in bytes, of pBuffList.
	pdwRequiredSize	Count of COM port association records returned in pBuffList.
Returns:	TRUE – Everything is OK. FALSE – Internal system error.	

The Remote Association record is defined by the structure tBT_REM_ASSOC_REC found in the header file BtIfDefinitions.h:

```
typedef struct
{
   BD_ADDR   bda;
   GUID      guid;
   char      szServiceName[BT_MAX_SERVICE_NAME_LEN];
   short     com_port;
} tBT_REM_ASSOC_REC;
```

SetLinkSupervisionTimeOut ( )

This function sets the link supervision timeout value for the connected device. The timeout value is used by the master or slave Bluetooth device to monitor link loss. The same timeout value is used for both SCO and ACL connections for the device. An ACL connection has to be established before calling this function. The Zero value for timeout will disable the Link_Supervision_Timeout check for the connected device. The default timeout is 20 seconds.

Prototype:

```
BOOL SetLinkSupervisionTimeOut (
                                BDADDR     BdAddr,
                                UINT16     timeout);
```

Parameters:	bdAddr	The remote BD address.
	timeout	The timeout period in number of slots. Each slot is .625 milliseconds. ***Note:*** Zero will disable Link_Supervision_Timeout check.
Returns:	TRUE, if successful; FALSE, otherwise.	

SendVendorSpecific_HCICMD ( )

This function sends vendor-specific HCI commands.

Prototype:

```
SENDVENDOR_HCICMD_RETURN_CODE SendVendorSpecific_HCICMD (
                                                  UINT16    OpCode,
                                                  UINT8     *pInParam,
                                                  UINT8     InParamLen,
                                                  UINT8     *pOutBuff,
                                                  UINT8     outBuffLen);
```

Parameters:

opCode	Vendor-specific HCI command Op_Code.
pInParam	Vendor-specific HCI command parameters.
InParamLen	Length of all the input parameters.
pOutBuff	A pointer to a buffer for the return data from the call. Caller must allocate the memory before the call.
outBuffLen	Length of the buffer at pOutBuff.

Returns:

SENDVENDOR_HCICMD_SUCCESS
SENDVENDOR_HCICMD_FAILURE
SENDVENDOR_HCICMD_UNKOWN_PLATFORM — Platform is unknown.
SENDVENDOR_HCICMD_NO_SUPPORTED — The HCI command being sent is not supported.
SENDVENDOR_HCICMD_BTRKNL_NOT_OPENED — Bluetooth kernel did not open.
SENDVENDOR_HCICMD_BTKRNL_FAILURE — Unable to open or talk to btkrnl.sys.
SENDVENDOR_HCICMD_DEVICE_FAILURE — Unable to open or talk to the device.
SENDVENDOR_HCICMD_NO_RESOURCE — Unable to allocate memory.
SENDVENDOR_HCICMD_BUFFER_TOO_BIG — Input buffer is larger than max_command for HCI command (BTM_MAX_VENDOR_SPECIFIC_LEN, see BtIfDefinitions.h).
SENDVENDOR_HCICMD_BUFFER_TOO_SMALL — Output buffer is too small to hold return data.
SENDVENDOR_HCICMD_INVALID_PARAM — Invalid parameters passed to the function.

SetSniffMode ( )

This method is used to attempt to configure HCI sniff mode for a connection. Sniff mode may not be successfully set if the remote device does not support sniff mode. Use *ReadLinkMode ( )* to ensure successful setting.

Prototype: `BOOL SetSniffMode (BD_ADDR bd_Addr);`

Parameters: bd_Addr — The address of the remote device.

Returns: TRUE — the sniff mode setting is successfully initiated.
Otherwise, FALSE.

CancelSniffMode ( )

This method is used to exit HCI sniff mode on a connection.

Prototype: `BOOL CancelSniffMode (BD_ADDR bd_Addr);`

Parameters: bd_Addr — The address of the remote device.

Returns: TRUE — the command to exit sniff mode is initiated successfully.
Otherwise, FALSE.

ReadLinkMode ( )

This method is used to return the link mode of a connection to a connected device.

Prototype:	`BOOLEAN ReadLinkMode (BD_ADDR bd_addr,`
	`                       UINT8   *pMode);`	

Parameters:	bd_addr	The address of the remote connected device.
	pMode	A pointer to the current link mode of the connection to the remote device. Valid only if the function returns TRUE. May be one of the following values (see LINK_MODE definition in BtIfDefinitions.h): • LINK_MODE_NORMAL • LINK_MODE_HOLD • LINK_MODE_SNIFF • LINK_MODE_PARK
Returns:	TRUE — if success Otherwise, FALSE	

IsRemoteDeviceConnected ( )

This function queries the ACL link status of a remote device.

Prototype:	`BOOL IsRemoteDeviceConnected (BD_ADDR bd_addr_remote);`	
Parameters:	bd_addr_remote	Bluetooth Device address to query.
Returns:	TRUE, if there is an active ACL connection to the remote device. Otherwise, FALSE.	

RegisterForLinkStatusChanges ( )

This function registers a callback to be executed when the ACL link connection database changes. This allows an application to track connection status. There can only be one callback registered at a time, and any previously registered callback must be cleared before registering a new callback.

Prototype:

```
BOOL RegisterForLinkStatusChanges (
                                tLINK_STATUS_CB      *p_link_status_cb,
                                BD_ADDR              bda);
```

Parameters:	p_link_status_cb	The pointer to the callback function to register. If this parameter is NULL, any previous registration will be cleared.
	bda	Bluetooth Device address filter. If this is set to 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, all device status changes will be reported, otherwise only the specified device will generate status change events.
Returns:	TRUE, if successful FALSE, otherwise.	

The tLINK_STATUS_CB callback function is defined in BtIfClasses.h as:

Prototype:	`typedef void (tLINK_STATUS_CB) ( BD_ADDR remote_bda, BOOL bLinkUp);`	
Parameters:	remote_bda	The Bluetooth Device address of the device that changed status.
	bLinkUp	TRUE if the link status is up, or FALSE if the link status is down.
Returns:	void	

CONNECTION STATISTICS — tBT_CONN_STATS

The common tBT_CONN_STATS structure is used by the *GetConnectionStats ( )* API implemented in many of the SDK classes. It is defined in header file BtIfDefinitions.h as:

```
typedef struct
{
    UINT32   bIsConnected;
    INT32    Rssi;
    UINT32   BytesSent;
    UINT32   BytesRcvd;
    UINT32   Duration;
} tBT_CONN_STATS;
```

Where:

- bIsConnected – 0 means not connected; any other value means connected.
- Rssi – Returned Signal Strength Indicator. Value 0 indicates that the connected Bluetooth devices are at optimal separation, the *golden zone*. Increasingly positive values are reported as the devices are moved closer to each other. Increasingly negative values are reported as the devices are moved apart. Rssi ranges from -128 to 127.
- BytesSent – Total bytes sent since the connection was established. This is a count of the bytes sent by the application. Bytes added by the protocol layers are not counted.
- BytesRcvd – Total bytes received since the connection was established. This is a count of the bytes received by the application. Bytes added by the protocol layers are not counted.
- Duration – Elapsed time since the connection was established, in seconds.

GetConnectionStats ( )

This function retrieves current connection statistics. For this class, it is only returning Rssi value.

Prototype:	`BOOL GetConnectionStats ( BD_ADDR bda, tBT_CONN_STATS *p_conn_stats);`	
Parameters:	bda	BD address of connected remote device.
	p_conn_stats	A pointer to the user's connection statistics structure; see above.
Returns:	FALSE if a connection attempt has not been initiated; TRUE otherwise.	

AUDIO CONNECTIONS

Audio connections can be established between Bluetooth devices providing there already exists a data connection, such as L2CAP or RFCOMM, between the devices. The *CreateAudioConnection ( ), RemoveAudioConnection ( ), OnAudioConnected ( )* and *OnAudioDisconnect ( )* methods are available for managing audio connections. However, it is strongly recommended that these methods not be used directly by applications, but rather that applications use the connection-specific methods defined in the CL2CapConn, CRfCommPort, CObexClient and CObexServer classes. These methods are each associated with a data connection. Using these data-connection-specific methods makes it easier for the application to manage the connection than using the methods defined in the CBtIf class.

Methods that manage audio connections may return an enumerated type *AUDIO_RETURN_CODE*.

Table 8: Audio Return Codes

AUDIO_RETURN_CODE Value	***Meaning***
AUDIO_SUCCESS	Operation initiated without error.
AUDIO_UNKNOWN_ADDR	There is no data (ACL) connection active.
AUDIO_BUSY	Another audio connection is active to the requested remote device. If another SCO is being set up to the same BD address and remote device is still in the connecting state.
AUDIO_NO_RESOURCES	The maximum number of audio connections is already active.
AUDIO_ALREADY_STARTED	An audio connection to the requested remote device is already active. NOT USED for now. Return AUDIO_SUCCESS in this case.
AUDIO_UNKNOWN_ERROR	An unknown or internal error has occurred.
AUDIO_INVALID_PARAM	One of the function parameters passed is invalid.
AUDIO_MISMATCH_ADDR	A connection attempt for a specific Bluetooth device address does not match the currently connected device.
AUDIO_INVALID_HANDLE	The handle storage location passed to *CreateAudioConnection ( )* was not initialized.
AUDIO_MODE_UNSUPPORTED	The controller version is not BT1.2 or later or does not support eSCO.
AUDIO_WRONG_MODE	There is no connection with a peer device or the audio handle does not refer to a valid audio connection.

CreateAudioConnection ( )

This method is used to establish an audio connection with a remote device. The same method is used as both a client initiating a connection and a server listening for an incoming connection. When used as a client, the Bluetooth device address of the remote device must be supplied. When used as a server, the Bluetooth device address of the remote device is optional. If an address is supplied, a connection will be accepted from that device only. If the parameter is the all-Fs wildcard (0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF), a connection will be accepted from any device.

If used as a client to initiate an audio connection, a data connection must be established with the remote device prior to attempting to initiate an audio connection to that remote device.

If used as a server to listen for an incoming audio connection, a data connection does not need to be established prior to making this request. However, a data connection must be established with a remote device before an audio connection will be accepted from that remote device.

The application should be aware that if another application has already attempted to open an audio connection, this method will fail if the application attempts to open an audio connection using the same Bluetooth device address. This restriction applies when both applications try to open a connection with a specific Bluetooth device address, or if both applications try to open a connection with the all-Fs wildcard as the Bluetooth device address.

In addition, applications should not expect to manage listening connections for standard audio profile GUIDs such as Hands-free Profile. The BTW audio services are typically already in charge of listening for the standard profile GUIDs. If an application requires opening listening connections for a standard audio profile GUID, contact Broadcom Technical Support at http://www.broadcom.com/products/bluetooth_support.php.

The application must implement *OnAudioConnected ( )* to get the connection establishment notice.

Prototype:

```
AUDIO_RETURN_CODE CreateAudioConnection (
                                         BD_ADDR   bda,
                                         BOOL      bIsClient,
                                         UINT16    *audioHandle);
```

Parameters:	bda	The Bluetooth device address of the remote device. {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF} (when used as a server) indicates that a connection from any device will be accepted. By default, the BTW Headset service is waiting on the 0xFF wildcard BD_ADDR. Headset service needs to be disabled if this function is used to accept a connection from any device.
	bIsClient	TRUE if this is a client connection (initiator); otherwise, it is a server connection (listener).
	audioHandle	Pointer to where the handle for the new audio connection should be stored.

Returns: See Table 8: "Audio Return Codes," on page 35.

RemoveAudioConnection ( )

This method is used to disconnect an audio connection. Either the client or the server may use this method to disconnect the connection. Note that usage of this method is optional. The audio connection will be disconnected automatically after the last data connection between the two devices is closed.

Prototype:

```
AUDIO_RETURN_CODE RemoveAudioConnection (UINT16 audioHandle);
```

Parameters: audioHandle Handle to an open audio connection.

Returns: See Table 8: "Audio Return Codes," on page 35.

virtual OnAudioConnected ( )

This is a virtual function. It is used to indicate that an audio connection has been established. Applications may provide a derived method to process the connection establishment notice.

Prototype:

```
virtual void OnAudioConnected (UINT16 audioHandle);
```

Parameters: audioHandle Handle to the new audio connection.

Returns: void

virtual OnAudioDisconnect ( )

This is a virtual function. It is used to indicate that an audio connection has been disconnected. Applications may provide a derived method to process the disconnection notice.

Prototype:	`virtual void OnAudioDisconnect (UINT16 audioHandle);`	
Parameters:	audioHandle	Handle to the disconnected audio connection.
Returns:	void	

SetEscoMode ( )

This method is used to set up the negotiated parameters for SCO or eSCO, and set the default mode used for *CreateAudioConnection ( ).* It can be called only when there are no active (e)SCO links.

Prototype:

```
static AUDIO_RETURN_CODE SetEScoMode (
                                        tBTM_SCO_TYPE      sco_mode,
                                        tBTM_ESCO_PARAMS   *p_parms);
```

Parameters:	sco_mode	The desired audio connection mode: • BTM_LINK_TYPE_SCO for SCO audio connections. • BTM_LINK_TYPE_ESCO for eSCO audio connections.
	p_parms	A pointer to the (e)SCO link settings to use.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_PARAMS structure definition is:

```
typedef struct
{
   UINT32         tx_bw;
   UINT32         rx_bw;
   UINT16         max_latency;
   UINT16         voice_contfmt;
   UINT16         packet_types;
   UINT8          retrans_effort;
} tBTM_ESCO_PARAMS;
```

Where:

- tx_bw – Transmit bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- rx_bw – Receive bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- max_latency – maximum latency. This is a value in milliseconds, from 0x0004 to 0xfffe. Set to 0xffff for "don't care" setting.
- voice_contfmt – voice content format; CSVD is supported: BTM_ESCO_VOICE_SETTING
- packet_types – the packet type; EV3 is supported: BTM_ESCO_PKT_TYPES_MASK_EV3
- retrans_effort – the retransmit effort; one of:
  - BTM_ESCO_RETRANS_OFF
  - BTM_ESCO_RETRANS_POWER
  - BTM_ESCO_RETRANS_QUALITY
  - BTM_ESCO_RETRANS_DONTCARE

RegForEScoEvts ( )

This function registers an eSCO event callback with the specified object instance. It should be used to receive connection indication events and change of link parameter events.

Prototype:	`static AUDIO_RETURN_CODE RegForEScoEvts ( UINT16 audioHandle, tBTM_ESCO_CBACK *p_esco_cback);`	
Parameters:	audioHandle	The handle to an open audio connection.
	p_esco_cback	A pointer to the callback function.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_CBACK definition is:

```
typedef void (tBTM_ESCO_CBACK) ( tBTM_ESCO_EVT      event,
                                 tBTM_ESCO_EVT_DATA *p_data);
```

The tBTM_ESCO_EVT definition is:

```
typedef UINT8  tBTM_ESCO_EVT;
```

The event could be one of the following values:

- BTM_ESCO_CHG_EVT – change of eSCO link parameter event.
- BTM_ESCO_CONN_REQ_EVT – connection indication event.

The tBTM_ESCO_EVT_DATA structure definition is:

```
typedef union
{
   tBTM_CHG_ESCO_EVT_DATA      chg_evt;
   tBTM_ESCO_CONN_REQ_EVT_DATA conn_evt;
} tBTM_ESCO_EVT_DATA;

typedef struct
{
   UINT16         sco_inx;
   UINT16         rx_pkt_len;
   UINT16         tx_pkt_len;
   BD_ADDR        bd_addr;
   UINT8          hci_status;
   UINT8          tx_interval;
   UINT8          retrans_window;
} tBTM_CHG_ESCO_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- rx_pkt_len – The receive pocket length.
- tx_pkt_len – The transmit pocket length.
- bd_addr – The Bluetooth address of peer device.
- hci_status – HCI status.
- tx_interval – The transmit interval.
- retrans_window – re-transmit window.

```
typedef struct
{
   UINT16          sco_inx;
   BD_ADDR         bd_addr;
   DEV_CLASS       dev_class;
   tBTM_SCO_TYPE   link_type;
} tBTM_ESCO_CONN_REQ_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- bd_addr – The Bluetooth address the device.
- dev_class – The device class.
- link_type – The audio connection type.

ReadEScoLinkData ( )

This function retrieves current eSCO link data for the specified handle. This can be called anytime when a connection is active.

Prototype:	`static AUDIO_RETURN_CODE ReadEScoLinkData ( UINT16 audioHandle, tBTM_ESCO_DATA *p_data);`	
Parameters:	audioHandle	The handle of an open audio connection.
	p_data	A pointer to the buffer where the current eSCO link settings will be stored.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_DATA structure definition is:

```
typedef struct
{
   UINT16      rx_pkt_len;
   UINT16      tx_pkt_len;
   BD_ADDR     bd_addr;
   UINT8       link_type
   UINT8       tx_interval;
   UINT8       retrans_window;
   UINT8       air_mode;
} tBTM_ESCO_DATA;
```

Where:

- rx_pkt_len – The transmit packet length.
- tx_pkt_len – The receive packet length.
- bd_addr – The device Bluetooth device address.
- link_type – the current audio connection type. It could be one of the following values:
  - BTM_LINK_TYPE_SCO
  - BTM_LINK_TYPE_ESCO
- tx_interval – The transmit interval.
- retrans_window – The retransmit window.
- air_mode – The air mode.

ChangeEscoLinkParms ( )

This function requests renegotiation of the parameters on the current eSCO link. If any of the changes are accepted by the controllers, the BTM_ESCO_CHG_EVT event is sent via the callback function registered in *RegForEScoEvts ( ),* with the current settings of the link.

Prototype:

```
static AUDIO_RETURN_CODE ChangeEScoLinkParms (
                             UINT16                  audioHandle,
                             tBTM_CHG_ESCO_PARAMS   *p_parms);
```

Parameters:	audioHandle	The handle to an open audio connection.
	p_parms	A pointer to the settings that need to be changed.
Returns:	See Table 8: “Audio Return Codes,” on page 35.	

The tBTM_CHG_ESCO_PARAMS structure definition is follows:

```
typedef struct
{
   UINT16     max_latency;
   UINT16     packet_types;
   UINT8      retrans_effort;
} tBTM_CHG_ESCO_PARAMS;
```

Where:

- max_latency – maximum latency. This is a value in milliseconds.
- packet_types – packet type.
- retrans_effort – The retransmit effort.

EScoConnRsp ( )

This function should be called upon receipt of an eSCO connection request event (BTM_ESCO_CONN_REQ_EVT) to accept or reject the request.

The hci_status parameter should be [0x00] to accept, [0x0d .. 0x0f] to reject.

The p_params tBTM_ESCO_PARAMS pointer argument is used to negotiate the settings on the eSCO link. If p_parms is NULL, the values set through *SetEScoMode ( )* are used.

Prototype:

```
static void EScoConnRsp (UINT16                 audioHandle,
                         UINT8                  hci_status,
                         tBTM_ESCO_PARAMS       *p_parms = NULL);
```

Parameters:	audioHandle	Handle to an open audio connection.
	hci_status	The HCI status. Zero is HCI success, defined as HCI_SUCCESS.
	p_parms	A pointer to the tBTM_ESCO_PARAMS to be negotiated.
Returns:	void	

CL2CapIf

An object of class CL2CapIf must work with an object of class CL2CapConn to communicate at the L2CAP layer.

This class associates a PSM value with a service GUID and registers the PSM value with the L2CAP protocol layer. The client and the server must perform these functions before an L2CAP connection is attempted.

Multiple simultaneous connections to the different devices can be created in the application. One CL2CapIf object needs to be created and used by a CL2CapConn object. One CL2CapConn object needs to be created and managed for each connection. One CL2CapIf object may be used with multiple CL2CapConn objects.

For more information on the *Logical Link Control and Adaptation Protocol* see the Bluetooth Special Interest Group, L2CAP Specification.

AssignPsmValue ( )

This function is used to assign a PSM value to the interface. It may be called with or without a PSM parameter. If it is called without a PSM value, one is assigned by L2CAP.

A server normally calls this method *without* the PSM parameter, and the L2CAP layer assigns an available PSM value.

A client normally calls this method *with* the PSM found during service discovery.

Prototype:

```
BOOL AssignPsmValue (
                        GUID      *p_guid,
                        UINT16    psm = 0);
```

Parameters:

p_guid	A pointer to the GUID of the service the PSM is for.
psm	A nonzero psm is invalid a) if the value is less than or equal to 0x0003, b) if the low-order bit of the low byte is not 1, or c) if the low-order bit of the high byte is not 0. Check a) protects the reserved PSM values 1 for Service Discovery and 3 for RFCOMM use. Values between 3 and 0x1000 should be used with care to avoid conflicts with other reserved PSM values. Check b) insures that the PSM value is odd. Check c) allows for future use of PSM values longer than 2 bytes.

Returns: TRUE if a PSM was assigned OK; FALSE, otherwise.

Note: Normally this method should not fail. It will only fail in the odd case that all PSM values are in use, or the application is trying to submit a PSM value that someone else is using, or is invalid.

Register ( )

After the PSM is assigned, both the client and server applications should call this function to register the PSM with the L2CAP layer.

A server must call this function before calling *CL2CapConn::Listen ( ).*

A client must call this function before calling *CL2CapConn::Connect ( )*.

Prototype:	`BOOL Register ( );`
Parameters:	None
Returns:	TRUE if the registration was OK; FALSE, otherwise.

Deregister ( )

This function removes the PSM registration from L2CAP. It must be called after *CL2CapConn::Disconnect ( )*.

Prototype:	`void Deregister ( );`
Parameters:	None
Returns:	void

GetPsm ( )

This function returns the PSM value assigned to the interface.

Prototype:	`UINT16 GetPsm ( );`
Parameters:	None
Returns:	The PSM assigned to the interface. Zero (0) if no PSM is assigned.

SetSecurityLevel ( )

This function sets the security level for all connections on this L2CAP interface. Both the client and server applications must call this function after registering with L2CAP.

The client always initiates a call, so the client uses values for the *outgoing* side of the call. The server sees the call as an *incoming* connection.

A connection cannot be established until this function is called.

Prototype:

```
BOOL SetSecurityLevel (
                               BT_CHAR   *p_service_name,
                               UINT8     security_level,
                               BOOL      is_server);
```

Parameters:

p_service_name	The name of the service.
security_level	The desired security level. BTM_SEC_NONE for no security, or one or more of the following: BTM_SEC_IN_AUTHORIZE (used by server side) BTM_SEC_IN_AUTHENTICATE (used by server side) BTM_SEC_IN_ENCRYPT (used by server side; cannot be set alone; has to have BTM_SEC_IN_AUTHENTICATE set) BTM_SEC_OUT_AUTHENTICATE (used by client side) BTM_SEC_OUT_ENCRYPT (used by client side; cannot be set alone; has to have BTM_SEC_OUT_AUTHENTICATE set) See BtIfDefinitions.h. ***Note:*** When multiple values other than BTM_SEC_NONE are used, they must be bit-ORed in the security_level field.
is_server	TRUE if the local device is a server; FALSE if it is a client.

Returns: TRUE if operation was successful; FALSE, otherwise.

CL2CAPCONN

This class controls L2CAP connections. An object of class CL2CapConn must work with an object of class CL2CapIf, which registers a PSM and sets security for L2CAP connections.

Methods are provided for all the commands and responses of the L2CAP protocol, including the writing of data and the processing of connections, data packets received, etc.

CL2CapConn is a base class; it defines a set of virtual methods that serve as event handlers for L2CAP protocol events. The application should provide a derived class that defines real methods for these virtual methods.

Multiple simultaneous connections to the different devices can be created in the application. One CL2CapConn object needs to be created and managed for each connection, and it needs to be associated with a CL2CapIf object. Multiple CL2CapConn objects may be associated with one CL2CapIf object.

In addition to the methods described in this section, there are three public member variables that an application may read, if desired:

- *m_isCongested* is a flag to show if the L2CAP connection is congested.
- *m_RemoteMtu* contains the remote MTU.
- *m_RemoteBdAddr* contains the BD Address of the remote device.

Listen ( )

This method is used by server applications to listen for incoming connections. Clients should not call this method.

The application is notified of incoming connection requests via its *OnIncomingConnection ( )* method. It should then accept or reject the incoming connection.

Prototype:	`BOOL Listen (CL2CapIf *p_if);`	
Parameters:	p_if	A pointer to the L2CAP interface object.
Returns:	TRUE if the connection was put into the *listen* state; FALSE, otherwise.	

Note: This function will fail if the interface is not registered with L2CAP or if the connection object is not in idle state, i.e., it is already listening or is connected.

Accept ( )

Server applications call this method to accept an incoming connection after receiving a connection indication from L2CAP.

When accepting a connection, an application can specify a nondefault MTU (Maximum Transmission Unit). The MTU sets an upper limit for a single packet at the Bluetooth protocol layer being used. For instance, when side A sets an MTU value to side B, side A is saying it has an input buffer MTU bytes long. Side B must react by segmenting its transmissions, if necessary, so that a single packet is no larger than MTU bytes. The receiving side reconstructs the segmented packets into the original message size. The segmenting and reconstruction occur below the application level.

The L2CAP layer accepts the connection and negotiates configuration. The server application is informed when the connection is established or has failed, via its notification functions.

Prototype:	`BOOL Accept (UINT16 desired_mtu = L2CAP_DEFAULT_MTU);`	
Parameter:	desired_mtu	An optional parameter that can be passed if the application wants a nondefault MTU. ***Note:*** L2CAP_DEFAULT_MTU is the maximum value for MTU; anything greater than this will be defaulted back to L2CAP_DEFAULT_MTU.
Returns:	This method will fail (return FALSE) only if the connection object has not received an incoming connection indication.	

Reject ( )

Server applications call this method to reject an incoming connection after receiving a connection indication from L2CAP.

This method will fail only if the connection object has not received an incoming connection indication.

The rejection reason L2CAP_CONN_PENDING is used by the server when extra time is needed. This is typically used when server-side security requires human intervention.

Prototype:	`BOOL Reject (UINT16 reason);`	
Parameter:	reason	The reason for rejecting the connection. The reasons are defined in *BtIfDefinitions.h*. They are: • L2CAP_CONN_OK • L2CAP_CONN_PENDING • L2CAP_CONN_NO_PSM • L2CAP_CONN_SECURITY_BLOCK • L2CAP_CONN_NO_RESOURCES • L2CAP_CONN_TIMEOUT • L2CAP_CONN_NO_LINK
Returns:	This method will fail (return FALSE) only if the connection object has not received an incoming connection indication.	

Connect ( )

Client applications call this method to create a connection to a server. Typically, the server's address is obtained through the CBtIf discovery process.

The client may specify a nondefault MTU (Maximum Transmission Unit). The MTU sets an upper limit for a single packet at the Bluetooth protocol layer being used. For instance, when side A sets an MTU value to side B, side A is saying it has an input buffer MTU bytes long. Side B must react by segmenting its transmissions, if necessary, so that a single packet is no larger than MTU bytes. The receiving side reconstructs the segmented packets into the original message size. The segmenting and reconstruction occur below the application level.

The client is notified of the success or failure of the connection attempt via its *OnConnected ( )* and *OnRemoteDisconnected ( )* methods.

Note: A TRUE return from this method does not indicate a successful connection, it simply indicates a successful channel assignment.

Prototype:

```
BOOL Connect (
                 CL2CapIf      *p_if,
                 BD_ADDR       p_bd_addr,
                 UINT16        desired_mtu = L2CAP_DEFAULT_MTU);
```

Parameters:	p_if	A pointer to the L2CAP interface object.
	p_bd_addr	The Bluetooth device address of the remote device.
	desired_mtu	An optional parameter that can be passed if the application wants a nondefault MTU.

Returns: TRUE if the L2CAP layer was able to assign a channel identifier (CID) for the connection; FALSE, otherwise.

Reconfigure ( )

Client and server applications call this function to change the MTU size. The MTU sets an upper limit for a single packet at the Bluetooth protocol layer being used. For instance, when side A sets an MTU value to side B, side A is saying it has an input buffer MTU bytes long. Side B must react by segmenting its transmissions, if necessary, so that a single packet is no larger than MTU bytes. The receiving side reconstructs the segmented packets into the original message size. The segmenting and reconstruction occur below the application level.

Prototype:

```
BOOL Reconfigure (tL2CAP_CONFIG_INFO *p_cfg);
```

Parameters:	p_cfg	A pointer to a configuration structure.

Returns: TRUE if the new configuration was accepted; FALSE, otherwise.

The tL2CAP_CONFIG_INFO structure definition is as follows:

```
typedef struct
{
    UINT16      result;
    BOOL        mtu_present;
    UINT16      mtu;
    BOOL        qos_present;
    FLOW_SPEC   qos;
    BOOL        flush_to_present;
    UINT16      flush_to;
    UINT16      flags;
} tL2CAP_CONFIG_INFO;
```

where:

- mtu_present – TRUE if the MTU field is present
- mtu – MTU value

The remaining fields in the structure are not supported by the SDK.

Disconnect ( )

Either client or server applications may call this function to disconnect an established connection.

Prototype:	`void Disconnect (void);`
Parameters:	None
Returns:	void

Write ( )

This function is used to send data to an established connection. Usually all the data will be written, but if there is congestion, the write may fail to complete. The application is told how much of its data was actually written to L2CAP.

Prototype:

```
BOOL Write (
             void      *p_data,
             UINT16    length,
             UINT16    *p_len_written);
```

Parameters:	p_data	A pointer to the user data.
	length	The length of the user data.
	p_len_written	A pointer to where the function can store the number of bytes actually written.
Returns:	This function will fail (return FALSE) if the connection is not established.	

GetConnectionStats ( )

This function retrieves current connection statistics, using the common statistics structure tBT_CONN_STATS defined in BtIfDefinitions.h. See "GetConnectionStats ( )" on page 34.

This class uses all the statistic fields in the structure.

Prototype:	`BOOL GetConnectionStats (tBT_CONN_STATS *p_conn_stats);`	
Parameters:	p_conn_stats	A pointer to the user's connection statistics structure; see above.
Returns:	FALSE if a connection attempt has not even been initiated. TRUE otherwise.	

SwitchRole ( )

The application uses this method to request that the device switch role to Master or Slave. If the application desires to accept multiple simultaneous connections, it must execute this command to request that the device switch role to Master.

Prototype:	`BOOL SwitchRole (MASTER_SLAVE_ROLE new_role);`	
Parameters:	new_role	The role to which the local device should switch. Valid values are NEW_MASTER or NEW_SLAVE.
Returns:	TRUE if successful.	

virtual OnIncomingConnection ( )

Server applications should provide this derived function to handle incoming connection attempts. A client application would not provide this function.

Typically, the application function invokes the *Accept ( )* method of the base class. The stack then proceeds with the L2CAP protocol dialog until the *OnConnected ( )* derived function is called.

Prototype:	`virtual void OnIncomingConnection ( );`
Parameters:	None
Returns:	void

virtual OnConnectPendingReceived ( )

Client applications may provide a function to handle connection pending notifications. If the application does not provide this method, the notifications are ignored.

This function is not called under most connection scenarios. When it is called, it is because the server expects an unusual delay setting up the connection. For example, the server may have to wait for resources, or a security dialog is in progress for a PIN code or authorization. The client application can use this function to notify an online user that a delay is expected.

Prototype:	`virtual void OnConnectPendingReceived (void);`
Parameters:	None
Returns:	void

virtual OnConnected ( )

This is a virtual function; all applications should provide a derived method to process the connection. The derived method is called for both client and server applications.

After this function is called, both client and server applications may send data packets.

Prototype:	`virtual void OnConnected ( );`
Parameters:	None
Returns:	void

virtual OnDataReceived ( )

This is a virtual function; all applications should provide a derived method to process the data packet received from the remote device.

Prototype:
```
virtual void OnDataReceived (
                       void      *p_data,
                       UINT16    length);
```

Parameters:	p_data	A pointer to the received data.
	length	The length of the received data.
Returns:	void	

virtual OnCongestionStatus ( )

This is a virtual function; all applications should provide a derived method to process changes in the congestion status. The derived method may be called for both client and server applications.

This function is called when the L2CAP level cannot accept more data for transmission. The application should suspend issuing *Write ( )* calls until this function is called again with parameter *is_congested* = FALSE. If the application fails to suspend *Write ( )* calls during congestion, data could be lost.

Prototype:	`virtual void OnCongestionStatus (BOOL is_congested);`	
Parameter:	is_congested	Set TRUE if congested, else FALSE if not.
Returns:	void	

virtual OnRemoteDisconnected ( )

This is a virtual function; all applications should provide a derived method to process disconnects from the remote device. The derived method may be called for both client and server applications.

Prototype:	`virtual void OnRemoteDisconnected (UINT16 reason);`	
Parameter:	reason	Zero (0) if the cause of disconnect was a *disconnect indicator callback* from the L2CAP stack layer.
		Nonzero (>< = 0) if the cause for disconnect was a *configuration confirm callback* that indicated failure to confirm.
Returns:	void	

SetLinkSupervisionTimeOut ( )

This function sets the link supervision timeout value for the connected device. The timeout value is used by the master or slave Bluetooth device to monitor link loss. The same timeout value is used for both SCO and ACL connections for the device. An ACL connection has to be established before calling this function. The Zero value for timeout will disable the Link_Supervision_Timeout check for the connected device. The default timeout is 20 seconds.

Prototype:	`BOOL SetLinkSupervisionTimeOut (UINT16 timeout)`	
Parameters:	timeout	The timeout duration, in number of slots. Each slot is .625 milliseconds. ***Note:*** Zero will disable Link_Supervision_Timeout check.
Returns:	TRUE, if successful; FALSE, otherwise.	

AUDIO CONNECTIONS

Audio connections can be established between Bluetooth devices and associated with a particular L2CAP connection. The application uses the following methods to manage these connections. The reader is directed to the Audio Connections discussion in the CBtIf class description above for an overview of audio connections.

CreateAudioConnection ( )

This method is used to establish an audio connection with a remote device. The same method is used as both a client initiating a connection and a server listening for an incoming connection.

If used as a client to initiate an audio connection, the client must have already established the L2CAP data connection using the *CL2CapConn::Connect ( )* method prior to initiating the audio connection.

If used as a server to listen for an incoming audio connection, the *CL2CapConn::Listen ( )* method must have already been executed to begin listening for an L2CAP data connection prior to executing the *CL2CapConn::CreateAudioConnection ( )* method to begin listening for an audio connection.

Once the audio connection is established, it will be associated with the L2CAP connection. As such, when the L2CAP connection is closed, the audio connection will automatically be closed. The application must implement *OnAudioConnected ( )* to get the connection establishment notice.

There are multiple signatures for this method. The orignal (Basic) version was supplemented with the Enhanced version, which allows a server application to specify a particular Bluetooth device address. The server will then only accept connections from that specific address. The Enhanced version should not be used for a client connection, as the remote address must be previously known from the prerequisite call to *CL2CapConn::Connect ( )*.

Basic Version

Prototype:

```
AUDIO_RETURN_CODE CreateAudioConnection (
                                        BOOL      bIsClient,
                                        UINT16    *audioHandle);
```

Parameters:	bIsClient	TRUE if this is a client connection (initiator); otherwise, it is a server connection (listener).
	audioHandle	Pointer to where the handle for the new audio connection should be stored.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

Enhanced Version

Prototype:

```
AUDIO_RETURN_CODE CreateAudioConnection (
                                        BOOL      bIsClient,
                                        UINT16    *audioHandle,
                                        BD_ADDR   bda);
```

Parameters:

bIsClient	TRUE if this is a client connection (initiator); otherwise, it is a server connection (listener).
audioHandle	Pointer to where the handle for the new audio connection should be stored.
bda	Specific remote device allowed to make connections (server-only). 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF indicates a connection will be accepted from any device.

Returns: See Table 8: "Audio Return Codes," on page 35.

RemoveAudioConnection ( )

This method is used to disconnect an audio connection. Either the client or the server may use this method to disconnect the connection. Note that usage of this method is optional. The audio connection will be disconnected automatically after the L2CAP connection is closed.

Prototype: `AUDIO_RETURN_CODE RemoveAudioConnection (UINT16 audioHandle);`

Parameters: audioHandle — Handle to an open audio connection.

Returns: See Table 8: "Audio Return Codes," on page 35.

virtual OnAudioConnected ( )

This is a virtual function. It is used to indicate that an audio connection has been established. Applications may provide a derived method to process the connection establishment notice.

Prototype: `virtual void  OnAudioConnected (UINT16 audioHandle);`

Parameters: audioHandle — Handle to an open audio connection.

Returns: void

virtual OnAudioDisconnect ( )

This is a virtual function. It is used to indicate an audio connection has been disconnected. Applications may provide a derived method to process the disconnection notice.

Prototype: `virtual void  OnAudioDisconnect (UINT16 audioHandle);`

Parameters: audioHandle — Handle to the disconnected audio connection.

Returns: void

SetEscoMode ( )

This method is used to set up the negotiated parameters for SCO or eSCO, and set the default mode used for *CreateAudioConnection ( ).* It can be called only when there are no active (e)SCO links.

Prototype:

```
AUDIO_RETURN_CODE SetEScoMode (
                                 tBTM_SCO_TYPE      sco_mode,
                                 tBTM_ESCO_PARAMS   *p_parms);
```

Parameters:

Parameters:	sco_mode	The desired audio connection mode: • BTM_LINK_TYPE_SCO for SCO audio connections. • BTM_LINK_TYPE_ESCO for eSCO audio connections.
	p_parms	A pointer to the (e)SCO link settings to use.

Returns: See Table 8: "Audio Return Codes," on page 35.

The tBTM_ESCO_PARAMS structure definition is:

```
typedef struct
{
   UINT32          tx_bw;
   UINT32          rx_bw;
   UINT16          max_latency;
   UINT16          voice_contfmt;
   UINT16          packet_types;
   UINT8           retrans_effort;
} tBTM_ESCO_PARAMS;
```

Where:

- tx_bw – Transmit bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- rx_bw – Receive bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- max_latency – Maximum latency (in milliseconds) from 0x0004 to 0xfffe. Set to 0xffff for "don't care" setting.
- voice_contfmt – voice content format; CSVD is supported: BTM_ESCO_VOICE_SETTING
- packet_types – the packet type; EV3 is supported: BTM_ESCO_PKT_TYPES_MASK_EV3
- retrans_effort – the retransmit effort; one of:
  - BTM_ESCO_RETRANS_OFF
  - BTM_ESCO_RETRANS_POWER
  - BTM_ESCO_RETRANS_QUALITY
  - BTM_ESCO_RETRANS_DONTCARE

RegForEScoEvts ( )

This function registers an eSCO event callback with the specified object instance. It should be used to receive connection indication events and change of link parameter events.

Prototype:

```
AUDIO_RETURN_CODE RegForEScoEvts ( UINT16            audioHandle,
                                   tBTM_ESCO_CBACK   *p_esco_cback);
```

Parameters:	audioHandle	The handle to an open audio connection.
	p_esco_cback	A pointer to the callback function.

Returns: See Table 8: "Audio Return Codes," on page 35.

The tBTM_ESCO_CBACK definition is:

```
typedef void (tBTM_ESCO_CBACK) ( tBTM_ESCO_EVT      event,
                                 tBTM_ESCO_EVT_DATA *p_data);
```

The tBTM_ESCO_EVT definition is:

```
typedef UINT8  tBTM_ESCO_EVT;
```

The event could be one of the following values:

- BTM_ESCO_CHG_EVT – change of eSCO link parameter event.
- BTM_ESCO_CONN_REQ_EVT – connection indication event.

The tBTM_ESCO_EVT_DATA structure definition is:

```
typedef union
{
   tBTM_CHG_ESCO_EVT_DATA       chg_evt;
   tBTM_ESCO_CONN_REQ_EVT_DATA  conn_evt;
} tBTM_ESCO_EVT_DATA;

typedef struct
{
   UINT16         sco_inx;
   UINT16         rx_pkt_len;
   UINT16         tx_pkt_len;
   BD_ADDR        bd_addr;
   UINT8          hci_status;
   UINT8          tx_interval;
   UINT8          retrans_window;
} tBTM_CHG_ESCO_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- rx_pkt_len – The receive pocket length.
- tx_pkt_len – The transmit pocket length.
- bd_addr – The Bluetooth address of peer device.
- hci_status – HCI status.
- tx_interval – The transmit interval.
- retrans_window – re-transmit window.

```
typedef struct
{
  UINT16          sco_inx;
  BD_ADDR         bd_addr;
  DEV_CLASS       dev_class;
  tBTM_SCO_TYPE   link_type;
} tBTM_ESCO_CONN_REQ_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- bd_addr – The Bluetooth address the device.
- dev_class – The device class.
- link_type – The audio connection type.

ReadEScoLinkData ( )

This function retrieves current eSCO link data for the specified handle. This can be called anytime when a connection is active.

Prototype:	`AUDIO_RETURN_CODE ReadEScoLinkData ( UINT16 audioHandle, tBTM_ESCO_DATA *p_data);`	
Parameters:	audioHandle	The handle of an open audio connection.
	p_data	A pointer to the buffer where the current eSCO link settings will be stored.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_DATA structure definition is:

```
typedef struct
{
   UINT16       rx_pkt_len;
   UINT16       tx_pkt_len;
   BD_ADDR      bd_addr;
   UINT8        link_type
   UINT8        tx_interval;
   UINT8        retrans_window;
   UINT8        air_mode;
} tBTM_ESCO_DATA;
```

Where:

- rx_pkt_len – The transmit packet length.
- tx_pkt_len – The receive packet length.
- bd_addr – The device Bluetooth device address.
- link_type – the current audio connection type. It could be one of the following values:
  - BTM_LINK_TYPE_SCO
  - BTM_LINK_TYPE_ESCO
- tx_interval – The transmit interval.
- retrans_window – The retransmit window.
- air_mode – The air mode.

ChangeEscoLinkParms ( )

This function requests renegotiation of the parameters on the current eSCO link. If any of the changes are accepted by the controllers, the BTM_ESCO_CHG_EVT event is sent via the callback function registered in *RegForEScoEvts ( ),* with the current settings of the link.

Prototype:	`AUDIO_RETURN_CODE ChangeEScoLinkParms ( UINT16 audioHandle, tBTM_CHG_ESCO_PARAMS *p_parms);`	
Parameters:	audioHandle	The handle to an open audio connection.
	p_parms	A pointer to the settings that need to be changed.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_CHG_ESCO_PARAMS structure definition is follows:

```
typedef struct
{
    UINT16     max_latency;
    UINT16     packet_types;
    UINT8      retrans_effort;
} tBTM_CHG_ESCO_PARAMS;
```

Where:

- max_latency – maximum latency. This is a value in milliseconds.
- packet_types – packet type.
- retrans_effort – The retransmit effort.

EScoConnRsp ( )

This function should be called upon receipt of an eSCO connection request event (BTM_ESCO_CONN_REQ_EVT) to accept or reject the request.

The hci_status parameter should be [0x00] to accept, [0x0d .. 0x0f] to reject.

The p_params tBTM_ESCO_PARAMS pointer argument is used to negotiate the settings on the eSCO link. If p_parms is NULL, the values set through *SetEScoMode ( )* are used.

Prototype:

```
void EScoConnRsp (UINT16               audioHandle,
                  UINT8                hci_status,
                  tBTM_ESCO_PARAMS     *p_parms = NULL);
```

Parameters:	audioHandle	Handle to an open audio connection.
	hci_status	The HCI status. Zero is HCI success, defined as HCI_SUCCESS.
	p_parms	A pointer to the tBTM_ESCO_PARAMS to be negotiated.
Returns:	void	

CSDPSERVICE

This class is used to create and manage SDP service records; it should only be used by applications that offer services.

For more information on the *Service Discovery Protocol,* see the Bluetooth Special Interest Group, SDP Specification.

One instance of this class should be instantiated for each service record that needs to be added (typically one application will only add one service record).

All methods return an enumerated type SDP_RETURN_CODE.

Table 9: SDP_RETURN_CODE

SDP_RETURN_CODE Value	*Meaning*
SDP_OK	Operation initiated without error.
SDP_COULD_NOT_ADD_RECORD	The SDP record could not be created (database full).
SDP_INVALID_RECORD	The function was called before the function AddServiceClassIdList was successfully called.
SDP_INVALID_PARAMETERS	One or more of the parameters passed to the function were invalid.

AddServiceClassIdList ( )

This function must be the first method of the class that the application calls. It adds a service class ID list attribute to the service record. Typically, applications will only have one GUID in the service class ID list, but the Bluetooth specification allows for a list. If a list is used, it should be sorted from most specific to least specific.

Prototype:

```
SDP_RETURN_CODE AddServiceClassIdList (
                                int    num_guids,
                                GUID   *p_service_guids);
```

Parameters:

num_guids	The number of GUIDs in the list. Maximum value allowed is MAX_UUIDS_PER_SEQUENCE (BtIfDefinitions.h), and cannot be less than 1.
p_service_guids	A pointer to a list of *num_guids* GUIDs. If this parameter is NULL, error is SDP_INVALID_PARAMETERS returned.

Note: No duplicate service GUID is checked in the SDK.

Returns: See Table 9: "SDP_RETURN_CODE," on page 56.

AddServiceName ( )

This function adds a service name attribute to the service record.

Prototype:

```
SDP_RETURN_CODE AddServiceName (BT_CHAR *p_service_name);
```

Parameters:

p_service_name	The name of the service, a null terminated ASCII string. If the string length is greater than BT_MAX_SERVICE_NAME_LEN (see BtIfDefinitions.h), error SDP_INVALID_PARAMETERS is returned. If this parameter is NULL, error SDP_INVALID_PARAMETERS is returned.

Returns: See Table 9: "SDP_RETURN_CODE," on page 56.

AddProfileDescriptorList ( )

This function adds a profile descriptor list attribute to the service record.

Prototype:

```
SDP_RETURN_CODE AddProfileDescriptorList (
                                  GUID      *p_profile_guid,
                                  UINT16    version);
```

Parameters:	p_profile_guid	A pointer to the GUID of the profile. If this parameter is NULL, error SDP_INVALID_PARAMETERS is returned.
	version	The profile version.
Returns:	See Table 9: "SDP_RETURN_CODE," on page 56.	

AddL2CapProtocolDescriptor ( )

This function adds a protocol descriptor list attribute to the service record for an L2CAP-based service.

Prototype:	`SDP_RETURN_CODE AddL2CapProtocolDescriptor (UINT16 psm);`	
Parameters:	psm	The PSM the service is using.
Returns:	See Table 9: "SDP_RETURN_CODE," on page 56.	

AddRFCommProtocolDescriptor ( )

This function adds a protocol descriptor list attribute to the service record for an RFCOMM-based service.

Prototype:	`SDP_RETURN_CODE AddRFCommProtocolDescriptor (UINT8 scn);`	
Parameters:	scn	The RFCOMM SCN for the service.
Returns:	See Table 9: "SDP_RETURN_CODE," on page 56.	

Note: An L2Cap UUID (100) with a value of 3 is added because RFCOMM protocol is over the L2CAP protocol.

AddProtocolList ( )

This function adds a protocol descriptor list attribute to the service record. It should only be needed if the specific L2CAP or RFCOMM functions do not suffice.

It gives the application full control over what goes in the protocol descriptor list attribute.

Each element is defined in the structure tSDP_PROTOCOL_ELEM, see BtIfDefinitions.h

```
#define SDP_MAX_PROTOCOL_PARAMS     1
typedef struct
{
    UINT16      protocol_uuid;
    UINT16      num_params;
    UINT16      params[SDP_MAX_PROTOCOL_PARAMS];
} tSDP_PROTOCOL_ELEM;
```

Where:

- *protocol_uuid* – UUID of the protocol;
- *num_params* – number of parameters; must be less than or equal to SDP_MAX_PROTOCOL_PARAMS;
- *params* – array of parameters;

Prototype:
```
SDP_RETURN_CODE AddProtocolList (
                                int                   num_elem,
                                tSDP_PROTOCOL_ELEM    *p_elem_list);
```

Parameters:

num_elem	The number of elements in the list, maximum value is MAX_PROTOCOL_LIST_ELEM = 3, see BtIfDefinitions.h; otherwise, error return is SDP_INVALID_PARAMETERS. Each element in the list is validated: if the uuid = 0, or the num_params values > SDP_MAX_PROTOCOL_PARAMS causes an error return SDP_INVALID_PARAMETERS. If this parameter is NULL, error SDP_INVALID_PARAMETERS is returned. ***Note:*** **Note:** No duplicate protocol_uuid is checked in the SDK.
p_elem_list	A pointer to the protocol list to add.

Returns: See Table 9: "SDP_RETURN_CODE," on page 56.

AddAdditionProtoLists ( )

This method adds the additional sequence of generic protocol descriptor lists to a service record. It should only be needed if the specific RFCOMM and L2CAP functions above do not suffice.

The definition of the tSDP_PROTO_LIST_ELEM is as follows:

```
#define SDP_MAX_LIST_ELEMS      3
typedef struct
{
   UINT16              num_elems;
   tSDP_PROTOCOL_ELEM  list_elem[SDP_MAX_LIST_ELEMS];
} tSDP_PROTO_LIST_ELEM;
```

Where:

- *num_elems* – number of elements;
- *list_elem* – array of elements.

Prototype:
```
SDP_RETURN_CODE AddAdditionProtoLists (
                                int                     num_list_elem,
                                tSDP_PROTO_LIST_ELEM    *p_proto_list);
```

Parameters:

num_list_elem	The number of elements in the list, maximum value is MAX_ELEM_IN_SEQ = 10, see BtIfDefinitions.h; otherwise, error return is SDP_INVALID_PARAMETERS. Each element in the list is validated: if the num_list_elem > SDP_MAX_LIST_ELEMS an SDP_INVALID_PARAMETERS error will be returned.
p_proto_list	A pointer to the protocol list to add. If this parameter is NULL, error SDP_INVALID_PARAMETERS is returned.

Returns: See Table 9: "SDP_RETURN_CODE," on page 56

AddLanguageBaseAttrIDList ( )

This function adds a language base attribute to the service record.

Prototype:	`SDP_RETURN_CODE AddLanguageBaseAttrIDList ( UINT16 lang, UINT16 char_enc, UINT16 base_id);`	
Parameters:	lang	The natural language code. The language is encoded according to ISO 639:1988 (E/F): *Code for the representation of names of languages.*
	char_enc	The character encoding scheme.
	base_id	The language base.
Returns:	See Table 9: "SDP_RETURN_CODE," on page 56.	

MakePublicBrowseable ( )

This function adds the browse group list attribute to the record, with the public browse root UUID (0x1002).

Prototype:	`SDP_RETURN_CODE MakePublicBrowseable (void);`
Parameter:	None
Returns:	See Table 9: "SDP_RETURN_CODE," on page 56.

SetAvailability ( )

This function sets the availability attribute in the service record.

Prototype:	`SDP_RETURN_CODE SetAvailability (UINT8 availability);`	
Parameters:	availability	The service availability: 0 = unavailable. 255 = fully available.
Returns:	See Table 9: "SDP_RETURN_CODE," on page 56.	

AddAttribute ( )

This function adds an attribute to a service record. It is used to add attributes not covered by the existing functions, for example *AddProtocolList ( ).*

All attribute values for the specified attribute ID must be included in a single call. A subsequent call to this function using the same attribute ID will replace the previous values with those in the subsequent call.

All attribute values must be provided in the form of a string of 'unsigned char'. This string may be a simple data type such as UINT_DESC_TYPE or TEXT_STR_DESC_TYPE — see Table 11: "Service Record Attribute Types," on page 62.

Note that numeric data types, UINT_DESC_TYPE and TWO_COMP_INT_DESC_TYPE, should be in big-endian order. For example, a value of type UINT_DESC_TYPE, length 2, and value 0x102 (= decimal 258), would be represented in the value string as '0x01, 0x02'.

More complex value strings are possible with the use of types DATA_ELE_SEQ_DESC_TYPE and DATA_ELE_ALT_DESC_TYPE, which define a sequence of values, each of which may be a simple data type or another level of sequence values.

The Bluetooth SIG Technical Standard Version 1.1, Part E, SERVICE DISCOVERY PROTOCOL provides a detailed description of the service attributes.

Prototype:

```
SDP_RETURN_CODE AddAttribute (
                              UINT16    attr_id,
                              UINT8     attr_type,
                              UINT8     attr_len,
                              UINT8     *p_val);
```

Parameters:

attr_id — The ID of the attribute, see Table 10: "Service Record Attribute IDs," on page 61 and BtIfDefinitions.h.

Note: The value ATTR_ID_SERVICE_CLASS_ID_LIST may not be used with this function. Otherwise, error SDP_INVALID_PARAMETERS is returned. There is a special SDK function, *AddServiceClassIdList ( )* that must be used for that attribute. New applications may define their own attribute identifiers as long as the value is greater than 0x300 and does not conflict with the existing values in the Table 10: "Service Record Attribute IDs," on page 61.

attr_type — The data type of the attribute.

The basic types supported are:

Unsigned integer

Signed integer

UUID

URL

Text string

Boolean

Sequences of the above types are also supported.

See Table 11: "Service Record Attribute Types," on page 62 and BtIfDefinitions.h.

attr_len — The length of the attribute.

Note: If the length is larger than maximum length supported by the BTW version (256 bytes on BTW versions prior to 5.0.1.400, or 496 bytes on BTW versions 5.0.1.400 or greater), the value will get truncated, but the call will still return SDP_OK.

p_val — A pointer to the attribute value. If this parameter is NULL, error SDP_INVALID_PARAMETERS is returned.

Returns: See Table 9: "SDP_RETURN_CODE," on page 56.

Table 10: Service Record Attribute IDs

Attribute Code[a]
ATTR_ID_SERVICE_RECORD_HDL
ATTR_ID_SERVICE_RECORD_STATE
ATTR_ID_SERVICE_ID
ATTR_ID_PROTOCOL_DESC_LIST
ATTR_ID_BROWSE_GROUP_LIST
ATTR_ID_LANGUAGE_BASE_ATTR_ID_LIST
ATTR_ID_SERVICE_INFO_TIME_TO_LIVE
ATTR_ID_SERVICE_AVAILABILITY
ATTR_ID_BT_PROFILE_DESC_LIST
ATTR_ID_DOCUMENTATION_URL
ATTR_ID_CLIENT_EXE_URL
ATTR_ID_ICON_URL
ATTR_ID_ADDITION_PROTO_DESC_LISTS
ATTR_ID_SERVICE_NAME
ATTR_ID_SERVICE_DESCRIPTION
ATTR_ID_PROVIDER_NAME
ATTR_ID_VERSION_NUMBER_LIST
ATTR_ID_GROUP_ID
ATTR_ID_SERVICE_DATABASE_STATE
ATTR_ID_SUPPORTED_DATA_STORES
ATTR_ID_EXTERNAL_NETWORK
ATTR_ID_FAX_CLASS_1_SUPPORT
ATTR_ID_REMOTE_AUDIO_VOLUME_CONTROL
ATTR_ID_DEVICE_NAME
ATTR_ID_SUPPORTED_FORMATS_LIST
ATTR_ID_FAX_CLASS_2_0_SUPPORT
ATTR_ID_SUPPORTED_FORMATS_LIST
ATTR_ID_FAX_CLASS_2_SUPPORT
ATTR_ID_FRIENDLY_NAME
ATTR_ID_AUDIO_FEEDBACK_SUPPORT

a. See the Bluetooth Core Spec 1.1, Part E Section 5.1 for precise definitions of the service attribute codes.

Table 11: Service Record Attribute Types

Attribute Type Code	*Meaning*
NULL_DESC_TYPE	Null
UINT_DESC_TYPE	Unsigned Integer
TWO_COMP_INT_DESC_TYPE	Signed 2s Complement Integer
UUID_DESC_TYPE	UUID – Universally Unique Identifier
TEXT_STR_DESC_TYPE	String of text
BOOLEAN_DESC_TYPE	Boolean, TRUE or FALSE
DATA_ELE_SEQ_DESC_TYPE	A sequence of data elements, all of which make up the data
DATA_ELE_ALT_DESC_TYPE	A sequence of data elements, one of which must be chosen, called a data sequence alternative
URL_DESC_TYPE	URL, Uniform Resource Locator

DeleteAttribute ( )

This function deletes an attribute from the service record.

Prototype:

```
SDP_RETURN_CODE DeleteAttribute (UINT16 attr_id);
```

Parameters: attr_id — The ID of the attribute.

Returns: See Table 9: "SDP_RETURN_CODE," on page 56.

AddSupportedFormatsList ( )

This method adds a list (sequence) for the 'supported formats' attribute.

Prototype:

```
SDP_RETURN_CODE AddSupportedFormatsList(
                                        UINT8     num_formats,
                                        UINT8     pDataType[],
                                        UINT8     pDataTypeLength[],
                                        UINT8     *pDataTypeValue[]);
```

Parameters:

num_formats	Number of elements in the pDataType and pDataTypeLength arrays. This parameter is validated so that if it is greater than MAX_ELEM_IN_SEQ = 10, see BtIfDefinitions.h, the function reads MAX_ELEM_IN_SEQ elements from the arrays.
pDataType	Array of data types.
pDataTypeLength	Array of lengths of elements in the pDataTypeValue array.
pDataTypeValue	Array of pointers to values. Each array element is an array of UINT8, sized by the corresponding element in the pDataTypeLength array parameter. As the data type for the pDataTypeLength array elements is also UINT8, this means the maximun length that can be set is 255 bytes. ***Note:*** If the length of any element of the pDataTypeValue array is larger than maximum length, the value will get truncated, but the call will still return SDP_OK.

Returns: See Table 9: "SDP_RETURN_CODE," on page 56.

CommitRecord()

This method publishes or republishes a record in the local SDP database. An application must call this method after creating a record to activate the record on Vista Profile Pack deployments. Similarly, the application must call this method to activate any changes made to the record after a previous call to CommitRecord(). On BTW 5 and previous deployments, the method is a no-op, but is recommended for application portability.

Prototype:	`SDP_RETURN_CODE CommitRecord ();`
Parameters:	None
Returns:	Table 9: "SDP_RETURN_CODE," on page 56.

CSdpDiscoveryRec

When an application reads the services from a remote device, via *CBtIf::ReadDiscoveryRecords ( )*, the service records are returned in objects of this class. Methods are provided to interrogate the discovery record.

In addition to the methods described below, there are two public member variables that can be read by an application:

- *m_service_name* is the service name, a null-terminated string of type 'BT_CHAR'
- *m_service_guid* is the service GUID, of type 'GUID'

FindRFCommScn ( )

This function looks through the discovery record for the protocol descriptor list and, if found, tries to extract the RFCOMM SCN parameter from it.

Prototype:	`BOOL FindRFCommScn (UINT8 *pScn);`	
Parameters:	pScn	The location where the function will store the SCN, if found.
Returns:	TRUE if an SCN was found; FALSE, otherwise.	

FindL2CapPsm ( )

This function looks through the discovery record for the protocol descriptor list and, if found, tries to extract the L2CAP PSM parameter from it.

Prototype:	`BOOL FindL2CapPsm (UINT16 *pPsm);`	
Parameters:	pPsm	The location where the function will store the PSM, if found.
Returns:	TRUE if a PSM was found; FALSE, otherwise.	

FindProtocolListElem ( )

This function looks through the discovery record for the protocol descriptor list and, if found, returns the list element for the specified protocol UUID to the caller.

Specialized versions of this function are defined above (*FindRFCommScn ( )* for the SCN parameter, and *FindL2CapPsm ( )* for the PSM parameter). This function would be used for other predefined protocols or new protocols.

Prototype:

```
BOOL FindProtocolListElem (
                           UINT16              layer_uuid,
                           tSDP_PROTOCOL_ELEM  *p_elem);
```

Parameters:	layer_uuid.	The 16-bit UUID of the protocol layer being looked for. This is a protocol UUID (such as UUID_PROTOCOL_OBEX, or UUID_PROTOCOL_L2CAP in BtIfDefinitions.h). This is not a service class UUID (such as UUID_SERVCLASS_SERIAL_PORT).
	p_elem	The location where the function will store the protocol list element. The object of interest here is the parameter associated with the protocol being used in the service record. See tSDP_PROTOCOL_ELEM definition in "AddProtocolList ( )" on page 57 or BtIfDefinitions.h.
Returns:	TRUE if the list element was found; FALSE, otherwise.	

FindAdditionalProtocolListElem ( )

This function looks through the discovery record for the additional protocol descriptor list and, if found, returns the list element for the specified protocol UUID to the caller.

Prototype:

```
BOOL FindAdditionalProtocolListElem (
                                     UINT16              layer_uuid,
                                     tSDP_PROTOCOL_ELEM  *p_elem);
```

Parameters:	layer_uuid.	The 16-bit UUID of the protocol layer being looked for. This is a protocol UUID (such as UUID_PROTOCOL_OBEX, or UUID_PROTOCOL_L2CAP in BtIfDefinitions.h). This is not a service class UUID (such as UUID_SERVCLASS_SERIAL_PORT).
	p_elem	The location where the function will store the protocol list element. The object of interest here is the parameter associated with the protocol being used in the service record. See tSDP_PROTOCOL_ELEM definition in "AddProtocolList ( )" on page 57 or BtIfDefinitions.h.
Returns:	TRUE if the list element was found; FALSE, otherwise.	

FindProfileVersion ( )

This function looks through the discovery record for the profile descriptor list and, if found, returns the profile version to the caller.

Prototype:

```
BOOL FindProfileVersion (
                                GUID      *p_profile_guid,
                                UINT16    *p_version);
```

Parameters:	p_profile_guid	The GUID of the profile being searched for.
	p_version	The location where the function should store the version.
Returns:	TRUE if the profile descriptor list was found; FALSE, otherwise.	

FindAttribute ( )

This function searches the discovery record for the specified attribute ID and, if found, returns the attribute values to the caller.

This is a generic function to be used when the specific functions above do not suffice.

This function returns an array of only the simple data types found (UINT_DESC_TYPE, TWO_COMP_INT_DESC_TYPE, UUID_DESC_TYPE, TEXT_STR_DESC_TYPE, URL_DESC_TYPE, and BOOLEAN_DESC_TYPE). The sequence data types (DATA_ELE_SEQ_DESC_TYPE and DATA_ELE_ALT_DESC_TYPE) are not returned.

Prototype:

```
BOOL FindAttribute (
                        UINT16              attr_id,
                        SDP_DISC_ATTTR_VAL  *p_val);
```

Parameters:	attr_id	The attribute ID being searched for.
	p_val	The location where the function should store the attribute. The structure SDP_DISC_ATTTR_VAL is defined in BtIfDefinitions.h. Note: the structure allows for an attribute that is a sequence of up to MAX_SEQ_ENTRIES (= 20), each with a value array up to MAX_ATTR_LEN (= 496) bytes long. Nested sequences up to 5 levels deep are supported.
Returns:	TRUE if the attribute was found; FALSE, otherwise.	

The SDP_DISC_ATTTR_VAL structure is used to hold attribute values when read from the discovery database. The attribute may be a sequence, in which case the number of elements will be greater than 1.

Note: On BTW versions prior to 5.0.1.400, the SDP MAX_ATTR_LEN was 256. If running SDK applications on those older BTW versions, attribute values longer than 256 bytes will be truncated to 256 bytes.

The definition of the SDP_DISC_ATTTR_VAL structure is as follows:

```
#define MAX_SEQ_ENTRIES      20
#define MAX_ATTR_LEN         496

typedef struct
{
    int num_elem;

    struct
    {
        #define ATTR_TYPE_INT      0    // Attribute value is an integer
        #define ATTR_TYPE_TWO_COMP 1    // Attribute value is an 2's complement integer
        #define ATTR_TYPE_UUID     2    // Attribute value is a UUID
        #define ATTR_TYPE_BOOL     3    // Attribute value is a boolean
        #define ATTR_TYPE_ARRAY    4    // Attribute value is an array of bytes

        int  type;
        int  len;          // Length of the attribute
        BOOL start_of_seq; // TRUE for each start of sequence

        union
        {
            unsigned char    u8;               // 8-bit integer
            unsigned short   u16;              // 16-bit integer
            unsigned long    u32;              // 32-bit integer
            unsigned __int64 u64;              // 64-bit integer
            struct
            {
                unsigned __int64 uLow64;       // low 64 bit part and
                unsigned __int64 uHigh64;      // high 64 bit part of a
            } u128;                            // 128-bit integer

            BOOL b;                            // Boolean
            unsigned char array[MAX_ATTR_LEN]; // Variable length array
        } val;

    } elem [MAX_SEQ_ENTRIES];

} SDP_DISC_ATTTR_VAL;
```

Where:

- num_elem – Number of the elements
- elem – Array of the elements

CRFCOMMIF

This class controls Service Channel Number (SCN) allocation for the SDK RFCOMM applications.

RFCOMM is the serial cable emulation protocol based on ETSI TS 07.10. For more information on the *RFCOMM Protocol,* see the Bluetooth Special Interest Group, RFCOMM with TS 07.10", Specification of the Bluetooth System.

AssignScnValue ( )

The server calls this method with no parameters to assign a new SCN value, or with an SCN value if it is using a fixed SCN.

An SCN value assigned for a server is automatically freed when either another call to *AssignScnValue ( )* is made, or the CRfCommIf object destructor is executed.

The client should call this method with the SCN found from service discovery.

Prototype:

```
BOOL AssignScnValue (
                        GUID      *p_service_guid,
                        UINT8     scn = 0,
                        LPCSTR    szServiceName = "");
```

Parameters:

p_service_guid	A pointer to the GUID of the service the SCN is for.
scn	Optional: may be passed if the application already knows the SCN value to use.
szServiceName	Optional: may be passed to differentiate an SCN for an SDK RFCOMM service using the same GUID as a native stack RFCOMM service. However, this cannot be used to differentiate SCNs between multiple SDK application RFCOMM services using the same GUID.

Returns: TRUE if a SCN was assigned; FALSE, otherwise.

Note: Normally this method should not fail, it will only fail in the odd case that all SCN values are in use, or the application is trying to use a SCN value that someone else is using.

GetScn ( )

This function returns the SCN value in use.

Prototype: `UINT8 GetScn ( );`

Parameters: None

Returns: The SCN assigned to the interface.
Zero (0) if an SCN is not assigned.

SetSecurityLevel ( )

Both client and server applications must call this function to set the desired security level for all connections. This function must be called before attempting to open an RFCOMM connection.

The client always initiates a call, so the client uses values for the *outgoing* side of the call. The server sees the call as an *incoming* connection.

Prototype:

```
BOOL SetSecurityLevel (
                        BT_CHAR   *p_service_name,
                        UINT8     security_level,
                        BOOL      is_server);
```

Parameters:

p_service_name	The name of the service.
security_level	The desired security level. BTM_SEC_NONE for no security or one or more of the following: BTM_SEC_IN_AUTHORIZE (used by server side) BTM_SEC_IN_AUTHENTICATE (used by server side) BTM_SEC_IN_ENCRYPT (used by server side, cannot be set alone, have to have BTM_SEC_IN_AUTHENTICATE set) BTM_SEC_OUT_AUTHENTICATE (used by client side) BTM_SEC_OUT_ENCRYPT (used by client side, cannot be set alone, have to have BTM_SEC_OUT_AUTHENTICATE set) See BtIfDefinitions.h. ***Note:*** When multiple values other than BTM_SEC_NONE are used, they must be bit-ORed in the security_level field.
is_server	TRUE if the local device is a server. FALSE if the local device is a client.

Returns: TRUE if operation was successful; FALSE, otherwise.

CRFCOMMPORT

This class controls RFCOMM connections. Methods are provided for the commands and responses of the RFCOMM protocol, including the writing of data and reacting to connections, data received, etc.

This is a base class that defines a set of virtual methods that serve as event handlers for RFCOMM protocol events. The application should provide a derived class that defines real methods for these virtual methods.

Most methods return an enumerated type PORT_RETURN_CODE.

PORT_RETURN_CODE

A common set of return codes is provided by the direct RFCommPort function calls – *OpenServer ( )*, *OpenClient ( )*, etc. See enumerated type PORT_RETURN_CODE in BtIfClasses.h.

Table 12: PORT_RETURN_CODE

PORT_RETURN_CODE Value	***Meaning***
SUCCESS	Operation initiated without error.
ALREADY_OPENED	Client tried to open a port to an existing DLCI/BD_ADDR. The port has been already open.
NOT_OPENED	The function was called before the connection opened or after it closed.
HANDLE_ERROR	Not used currently – handle errors should result in NOT_OPENED.
LINE_ERR	Line error.
START_FAILED	Connection attempt failed.
PAR_NEG_FAILED	Parameter negotiation failed, currently only MTU.
PORT_NEG_FAILED	Port negotiation failed.
PEER_CONNECTION_FAILED	Connection ended by remote side.
PEER_TIMEOUT	Timeout by remote side.
INVALID_PARAMETER	One or more of function parameters are invalid.
UNKNOWN_ERROR	Any condition other than the above.

OpenServer ( )

This function opens a server connection for an RFCOMM serial port and listens for a connection attempt.

Prototype:

```
PORT_RETURN_CODE OpenServer (
                           UINT8      scn,
                           UINT16     desired_mtu = RFCOMM_DEFAULT_MTU);
```

Parameters:	scn	Must have already been assigned by *CRfCommScn::AssignScnValue ( )*.
	desired_mtu	Optional: can be passed if the application wants a nondefault MTU. If this parameter is less then L2CAP_MIN_MTU (see BtIfDefintions.h), error INVALID_PARAMETER is returned.

Returns: SUCCESS – Everything is OK. Otherwise, see Table 12: "PORT_RETURN_CODE," on page 69.

OpenClient ( )

This function opens a client connection for an RFCOMM serial port and initiates the connection.

The SCN for the service must have already been obtained from the service discovery procedure.

Prototype:

```
PORT_RETURN_CODE OpenClient (
             UINT8      scn,
             BD_ADDR    RemoteBdAddr,
             UINT16     desired_mtu = RFCOMM_DEFAULT_MTU);
```

Parameters:	scn	The Service Channel Number obtained from the service discovery procedure.
	RemoteBdAddr	The BD address of the server, obtained from the service discovery procedure. If this parameter is NULL, error INVALID_PARAMETER is returned.
	desired_mtu	Optional: can be passed if the application wants a nondefault MTU. If this parameter is less then L2CAP_MIN_MTU (see BtIfDefintions.h), error INVALID_PARAMETER is returned.

Returns: SUCCESS – Everything is OK. Otherwise, see Table 12: “PORT_RETURN_CODE,” on page 69.

Close ( )

This function closes the connection.

Prototype: `PORT_RETURN_CODE Close (void);`

Parameters: None

Returns: SUCCESS – Everything is OK. Otherwise, see Table 12: “PORT_RETURN_CODE,” on page 69.

IsConnected ( )

This function tests to see if the connection has been established.

Prototype: `BOOL IsConnected (BD_ADDR *p_remote_bdaddr);`

Parameters: p_remote_bdaddr A pointer to the caller's buffer for storing the remote BD address.

Returns: TRUE if a connection has been made; FALSE, otherwise.

SetFlowEnabled ( )

This function enables/disables flow control. Signals the remote device to stop/resume sending data packets.

Prototype: `PORT_RETURN_CODE SetFlowEnabled (BOOL enabled);`

Parameters: enabled Set TRUE/FALSE to enable/disable incoming data packets.

Returns: SUCCESS – Everything is OK. Otherwise, see Table 12: “PORT_RETURN_CODE,” on page 69.

SetModemSignal ( )

This function sets the modem control signals.

Prototype:	`PORT_RETURN_CODE SetModemSignal (UINT8 signal);`	
Parameters:	signal	One of the modem signal values defined in BtIfDefinitions.h (see Table 13: "Modem Signals," on page 71). Otherwise, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 12: "PORT_RETURN_CODE," on page 69.	

Table 13: Modem Signals

Modem Signal Value	*Meaning*
PORT_SET_DTRDSR	Sets data-terminal-ready/data-set-ready signal.
PORT_CLR_DTRDSR	Clears data-terminal-ready/data-set-ready signal.
PORT_SET_CTSRTS	Sets clear-to-send/request-to-send signal.
PORT_CLR_CTSRTS	Clears clear-to-send/request-to-send signal.
PORT_SET_RI	Sets ring indicator signal.
PORT_CLR_RI	Clears ring indicator signal.
PORT_SET_DCD	Sets data-carrier-detected signal.
PORT_CLR_DCD	Clears data-carrier-detected signal.
PORT_SET_BREAK	Suspends character transmission and places the transmission line in a break state.
PORT_CLR_BREAK	Restores character transmission and places the transmission line in a nonbreak state.

GetModemStatus ( )

This function reads the modem status.

Prototype:	`PORT_RETURN_CODE GetModemStatus (UINT8 *p_signal);`	
Parameters:	p_signal	One or more of the modem status flags from BtIfDefinitions.h: (see Table 14: "Modem Status Codes," on page 71). If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 12: "PORT_RETURN_CODE," on page 69.	

Table 14: Modem Status Codes

Modem Status Code	*Meaning*
PORT_DTRDSR_ON	Data-terminal-ready/data-set-ready signal is on.
PORT_CTSRTS_ON	Clear-to-send/request-to-send signal is on.
PORT_RING_ON	Ring indicator signal is on.
PORT_DCD_ON	Data-carrier-detected signal is on.

SendError ( )

This function sends a specific error code to the remote device.

Prototype:	`PORT_RETURN_CODE SendError (UINT8 errors);`	
Parameters:	errors	One or more of the Error Code flags from BtIfDefinitions.h: (see Table 15: "Error Code Flags," on page 72).
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 12: "PORT_RETURN_CODE," on page 69.	

Table 15: Error Code Flags

Error Code Flag	*Meaning*
PORT_ERR_BREAK	Break condition occurred on the peer device.
PORT_ERR_OVERRUN	Overrun is reported by peer device.
PORT_ERR_FRAME	Framing error reported by peer device.
PORT_ERR_RXOVER	Input queue overflow occurred.
PORT_ERR_TXFULL	Output queue overflow occurred.

Purge ( )

This function discards all the data from the output or input queues of the specified connection.

Prototype:	`PORT_RETURN_CODE Purge (UINT8 purge_flags);`	
Parameters:	purge_flags	One or both of the following flags from BtIfDefinitions.h: PORT_PURGE_TXCLEAR — clears the output buffer (if the device driver has one). PORT_PURGE_RXCLEAR — clears the input buffer (if the device driver has one). Otherwise, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 12: "PORT_RETURN_CODE," on page 69.	

Write ( )

The client or server calls this function to send data to an existing connection. On return, the length actually written is set.

Prototype:

```
PORT_RETURN_CODE Write (
                        void      *p_data,
                        UINT16    len_to_write,
                        UINT16    *p_len_written);
```

Parameters:	p_data	A pointer to an array of characters. If this parameter is NULL, error INVALID_PARAMETER is returned.
	len_to_write	The number of characters to write.
	p_len_written	The number of characters actually written, returned to the caller. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 12: "PORT_RETURN_CODE," on page 69.	

GetConnectionStats ( )

This function retrieves current connection statistics, using the common statistics structure tBT_CONN_STATS defined in BtIfDefinitions.h. See "GetConnectionStats ( )" on page 34.

This class uses all the statistic fields in the structure.

Prototype:	`PORT_RETURN_CODE GetConnectionStats (tBT_CONN_STATS *p_conn_stats);`	
Parameters:	p_conn_stats	A pointer to the user's connection statistics structure; see above. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS if statistics are returned. Otherwise, see Table 12: "PORT_RETURN_CODE," on page 69.	

virtual OnDataReceived ( )

The client and server should provide this method to be notified when data is received from the remote side.

Prototype:

```
virtual void OnDataReceived (
                               void      *p_data,
                               UINT16    len);
```

Parameters:	p_data	A pointer to the data. The application must move this data to an application level buffer.
	len	The number of characters received.
Returns:	void	

virtual OnEventReceived ( )

The client and server should provide this method to be notified when an event is received from the remote side.

One event, PORT_EV_RXCHAR (data character received) is intercepted and reported via the *OnDataReceived ( )* method, rather than this method.

This method is used to detect a disconnect (PORT_EV_CONNECT_ERR) from the remote device.

Prototype:	`virtual void OnEventReceived (UINT32 event_code);`	
Parameters:	event_code	One or more of the port events from BtIfDefinitions.h (see Table 16: "Port Events," on page 74).
Returns:	void	

Table 16: Port Events

Event	Meaning
PORT_EV_RXFLAG	Received certain character
PORT_EV_TXEMPTY	Transmit Queue Empty
PORT_EV_CTS	CTS changed state
PORT_EV_DSR	DSR changed state
PORT_EV_RLSD	RLSD changed state
PORT_EV_BREAK	BREAK received
PORT_EV_ERR	Line status error occurred
PORT_EV_RING	Ring signal detected
PORT_EV_CTSS	CTS state
PORT_EV_DSRS	DSR state
PORT_EV_RLSDS	RLSD state
PORT_EV_OVERRUN	Receiver buffer overrun
PORT_EV_TXCHAR	Any character transmitted
PORT_EV_CONNECTED	RFCOMM connection established
PORT_EV_CONNECT_ERR	Was not able to establish connection or disconnected
PORT_EV_FC	Flow control enabled flag changed by remote
PORT_EV_FCS	Flow control status true = enabled

virtual OnFlowEnabled ( )

Not implemented.

SwitchRole ( )

The application uses this method to request that the device switch role to Master or Slave. If the application wants to accept multiple simultaneous connections, it must execute this command to request that the device switch role to Master.

Prototype:	`BOOL SwitchRole (MASTER_SLAVE_ROLE new_role);`	
Parameters:	new_role	The role to which the device should switch. Valid values are NEW_MASTER or NEW_SLAVE.
Returns:	TRUE if successful.	

SetLinkSupervisionTimeOut ( )

This function sets the link supervision timeout value for the connected device. The timeout value is used by the master or slave Bluetooth device to monitor link loss. The same timeout value is used for both SCO and ACL connections for the device. An ACL connection has to be established before calling this function. The Zero value for timeout will disable the Link_Supervision_Timeout check for the connected device. The default timeout is 20 seconds.

Prototype:	`PORT_RETURN_CODE SetLinkSupervisionTimeOut (UINT16 timeoutSlot);`	
Parameters:	timeoutSlot	The timeout duration in number of slots. Each slot is .625 milliseconds. Note: Zero will disable Link_Supervision_Timeout check.
Returns:	See Table 12: "PORT_RETURN_CODE," on page 69.	

AUDIO CONNECTIONS

Audio connections can be established between Bluetooth devices and associated with a particular RFCOMM connection. The application uses the following methods to manage these connections. The reader is directed to the Audio Connections discussion in the CBtIf class description above for an overview of audio connections.

CreateAudioConnection ( )

This method is used to establish an audio connection with a remote device. The same method is used as both a client initiating a connection and a server listening for an incoming connection.

If used as a client to initiate an audio connection, the client must have already established the RFCOMM data connection using the *CRfCommPort::OpenClient ( )* method prior to initiating the audio connection.

If used as a server to listen for an incoming audio connection, the *CRfCommPort::OpenServer ( )* method must have already been executed to begin listening for an RFCOMM data connection prior to executing the *CRfCommPort::CreateAudioConnection ( )* method to begin listening for an audio connection.

Once the audio connection is established, it will be associated with the RFCOMM connection. As such, when the RFCOMM connection is closed, the audio connection will automatically be closed.

The application must implement *OnAudioConnected ( )* to get the connection establishment notice.

Note: Applications should not expect to manage listening connections for standard audio profile GUIDs such as Hands-free Profile. The BTW audio services are typically already in charge of listening for the standard profile GUIDs. If an application requires opening listening connections for a standard audio profile GUID, contact Broadcom Technical Support at http://www.broadcom.com/products/bluetooth_support.php.

There are multiple signatures for this method. The orignal (Basic) version was supplemented with the Enhanced version, which allows a server application to specify a particular Bluetooth device address. The server will then only accept connections from that specific address. The Enhanced version should not be used for a client connection, as the remote address must be previously known from the prerequisite call to *CRFCommPort::OpenClient ( )*.

Basic Version

Prototype:

```
AUDIO_RETURN_CODE CreateAudioConnection (
                                        BOOL      bIsClient,
                                        UINT16    *audioHandle);
```

Parameters:

bIsClient	TRUE if this is a client connection (initiator); otherwise, it is a server connection (listener).
audioHandle	Pointer to where the handle for the new audio connection should be stored.

Returns: See Table 8: "Audio Return Codes," on page 35.

Enhanced Version

Prototype:

```
AUDIO_RETURN_CODE CreateAudioConnection (
                                        BOOL      bIsClient,
                                        UINT16    *audioHandle,
                                        BD_ADDR   bda);
```

Parameters:

bIsClient	TRUE if this is a client connection (initiator); otherwise, it is a server connection (listener).
audioHandle	Pointer to where the handle for the new audio connection should be stored.
bda	Specific remote device allowed to make connections (server-only). 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF indicates a connection will be accepted from any device.

Returns: See Table 8: "Audio Return Codes," on page 35.

RemoveAudioConnection ( )

This method is used to disconnect an audio connection. Either the client or the server may use this method to disconnect the connection. Note that usage of this method is optional. The audio connection will be disconnected automatically after the RFCOMM connection is closed.

Prototype: `AUDIO_RETURN_CODE RemoveAudioConnection (UINT16 audioHandle);`

Parameters: audioHandle Handle to an open audio connection.

Returns: See Table 8: "Audio Return Codes," on page 35.

virtual OnAudioConnected ( )

This is a virtual function. It is used to indicate that an audio connection has been established. Applications may provide a derived method to process the connection establishment notice.

Prototype: `virtual void OnAudioConnected (UINT16 audioHandle);`

Parameters: audioHandle Handle to an open audio connection.

Returns: void

virtual OnAudioDisconnect ( )

This is a virtual function. It is used to indicate an audio connection has been disconnected. Applications may provide a derived method to process the disconnection notice.

Prototype:	`virtual void OnAudioDisconnect (UINT16 audioHandle);`	
Parameters:	audioHandle	Handle to the disconnected audio connection.
Returns:	void	

SetEscoMode ( )

This method is used to set up the negotiated parameters for SCO or eSCO, and set the default mode used for *CreateAudioConnection ( ).* It can be called only when there are no active (e)SCO links.

Prototype:

```
AUDIO_RETURN_CODE SetEScoMode (
                                    tBTM_SCO_TYPE       sco_mode,
                                    tBTM_ESCO_PARAMS    *p_parms);
```

Parameters:	sco_mode	The desired audio connection mode: • BTM_LINK_TYPE_SCO for SCO audio connections. • BTM_LINK_TYPE_ESCO for eSCO audio connections.
	p_parms	A pointer to the (e)SCO link settings to use.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_PARAMS structure definition is:

```
typedef struct
{
   UINT32          tx_bw;
   UINT32          rx_bw;
   UINT16          max_latency;
   UINT16          voice_contfmt;
   UINT16          packet_types;
   UINT8           retrans_effort;
} tBTM_ESCO_PARAMS;
```

Where:

- tx_bw – Transmit bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- rx_bw – Receive bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- max_latency – maximum latency. This is a value in milliseconds, from 0x0004 to 0xfffe. Set to 0xffff for "don't care" setting.
- voice_contfmt – voice content format; CSVD is supported: BTM_ESCO_VOICE_SETTING
- packet_types – the packet type; EV3 is supported: BTM_ESCO_PKT_TYPES_MASK_EV3
- retrans_effort – the retransmit effort; one of:
  - BTM_ESCO_RETRANS_OFF
  - BTM_ESCO_RETRANS_POWER
  - BTM_ESCO_RETRANS_QUALITY
  - BTM_ESCO_RETRANS_DONTCARE

RegForEScoEvts ( )

This function registers an eSCO event callback with the specified object instance. It should be used to receive connection indication events and change of link parameter events.

Prototype:	`AUDIO_RETURN_CODE RegForEScoEvts ( UINT16 audioHandle, tBTM_ESCO_CBACK *p_esco_cback);`	
Parameters:	audioHandle	The handle to an open audio connection.
	p_esco_cback	A pointer to the callback function.
Returns:	See Table 8: “Audio Return Codes,” on page 35.	

The tBTM_ESCO_CBACK definition is:

```
typedef void (tBTM_ESCO_CBACK) ( tBTM_ESCO_EVT      event,
                                 tBTM_ESCO_EVT_DATA *p_data);
```

The tBTM_ESCO_EVT definition is:

```
typedef UINT8  tBTM_ESCO_EVT;
```

The event could be one of the following values:

- BTM_ESCO_CHG_EVT – change of eSCO link parameter event.
- BTM_ESCO_CONN_REQ_EVT – connection indication event.

The tBTM_ESCO_EVT_DATA structure definition is:

```
typedef union
{
   tBTM_CHG_ESCO_EVT_DATA       chg_evt;
   tBTM_ESCO_CONN_REQ_EVT_DATA  conn_evt;
} tBTM_ESCO_EVT_DATA;

typedef struct
{
   UINT16          sco_inx;
   UINT16          rx_pkt_len;
   UINT16          tx_pkt_len;
   BD_ADDR         bd_addr;
   UINT8           hci_status;
   UINT8           tx_interval;
   UINT8           retrans_window;
} tBTM_CHG_ESCO_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- rx_pkt_len – The receive pocket length.
- tx_pkt_len – The transmit pocket length.
- bd_addr – The Bluetooth address of peer device.
- hci_status – HCI status.
- tx_interval – The transmit interval.
- retrans_window – re-transmit window.

```
typedef struct
{
   UINT16          sco_inx;
   BD_ADDR         bd_addr;
   DEV_CLASS       dev_class;
   tBTM_SCO_TYPE   link_type;
} tBTM_ESCO_CONN_REQ_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- bd_addr – The Bluetooth address the device.
- dev_class – The device class.
- link_type – The audio connection type.

ReadEScoLinkData ( )

This function retrieves current eSCO link data for the specified handle. This can be called anytime when a connection is active.

Prototype:	`AUDIO_RETURN_CODE ReadEScoLinkData ( UINT16 audioHandle, tBTM_ESCO_DATA *p_data);`	
Parameters:	audioHandle	The handle of an open audio connection.
	p_data	A pointer to the buffer where the current eSCO link settings will be stored.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_DATA structure definition is:

```
typedef struct
{
   UINT16      rx_pkt_len;
   UINT16      tx_pkt_len;
   BD_ADDR     bd_addr;
   UINT8       link_type
   UINT8       tx_interval;
   UINT8       retrans_window;
   UINT8       air_mode;
} tBTM_ESCO_DATA;
```

Where:

- rx_pkt_len – The transmit packet length.
- tx_pkt_len – The receive packet length.
- bd_addr – The device Bluetooth device address.
- link_type – the current audio connection type. It could be one of the following values:
  - BTM_LINK_TYPE_SCO
  - BTM_LINK_TYPE_ESCO
- tx_interval – The transmit interval.
- retrans_window – The retransmit window.
- air_mode – The air mode.

ChangeEscoLinkParms ( )

This function requests renegotiation of the parameters on the current eSCO link. If any of the changes are accepted by the controllers, the BTM_ESCO_CHG_EVT event is sent via the callback function registered in *RegForEScoEvts ( ),* with the current settings of the link.

Prototype:

```
AUDIO_RETURN_CODE ChangeEScoLinkParms ( UINT16               audioHandle,
                                        tBTM_CHG_ESCO_PARAMS  *p_parms);
```

Parameters:	audioHandle	The handle to an open audio connection.
	p_parms	A pointer to the settings that need to be changed.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_CHG_ESCO_PARAMS structure definition is follows:

```
typedef struct
{
    UINT16     max_latency;
    UINT16     packet_types;
    UINT8      retrans_effort;
} tBTM_CHG_ESCO_PARAMS;
```

Where:

- max_latency – maximum latency. This is a value in milliseconds.
- packet_types – packet type.
- retrans_effort – The retransmit effort.

EScoConnRsp ( )

This function should be called upon receipt of an eSCO connection request event (BTM_ESCO_CONN_REQ_EVT) to accept or reject the request.

The hci_status parameter should be [0x00] to accept, [0x0d .. 0x0f] to reject.

The p_params tBTM_ESCO_PARAMS pointer argument is used to negotiate the settings on the eSCO link. If p_parms is NULL, the values set through *SetEScoMode ( )* are used.

Prototype:

```
void EScoConnRsp (UINT16             audioHandle,
                  UINT8              hci_status,
                  tBTM_ESCO_PARAMS   *p_parms = NULL);
```

Parameters:	audioHandle	Handle to an open audio connection.
	hci_status	The HCI status. Zero is HCI success, defined as HCI_SUCCESS.
	p_parms	A pointer to the tBTM_ESCO_PARAMS to be negotiated.
Returns:	void	

CFtpClient

This class provides a client interface for the File Transport Profile. A client application must first obtain a Bluetooth server device address using the CBtIf class for inquiry and service discovery.

The CFtpClient class then provides connection and file transfer functions.

For more information on the *File Transfer Profile,* see the Bluetooth Special Interest Group, File Transfer Profile.

Special Registry Settings

On the server, there are two registry keys (HKCU\Software\Widcomm\BTConfig\Services\0005\DenyReadOnlyAccess and HKCU\Software\Widcomm\BTConfig\Services\0005\DenyHiddenAccess). The FTP server will behave in the following manner based on the settings of these keys.

HKCU\Software\Widcomm\BTConfig\Services\0005\DenyReadOnlyAccess:

- Does not exist or is equal to zero:
  - Server will ignore all read-only attributes on files and folders and treat them as if they are read-write.
- Exists and is nonzero:
  - Server will not allow read-only files to be overwritten.
  - Server will not allow read-only files or folders to be deleted.
  - Server will not allow files or subfolders to be created in read-only folders.
  - Server will not allow files or subfolders to be deleted from read-only folders.

HKCU\Software\Widcomm\BTConfig\Services\0005\DenyHiddenAccess:

- Does not exist or is equal to zero:
  - Server will ignore all hidden attributes on files and folders and treat them as if they are not hidden.
- Exists and is nonzero:
  - Server will not allow hidden files or folders to be overwritten or deleted.
  - Server will not allow hidden files to be read (GET).
  - Server will not allow hidden folders to be browsed (Change Directory into).
  - Server will not allow files or subfolders to be created in hidden folders.
  - Server will not allow files or subfolders to be deleted from hidden folders.

FTP_RETURN_CODE

A common set of return codes is provided by the direct FTP function calls – *OpenConnection ( )*, *Get ( )*, *Put ( )*, etc. See the enumerated type FTP_RETURN_CODE in BtIfClasses.h

Table 17: FTP_RETURN_CODE

FTP_RETURN_CODE Value	*Meaning.*
SUCCESS	Operation initiated without error.
OUT_OF_MEMORY	Not enough memory to initiate operation.
SECURITY_ERROR	Error implementing requested security level.
FTP_RETURN_ERROR	FTP-specific error.
NO_BT_SERVER	Cannot access the local Bluetooth COM server.
FILE_NOT_FOUND	No such file or directory.
FILE_SHARING	File sharing violation.
ALREADY_CONNECTED	Only one connection at a time is supported for each instantiated CFtpClient object.
NOT_OPENED	Connection must be opened before requesting this operation.
INVALID_PARAMETER	One or more function parameters are not valid.
UNKNOWN_RETURN_ERROR	Any condition other than the above.
LICENSE_ERROR	License error.

OpenConnection ( )

The client calls this method to connect to the FTP server. The *OnOpenResponse ( )* function will be called with the server’s response.

Prototype:
```
FTP_RETURN_CODE OpenConnection (
                        BD_ADDR          bdAddr,
                        CSdpDiscoveryRec &sdp_rec);
```

Parameters:

bdAddr	The FTP server’s BT device address, from the service discovery process.
sdp_rec	The service discovery record.

Returns: SUCCESS – Everything is OK. Otherwise, see Table 17: “FTP_RETURN_CODE,” on page 82.

CloseConnection ( )

This function closes the connection. The *OnCloseResponse ( )* function will be called with the server’s response.

Prototype:
```
FTP_RETURN_CODE CloseConnection ( );
```

Parameters: None

Returns: SUCCESS – Everything is OK. Otherwise, see Table 17: “FTP_RETURN_CODE,” on page 82.

PutFile ( )

This function sends a file to the server. The *OnPutResponse ( )* function will be called with the server's response.

Prototype:	`FTP_RETURN_CODE PutFile (WCHAR *localFileName);`	
Parameters:	localFileName	The name of the file to be sent to server's current folder. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.	

GetFile ( )

This function requests a file from the server. The *OnGetResponse ( )* function will be called with the server's response.

Prototype:

```
FTP_RETURN_CODE GetFile (
                    WCHAR *remoteFileName,
                    WCHAR *localFolder);
```

Parameters:	remoteFileName	The name of the source file on the server. If this parameter is NULL, error INVALID_PARAMETER is returned.
	localFolder	The name of the destinationlocal folder on the client. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.	

FolderListing ( )

This function requests a list of the file names in the server's current directory. Both the *OnFolderListingResponse ( )* and the *OnXmlFolderListingResponse ( )* functions will be called with the server's response.

The application can provide one or both of these response functions.

Prototype:	`FTP_RETURN_CODE FolderListing ( );`
Parameters:	None.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.

ChangeFolder ( )

This function requests that the server change the current folder location to a subfolder of the current folder.

Note: This operation does not change the folder name; it just makes the named folder the new current folder.

When the operation is complete, *OnChangeFolderResponse ( )* will be called with the result.

Prototype:	`FTP_RETURN_CODE ChangeFolder (WCHAR *szFolder);`	
Parameters:	szFolder	The name of the new server folder. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.	

DeleteFile ( )

This function requests that the server delete a file or folder in the server's current folder.

When the operation is complete, *OnDeleteResponse ( )* will be called with the result.

Note: Only an empty folder may be deleted in the server.

Prototype:	`FTP_RETURN_CODE DeleteFile (WCHAR *szFile);`	
Parameters:	szFile	The name of the file or folder to be deleted. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.	

Abort ( )

This function aborts an operation. The *OnAbortResponse ( )* function will be called with the server's response.

Prototype:	`FTP_RETURN_CODE Abort ( );`
Parameters:	None.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.

Parent ( )

This function requests that the server make the parent of the current folder the new current folder.

When the operation is complete, *OnChangeFolderResponse ( )* will be called with the result.

Prototype:	`FTP_RETURN_CODE Parent ( );`
Parameters:	None
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.

Root ( )

This function requests that the server make the FTP root folder the new current folder.

When the operation is complete, *OnChangeFolderResponse ( )* will be called with the result.

Prototype:	`FTP_RETURN_CODE Root ( );`
Parameters:	None
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.

CreateEmpty ( )

This function requests that the server create an empty file in the current folder.

When the operation is completed by the server, *OnCreateResponse ( )* will be called with a result code.

Prototype:	`FTP_RETURN_CODE CreateEmpty (WCHAR *szFile);`	
Parameters:	szFile	The name of the file to be created. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.	

CreateFolder ( )

This function requests that the server create a new folder, as a subfolder in the current folder.

When the operation is completed by the server, *OnCreateResponse ( )* will be called with a result code.

When the operation is successful, the created folder becomes the current folder on the server.

Prototype:	`FTP_RETURN_CODE CreateFolder (WCHAR *szFolder);`	
Parameters:	szFolder	The name of the new folder. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 17: "FTP_RETURN_CODE," on page 82.	

SetSecurity ( )

This function sets authentication and encryption parameters for OBEX FTP connections.

Encrypting translates data into an unreadable format using a secret key or password. Decrypting the data requires the same key or password that was used to encrypt it.

Encryption in Bluetooth for Windows is based on the same passkey or Link Key that is used for Authentication. If Authentication is not enabled, the key is not available and encryption will not take place.

To use Encryption, Authentication must be enabled.

If this function is not called the OBEX FTP security settings in the Windows Registry will be used. These settings are used by BTExplorer and controlled from the Advanced Configuration -> Client Applications tab in the BTTray application.

Prototype:

```
void SetSecurity (
                 BOOL authentication,
                 BOOL encryption);
```

Parameters:	authentication	TRUE means use authentication procedures on future connections using this object.
	encryption	TRUE means use encryption procedures on data transfers.
Returns:	void	

GetExtendedError ( )

This function returns the object's last-error code. The last-error code is maintained on a per-object basis. Multiple objects do not overwrite each other's last-error code.

This function is intended for obtaining extended error information when the SDK return/result codes are not sufficient. This is useful when the UNKNOWN_RETURN_ERROR code is returned from an SDK API function, or the UNKNOWN_RESULT_ERROR code is returned in an SDK callback function.

To retrieve an extended error code of a method of this class in the case where the method has failed, call *GetExtendedError ( )* directly after the failed method has been called. *GetExtendedError ( )* can be called in callback functions of this class as well.

Prototype:	`WBtRc GetExtendedError ( );`
Parameters:	None
Returns:	See definition of the WBtRc enumeration in com_error.h

SetExtendedError ( )

This function sets the last-error code for the object of this class. *SetExtendedError ( )* is intended to localize the place where exactly an error occurs in the code. It can be called to reset the last-error code for an object of this class to WBT_SUCCESS (see com_error.h) before any other class method is called. After a class method has been called, *GetExtendedError ( )* can be invoked to read the error information. If *GetExtendedError ( )* reports WBT_SUCCESS, then the error occurs somewhere else in the program.

The last-error code is kept in a local variable of the object so that multiple objects of this class do not overwrite each other's values.

Prototype:	`void SetExtendedError (WBtRc code);`	
Parameters:	code	One of the values from the WBtRc enumeration defined in com_error.h
Returns:	void	

FTP_RESULT_CODE

A common set of result codes is provided by the callback FTP functions. Use the enumerated type FTP_RETURN_CODE, from BtIfClasses.h.

Table 18: FTP_RESULT_CODE

FTP_RESULT_CODE Value	*Meaning*
COMPLETED	Operation completed without error.
BAD_ADDR	Bad Bluetooth device address.
FILE_EXISTS	File already exists.
BAD_STATE	Could not handle the request in present state.
BAD_REQUEST	Invalid request.
NOT_FOUND	No such file.
NO_SERVICE	Could not find the specified FTP server.

Table 18: FTP_RESULT_CODE

FTP_RESULT_CODE Value	*Meaning*
DISCONNECT	Connection lost.
READ	Read error.
WRITE	Write error.
OBEX_AUTHEN	OBEX Authentication is required.
DENIED	Request could not be honored.
DATA_NOT_SUPPORTED	Server does not support the requested data.
CONNECT	Error establishing the connection.
PERMISSIONS	Prohibited by file permissions.
NOT_INITIALIZED	Not initialized.
PARAM	Bad parameter.
RESOURCES	A file system resource limit has been reached — may be file handles, disk space, etc.
SHARING	File sharing violation.
UNKNOWN_RESULT_ERROR	Any condition other than the above.

virtual OnProgress ( )

Client applications may provide a function to handle progress reports from the server.

This function is called each time a block is sent or received over the Bluetooth connection. A block is in most cases no more than the MTU bytes in length.

Prototype:

```
virtual void OnProgress (
                    FTP_RESULT_CODE  result_code,
                    WCHAR            *name,
                    long             current,
                    long             total);
```

Parameters:

result_code	See Table 18: "FTP_RESULT_CODE," on page 86.
name	The file/folder name being processed.
current	The number of bytes transferred so far in the current file transfer operation.
total	The total number of bytes to be transferred in the current operation (i.e., the file length).

Returns: void

virtual OnOpenResponse ( )

The client application may provide a function to handle server responses to the *OpenConnection ( )* request. In most applications, the developer will provide functions for *Open*, *Close* and other functions that are actually used by the application. In any event, the SDK provides default response handlers, which take no action.

Prototype:

```
virtual void OnOpenResponse (FTP_RESULT_CODE result_code);
```

Parameters: result_code — See Table 18: "FTP_RESULT_CODE," on page 86.

Returns: void

virtual OnCloseResponse ( )

The client application may provide a function to handle server responses to the *CloseConnection ( )* request. In most applications the developer will provide functions for *Open*, *Close* and other functions that are actually used by the application. In any event, the SDK provides default response handlers, which take no action.

Prototype:	`virtual void OnCloseResponse (FTP_RESULT_CODE result_code);`	
Parameters:	result_code	See Table 18: "FTP_RESULT_CODE," on page 86.
Returns:	void	

virtual OnPutResponse ( )

If the *put* function is used, client applications should provide a function to handle the put response event from the server. Otherwise, a default handler is provided in CFtpClient to ignore a put response from the server.

Prototype:

```
virtual void OnPutResponse (
                              FTP_RESULT_CODE  result_code,
                              WCHAR            *name);
```

Parameters:	result_code	See Table 18: "FTP_RESULT_CODE," on page 86.
	name	The file name.
Returns:	void	

virtual OnGetResponse ( )

If the *get* function is used, client applications should provide a function to handle the get response event from the server. Otherwise, a default handler is provided in CFtpClient to ignore a get response from the server.

Prototype:

```
virtual void OnGetResponse (
                              FTP_RESULT_CODE  result_code,
                              WCHAR            *name);
```

Parameters:	result_code	See Table 18: "FTP_RESULT_CODE," on page 86.
	name	The file name.
Returns:	void	

virtual OnCreateResponse ( )

If one of the create functions (*CreateEmpty ( )* or *CreateFolder ( )*) is to be used, the client application should provide this function to handle the create response from the server. Otherwise, a default handler is provided in CFtpClient to ignore a create response from the server.

Prototype:

```
virtual void OnCreateResponse (
                              FTP_RESULT_CODE  result_code,
                              WCHAR            *name);
```

Parameters:	result_code	See Table 18: "FTP_RESULT_CODE," on page 86.
	name	The created file name in response to a CreateEmpty call or an empty string in response to a CreateFolder call.
Returns:	void	

virtual OnDeleteResponse ( )

If the delete function *DeleteFile ( )* is to be used, the client application should provide this function to handle the delete response from the server. Otherwise, a default handler is provided in CFtpClient to ignore a delete response from the server.

Prototype:

```
virtual void OnDeleteResponse (
                              FTP_RESULT_CODE  result_code,
                              WCHAR            *name);
```

Parameters:	result_code	See Table 18: "FTP_RESULT_CODE," on page 86.
	name	The file name or the folder name deleted.
Returns:	void	

virtual OnChangeFolderResponse ( )

If any of the functions *ChangeFolder ( )*, *Parent ( )*, or *Root ( )* are to be used, the client application should provide this function to handle the response from the server. Otherwise, a default handler is provided in CFtpClient to ignore a change folder response from the server.

This function is called in response to a previous call to *ChangeFolder ( ), Parent ( ),* or *Root ( )*.

Prototype:

```
virtual void OnChangeFolderResponse (
                                    FTP_RESULT_CODE  result_code,
                                    tFtpFolder       folder_type,
                                    WCHAR            *szFolder);
```

Parameters:	result_code	See Table 18: "FTP_RESULT_CODE," on page 86.
	folder_type	A value from enumerated type tFtpFolder – one of the values: FTP_ROOT_FOLDER = 1, FTP_PARENT_FOLDER, FTP_SUBFOLDER, in BtIfClasses.h.
	szFolder	Name of the current folder in the server. When the change folder command was called for a folder within the current folder (down the directory tree), the name is the name of the subfolder, which now becomes the server's current folder. If the parent or root function was called, the result is the text string ".."
Returns:	void	

virtual OnFolderListingResponse ( )

This function provides a listing of the current server folder, in the form of an array of structures, one file entry per array element.

If the *FolderListing ( )* function is to be used, the client application should provide either this function or the *OnXmlFolderListingResponse ( )* function to receive the response from the server. Otherwise, default functions are provided in CFtpClient to ignore a folder listing response from the server.

In this format, each array element is a structure tFTP_FILE_ENTRY, which contains:

- File name
- A file/folder flag
- Date created – in ISO 8601 format
- Date last modified – in ISO 8601 format
- Date last accesed – in ISO 8601 format
- File size
- Permissions characters

The definition of the structure is as follows:

```
#define MAX_NAME_SIZE   255
#define MAX_ENTRY_SIZE  1000
#define DATE_TIME_SIZE  15

typedef struct
{
    WCHAR     name[MAX_NAME_SIZE + 1];
    BOOL      is_folder;
    WCHAR     date_created[DATE_TIME_SIZE + 1];
    WCHAR     date_modified[DATE_TIME_SIZE + 1];
    WCHAR     date_accessed[DATE_TIME_SIZE + 1];
    ULONG     file_size;

#define FTP_READ_PERM       (0x01)
#define FTP_WRITE_PERM      (0x02)
#define FTP_DELETE_PERM     (0x04)

    WCHAR    user_perm;
    WCHAR    group_perm;
    WCHAR    other_perm;
} tFTP_FILE_ENTRY;
```

This information is in volatile memory; the application must use it within the response function or store it safely before returning.

Prototype:

```
virtual void OnFolderListingResponse (
                    FTP_RESULT_CODE  result_code,
                    tFTP_FILE_ENTRY  *listing,
                    long             entries);
```

Parameters:	result_code	See Table 18: "FTP_RESULT_CODE," on page 86.
	listing	A pointer to an array of structures of type tFTP_FILE_ENTRY. See BtIfDefinitions.h. In this structure the file name size is limited by MAX_NAME_SIZE, and the date fields are limited to length by DATE_TIME_SIZE. The format of the date fields is ISO 8601 - yyyymmddThhmmss, as in 20011225T235959.
	entries	The number of entries (files/folders) in the array.
Returns:	void	

virtual OnXmlFolderListingResponse ( )

If the *FolderListing ( )* function is to be used, the client application should provide either this function or the *OnFolderListingResponse ( )* function to receive the response from the server. Otherwise, default functions are provided in CFtpClient to ignore a folder listing response from the server.

This function provides a listing of the current server folder, in the form of a sequence of XML strings.

This information is in volatile memory; the application must use it within the response function or store it safely before returning.

The sequence of items returned is in complete XML format. For example, for a server folder having two files, this function would be called six times, with the following contents:

```
<?xml version = "1.0"?>
<!DOCTYPE folder-listing SYSTEM "obex-folder-listing.dtd">
<folder-listing version = "1.0">
<file name = "BT SDK User's Guide.doc" size = "323584" user-perm = "RWD" modified =
"20010906T152825Z" created = "20010906T152301Z" accessed = "20011002T142525Z"/>
<file name = ”File X.doc" size = "323584" user-perm = "RWD" modified = "20010906T152825Z"
created = "20010906T152301Z" accessed = "20011002T142525Z"/>
</folder-listing>
```

Prototype:

```
void OnXmlFolderListingResponse (
                    FTP_RESULT_CODE  rc,
                    WCHAR            *pfolder_listing,
                    long             folder_length );
```

Parameters:	rc	See Table 18: “FTP_RESULT_CODE,” on page 86.
	pfolder_listing	A pointer to a NULL-terminated character string, formatted in XML, which contains the folder listing information.
	folder_length	The number of characters in the string.
Returns:	void	

virtual OnAbortResponse ( )

If the *Abort ( )* function is to be used, client applications should provide a function to handle the abort response event from the server. Otherwise, a default handler is provided in CFtpClient to ignore an abort response from the server.

Prototype: `virtual void OnAbortResponse (FTP_RESULT_CODE result_code);`

Parameters:	result_code	See Table 18: “FTP_RESULT_CODE,” on page 86.
Returns:	void	

COPPCLIENT

This class provides a client interface for the Object Push Protocol. Before using this class to provide object transfer functions, the client application must obtain a Bluetooth server device address using the CBtIf class for inquiry and service discovery.

There are no explicit connection functions for this service. Each operation, such as push or pull, has an implicit connection, object transfer, and implicit disconnect. An enhanced version of this OPP client class, COppMultiPush (see "COppMultiPush" on page 99), has been created to allow for multiple object transfer over a single connection. That class provides explicit connection functions, but supports only the Push operation. While the two classes share many concepts and have similar functionality, they must operate in a distinctly separate context - do not attempt to create a multiple-inheritance class using the two classes, or try to use COppClient methods in conjunction with a COppMultiPush connection.

As implemented on Windows, the Bluetooth objects supported by this class are files, one per Bluetooth object. The file extension(s) for each object type are:

- Business card:
  - vcd
  - vcf
- Calendar:
  - vcs for Version 1.0
  - ics for Version 2.0
- Note:
  - vnt
- Message:
  - vmg

The OPP server will accept all file types — business card, calendar, note, and message — that are pushed by the client.

Note: Incoming objects are sent either to the local PIM or to the designated folder. See the Bluetooth configuration dialog, *Information Exchange* tab, where the user can specify which types of object will be accepted and whether they will be directed to the PIM or to a designated folder.

The OPP server will send only business cards pulled by the client, and then only if the server is configured to do so. The server's Bluetooth configuration function in the *Objects* tab must have the *My Business Card* field filled with an absolute path and file name for the server's business card file. The *Send Business Card on request* option must also be selected.

For more information on the *Object Push Profile,* see the Bluetooth Special Interest Group, Object Push Profile.

OPP_RETURN_CODE

A common set of return codes is provided by the direct OPP function calls, *Push ( )*, *Pull ( )*, *Exchange ( )*, and *Abort ( )*. They are defined with enumerated type OPP_RETURN_CODE, from BtIfClasses.h.

Table 19: OPP_RETURN_CODE

OPP_RETURN_CODE Value	*Meaning*
OPP_CLIENT_SUCCESS	Operation initiated without error.
OUT_OF_MEMORY	Not enough memory to initiate operation.
SECURITY_ERROR	Error implementing requested security level.
OPP_ERROR	OPP-specific error.
OPP_NO_BT_SERVER	Cannot access the local Bluetooth COM server.
ABORT_INVALID	Abort not valid, no operation is in progress.
INVALID_PARAMETER	One or more of function parameters are not valid.
UNKNOWN_ERROR	Any condition other than the above.

Push ( )

This function requests that a local object be transferred to the server. The OPP server will accept only business cards, calendars, notes, and messages.

Prototype:

```
OPP_RETURN_CODE Push (
                    BD_ADDR             bda,
                    WCHAR               *pszPathName,
                    CSdpDiscoveryRec    &sdp_rec);
```

Parameters:	bda	The Bluetooth device address of the server. If this parameter is NULL, error INVALID_PARAMETER is returned.
	pszPathName	The local path and file name for the object to be sent to the server—a null terminated string. This must be an absolute file path. If this parameter is NULL, error INVALID_PARAMETER is returned.
	sdp_rec	The OPP service discovery record obtained from the OPP server.
Returns:	See Table 19: "OPP_RETURN_CODE," on page 93.	

Pull ( )

This function requests that a server object be transferred to a local file. The OPP server will send only a business card file, and then only if properly configured – see above.

Prototype:

```
OPP_RETURN_CODE Pull (
                    BD_ADDR             bda,
                    WCHAR               *pszPathName,
                    CSdpDiscoveryRec    &sdp_rec);
```

Parameters:	bda	The Bluetooth device address of the server. If this parameter is NULL, error INVALID_PARAMETER is returned.
	pszPathName	Currently not used. Vcard will be save to Bluetooth exchange folder from Control panel->Local Service ->PIM Item Transfer property page.
	sdp_rec	The OPP Service discovery record obtained from the OPP server.
Returns:	See Table 19: "OPP_RETURN_CODE," on page 93.	

Exchange ( )

This function requests the exchange of business card objects between the server and the client. The OPP server must be properly configured, as described above.

Prototype:	`OPP_RETURN_CODE Exchange (`	
	`BD_ADDR`	`bda,`
	`WCHAR`	`*pszName,`
	`WCHAR`	`*pszFolder,`
	`CSdpDiscoveryRec`	`&sdp_rec);`
Parameters:	bda	The Bluetooth device address of the server. If this parameter is NULL, error INVALID_PARAMETER is returned.
	pszName	The local path and file name for the business card object to be sent to server—a null terminated string. This must be an absolute path and file name. If this parameter is NULL, error INVALID_PARAMETER is returned.
	pszFolder	Not used now. Vcard will be save to Bluetooth exchange folder from Control panel->Local Service ->PIM Item Transfer property page.
	sdp_rec	The OPP Service discovery record obtained from the OPP server.
Returns:	See Table 19: "OPP_RETURN_CODE," on page 93.	

Abort ( )

This function requests that the current operation be aborted.

Prototype:	`OPP_RETURN_CODE Abort ( );`
Parameters:	None
Returns:	See Table 19: "OPP_RETURN_CODE," on page 93.

SetSecurity ( )

This function sets the authentication and encryption parameters for OBEX OPP operations.

Encrypting translates data into an unreadable format using a secret key or password. Decrypting the data requires the same key or password that was used to encrypt it.

Encryption in Bluetooth for Windows is based on the same passkey or Link Key that is used for Authentication. If Authentication is not enabled, the key is not available, and encryption will not take place.

To use Encryption, Authentication must be enabled.

If this function is not called, the OBEX OPP PIM Item Transfer security settings in the Windows Registry will be used. These settings are used by BTExplorer and controlled from the Advanced Configuration -> Client Applications tab in the BTTray application.

Prototype:

```
void SetSecurity (
                 BOOL authentication,
                 BOOL encryption);
```

Parameters:	authentication	TRUE means use authentication procedures on future operations using this object.
	encryption	TRUE means use encryption procedures on data transfers that use this connection.
Returns:	void	

GetExtendedError ( )

This function returns the object's last-error code. The last-error code is maintained on a per-object basis. Multiple objects do not overwrite each other's last-error code.

This function is intended for obtaining extended error information when the SDK return/result codes are not sufficient. This is useful when the UNKNOWN_RETURN_ERROR code is returned from an SDK API function or the UNKNOWN_RESULT_ERROR code is returned in an SDK callback function.

To retrieve an extended error code of a method of this class in the case where the method has failed, call *GetExtendedError ( )* directly after this method has been called. *GetExtendedError ( )* can be called in callback functions of this class as well.

Prototype:	`WBtRc GetExtendedError ( );`
Parameters:	None
Returns:	See definition of the WBtRc enumeration in com_error.h

SetExtendedError ( )

This function sets the last-error code for the object of this class. *SetExtendedError ( )* is intended to localize the place where exactly an error occurs in the code. It can be called to reset the last-error code for an object of this class to WBT_SUCCESS (see com_error.h) before any other class method is called. After a class method has been called, *GetExtendedError ( )* can be invoked to read the error information. If *GetExtendedError ( )* reports WBT_SUCCESS, then the error occurs somewhere else in the program.

The last-error code is kept in a local variable of the object so that multiple objects of this class do not overwrite each other's values.

Prototype:	`void SetExtendedError (WBtRc code);`	
Parameters:	code	One of the values from the WBtRc enumeration defined in com_error.h
Returns:	void	

OPP_RESULT_CODE

A common set of result codes is provided by the OPP response functions for push, pull, exchange, and abort. Use enumerated type OPP_RESULT_CODE, from BtIfClasses.h.

Table 20: OPP_RESULT_CODE

OPP_RESULT_CODE Value	*Meaning*
COMPLETED	Operation completed without error.
BAD_ADDR	Bad Bluetooth device address.
BAD_STATE	Could not handle the request in the present state.
BAD_REQUEST	Invalid request.
NOT_FOUND	No such file.
NO_SERVICE	Could not find the specified FTP server.
DISCONNECT	Connection lost.
READ	Read error.
WRITE	Write error.
OBEX_AUTH	OBEX authentication required.
DENIED	Request could not be honored.
DATA_NOT_SUPPORTED	Server does not support the requested data.
CONNECT	Error establishing the connection.
NOT_INITIALIZED	Not initialized.
PARAM	Bad parameter.
BAD_INBOX	Inbox is not valid.
BAD_NAME	Bad object name.
PERMISSIONS	Prohibited by file permissions.
SHARING	File is shared.
RESOURCES	File system resource limit has been reached — may be file handles, disk space, etc.
FILE_EXISTS	File already exists.
UNKNOWN_RESULT_ERROR	Any condition other than the above.

virtual OnProgress ( )

This function is called each time a block is sent or received over the Bluetooth connection. A block is in most cases no more than the MTU bytes in length. Client applications may provide a function to handle progress reports from the server.

Prototype:

```
virtual void OnProgress (
                    OPP_RESULT_CODE  result_code,
                    BD_ADDR          bda,
                    WCHAR            *string,
                    long             current,
                    long             total);
```

Parameters:	result_code	See Table 20: "OPP_RESULT_CODE," on page 96.
	bda	The Bluetooth device address of the server.
	string	The file/folder name being transferred.
	current	The total number of bytes transferred so far in the current file transfer operation.
	total	The total number of bytes to be transferred in the current operation, the file length.
Returns:	void	

virtual OnPushResponse ( )

If the *Push ( )* function is used, client applications should provide a function to handle the push response event from the server. Otherwise, a default handler is provided in COppClient to ignore a push response from the server.

Prototype:

```
virtual void OnPushResponse (
                    OPP_RESULT_CODE  result_code,
                    BD_ADDR          bda,
                    WCHAR            *string);
```

Parameters:	result_code	See Table 20: "OPP_RESULT_CODE," on page 96.
	bda	The address of the remote device.
	string	The name of the file being pushed.
Returns:	void	

virtual OnPullResponse ( )

If the *Pull ( )* function is used, client applications should provide a function to handle the pull response event from the server. Otherwise, a default handler is provided in COppClient to ignore a pull response from the server.

Prototype:

```
virtual void OnPullResponse (
                    OPP_RESULT_CODE  result_code,
                    BD_ADDR          bda,
                    WCHAR *          file_name);
```

Parameters:	result_code	See Table 20: "OPP_RESULT_CODE," on page 96.
	bda	The address of the remote device.
	file_name	The name of the file being pulled.
Returns:	void	

virtual OnExchangeResponse ( )

If the *Exchange ( )* function is used, client applications should provide a function to handle the exchange response event from the server. Otherwise, a default handler is provided in COppClient to ignore an exchange response from the server.

There are multiple signatures for this method. The first contains basic information, the second also includes an OPP transaction code for applications that require more information.

Basic Version

Prototype:

```
virtual void OnExchangeResponse (
                       OPP_RESULT_CODE  result_code,
                       BD_ADDR          bda,
                       WCHAR            *string);
```

Parameters:	result_code	See Table 20: "OPP_RESULT_CODE," on page 96.
	bda	The address of the remote device.
	string	The name of the file being transferred to server.
Returns:	void	

Enhanced Version

This version returns a transaction code which identifies the operation the result applies to. For the exchange operation there are two activations of this function, one each for the GET and the PUT. This allows the application to distinguish which operation had a problem in the case where one succeeds and one fails.

The transaction codes are defined in the enumerated type OPP_TRANSACTION_CODE.

```
typedef enum
{
   OPP_PUT_TRANS = 1,
   OPP_GET_TRANS = 2,
   OPP_EXCHANGE_PUT_TRANS = 3,
   OPP_EXCHANGE_GET_TRANS = 4,
   OPP_ABORT_TRANS = 5

} OPP_TRANSACTION_CODE;
```

Prototype:

```
virtual void OnExchangeResponse (
                          OPP_RESULT_CODE        result_code,
                          BD_ADDR                bda,
                          WCHAR                  *string,
                          OPP_TRANSACTION_CODE   transaction_code);
```

Parameters:	result_code	See the definition of OPP_RESULT_CODE.
	bda	The address of the remote device.
	string	The name of the file being transferred to server.
	transaction_code	Indicates the transaction to which the result_code applies. See definition of OPP_TRANSACTION_CODE above.
Returns:	void	

virtual OnAbortResponse ( )

When the *Abort ( )* function is used, client applications should provide a function to handle the abort response event from the server. Otherwise, a default handler is provided in COppClient to ignore an abort response from the server.

Prototype:	`virtual void OnAbortResponse (OPP_RESULT_CODE result_code);`	
Parameters:	result_code	See Table 20: "OPP_RESULT_CODE," on page 96.
Returns:	void	

COPPMULTIPUSH

This class is only added as an alternative to the COppClient class (see "COppClient" on page 92) for performing multiple object pushes. As such, explicit connection management functions are available in this class, as opposed to the original COppClient. While the two classes share many concepts and have similar functionality, they must operate in a distinctly separate context - do not attempt to create a multiple-inheritance class using the two classes, or try to use COppClient methods in conjunction with a COppMultiPush connection.

OpenOppConnection ( )

This function creates an OPP connection to a remote device. To obtain the connection status, the application must register the callback by calling the RegOppMultiOpenCB () function.

Prototype:

```
OPP_RETURN_CODE OppOpenConnection (
                                  BD_ADDR          bda,
                                  CSdpDiscoveryRec &sdp_rec);
```

Parameters:	bda	The server's Bluetooth device address. If this parameter is NULL, an INVALID_PARAMETER error is returned.
	sdp_rec	OPP service discovery record obtained from the OPP server.
Returns:	SUCCESS - Everything is OK. Otherwise, see Table 20: "OPP_RESULT_CODE," on page 96.	

CloseOppConnection ( )

This function closes the connection. The RegOppMultiCloseCB () must be called to register the callback.

Prototype:	`OPP_RETURN_CODE CloseOppConnection ();`
Parameters:	None
Returns:	SUCCESS - Everything is OK. Otherwise, see Table 20: "OPP_RESULT_CODE," on page 96.

MultiPush ( )

This function is used to push the object on an opened connection. OpenOppConnection () must be called before calling this function. To get the status, the application must register the callback by calling the RegOppMultiPushCB () function.

Prototype:	`OPP_RETURN_CODE MultiPush (WCHAR * pszPathName);`	
Parameters:	pszPathName	The local path and file name for the object to be sent to the server- a null terminated string. This must be an absolute file path. If this parameter is NULL, error INVALID_PARAMETER is returned.
Returns:	SUCCESS - Everything is OK. Otherwise, see Table 20: "OPP_RESULT_CODE," on page 96.	

RegOppMultiOpenCB ( )

This function is used to register a callback to receive events when a connection to the OPP server is established. The function needs to be called before calling OpenOppConnection ().

Prototype:	`OPP_RETURN_CODE RegOppMultiOpenCB (tOnOppMultiOpenCB pOnOppMultiOpenCB);`	
Parameters:	pOnOppMultiOpenCB	Pointer to the callback function.
Returns:	SUCCESS - Everything is OK. Otherwise, see Table 20: "OPP_RESULT_CODE," on page 96.	

The callback function prototype is defined in the header file BtIfClasses.h as follows:

Prototype:	`typedef void (*tOnOppMultiOpenCB)(long lError);`	
Parameters:	lError	OPP result code, defined in OPP_RESULT_CODE.
Returns:	void	

RegOppMultiCloseCB ( )

This function is used to register a callback to receive events when a connection to the OPP server is closed. The function must be called before calling CloseOppConnection ().

Prototype:	`OPP_RETURN_CODE RegOppMultiCloseCB (tOnOppMultiCloseCB pOnOppMultiCloseCB);`	
Parameters:	pOnOppMultiCloseCB	Pointer to the callback function.
Returns:	SUCCESS - Everything is OK. Otherwise, see Table 20: "OPP_RESULT_CODE," on page 96.	

The callback function prototype is defined in the header file BtIfClasses.h as follows:

Prototype:	`typedef void (*tOnOppMultiCloseCB)(long lError);`	
Parameters:	lError	OPP result code, defined in OPP_RESULT_CODE.
Returns:	void	

RegOppMultiPushCB ( )

This function is used to register a callback to receive events when a object is pushed to to the OPP server is closed. The function must be called before calling MultiPush ().

Prototype:

```
OPP_RETURN_CODE RegOppMultiPushCB (
                              tOnOppMultiPushCB pOnOppMultiPushCB);
```

Parameters: pOnOppMultiPushCB Pointer to the callback function.

Returns: SUCCESS - Everything is OK. Otherwise, see Table 20: "OPP_RESULT_CODE," on page 96.

The callback function prototype is defined in the header file BtIfClasses.h as follows:

Prototype:

```
typedef void (*tOnOppMultiPushCB) (
                                    long        lOPPHandle,
                                    BD_ADDR     bda,
                                    LPCWSTR     pszName
                                    long        lError);
```

Parameters:	lOPPHandle	OPP connection handle
	bda	Server's Bluetooth device address
	pszName	The local path and file name for the object sent to the server
	lError	The OPP result code, defined in OPP_RESULT_CODE
Returns:	void	

CLAPCLIENT

This class allows the application to establish IP access via a PPP link.

Before a connection can be established, the client application must obtain a Bluetooth LAN server device address using the CBtIf class for inquiry and service discovery. This class provides the create connection, remove connection, and state change event functions.

This class is a base class. A pure virtual method, *OnStateChange ( )*, is defined to process connection state changes. The application must provide a derived class that defines the state change method for the application.

For more information on the *Link Access Protocol* see the Infrared Data Association, “Serial Infrared Link Access Protocol (IrLAP)”, Version 1.0, June 23, 1994.

LAP_RETURN_CODE

A common set of return codes is provided by the LAP function calls, which are defined with enumerated type LAP_RETURN_CODE, from BtIfClasses.h.

Table 21: LAP_RETURN_CODE

LAP_RETURN_CODE Value	*Meaning*
SUCCESS	Operation initiated without error.
NO_BT_SERVER	COM server could not be started.
ALREADY_CONNECTED	Attempt to connect before the previous connection closed.
NOT_CONNECTED	Attempt to close an unopened connection.
NOT_ENOUGH_MEMORY	Local processor could not allocate memory for open.
INVALID_PARAMETER	One or more of function parameters are not valid.
UNKNOWN_ERROR	Any condition other than the above.
LICENSE_ERROR	License error.

CreateConnection ( )

There are two versions of this function. The first version requests a PANU/NAP/GN connection to the server.

PANU/NAP/GN Version

Prototype:

```
LAP_RETURN_CODE CreateConnection (
                                BD_ADDR             bda,
                                GUID                guid,
                                CSdpDiscoveryRec    &sdp_rec);
```

Parameters:

bda	The server's Bluetooth device address. If this parameter is NULL, error INVALID_PARAMETER is returned.
guid	GUID for the connection type. Could be one of the following values: • CBtIf::guid_SERVCLASS_PANU • CBtIf::guid_SERVCLASS_NAP • CBtIf::guid_SERVCLASS_GN
sdp_rec	The service discovery record obtained for LAP server. Not used now.

Returns: SUCCESS – Everything is OK. Otherwise, see Table 21: "LAP_RETURN_CODE," on page 102.

PPP Version

The second version of this function requests a PPP connection to the server. Before calling this function, the application must use a CBtIf class to locate a LAN using PPP service.

Prototype:

```
LAP_RETURN_CODE CreateConnection (
                              BD_ADDR           bda,
                              CSdpDiscoveryRec &sdp_rec);
```

Parameters:

bda	The server's Bluetooth device address. If this parameter is NULL, error INVALID_PARAMETER is returned.
sdp_rec	The service discovery record obtained for LAP server. Not used now.

Returns: SUCCESS – Everything is OK. Otherwise, see Table 21: "LAP_RETURN_CODE," on page 102.

RemoveConnection ( )

This function closes the connection.

Prototype: `LAP_RETURN_CODE RemoveConnection ( );`

Parameters: None

Returns: SUCCESS – Everything is OK. Otherwise, see Table 21: "LAP_RETURN_CODE," on page 102.

GetConnectionStats ( )

This function retrieves current connection statistics, using the common statistics structure tBT_CONN_STATS defined in BtIfDefinitions.h. See "GetConnectionStats ( )" on page 34.

This class uses all the statistic fields in the structure.

Prototype:	`LAP_RETURN_CODE GetConnectionStats (tBT_CONN_STATS *p_conn_stats);`	
Parameters:	p_conn_stats	A pointer to the user's connection statistics structure; see above.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 21: "LAP_RETURN_CODE," on page 102.	

SetSecurity ( )

This function sets the authentication and encryption parameters for LAP via PPP operations.

Encrypting translates data into an unreadable format using a secret key or password. Decrypting the data requires the same key or password that was used to encrypt it.

Encryption in Bluetooth for Windows is based on the same passkey or Link Key that is used for Authentication. If Authentication is not enabled, the key is not available and encryption will not take place.

To use Encryption, Authentication must be enabled.

If this function is not called, the Network Access security settings in the Windows Registry will be used. These settings are used by BTExplorer and controlled from the Advanced Configuration -> Client applications tab in the BTTray application.

Prototype:

```
void SetSecurity (
                  BOOL authentication,
                  BOOL encryption);
```

Parameters:	authentication	TRUE means use authentication procedures on future operations using this object.
	encryption	TRUE means use encryption procedures on future data transfers that use this object.
Returns:	void	

GetExtendedError ( )

This function returns the object's last-error code. The last-error code is maintained on a per-object basis. Multiple objects do not overwrite each other's last-error code.

This function is intended for obtaining extended error information when the SDK return/result codes are not sufficient. This is useful when the UNKNOWN_RETURN_ERROR code is returned from an SDK API function, or the UNKNOWN_RESULT_ERROR code is returned in an SDK callback function.

To retrieve an extended error code of a method of this class in the case where the method has failed, call *GetExtendedError ( )* directly after this method has been called. *GetExtendedError ( )* can be called in callback functions of this class as well.

Prototype:	`WBtRc GetExtendedError ( );`
Parameters:	None
Returns:	See definition of the WBtRc enumeration in com_error.h

SetExtendedError ( )

This function sets the last-error code for the object of this class. *SetExtendedError ( )* is intended to localize the place where exactly an error occurs in the code. It can be called to reset the last-error code for an object of this class to WBT_SUCCESS (see com_error.h) before any other class method is called. After a class method has been called, *GetExtendedError ( )* can be invoked to read the error information. If *GetExtendedError ( )* reports WBT_SUCCESS, then the error occurs somewhere else in the program.

The last-error code is kept in a local variable of the object, so that multiple objects of this class do not overwrite each other's values.

Prototype:	`void SetExtendedError (WBtRc code);`	
Parameters:	code	One of the values from the WBtRc enumeration defined in com_error.h
Returns:	void	

Pure virtual OnStateChange ( )

This function must be provided by the application. It allows the application to detect when a connection is established or cleared to the LAP server.

Prototype:

```
virtual void OnStateChange (
                                BD_ADDR          bda,
                                DEV_CLASS        dev_class,
                                BD_NAME          name,
                                short            com_port,
                                LAP_STATE_CODE   state) = 0;
```

Parameters:

bda	The address of the LAP server.
dev_class	The class of the LAP server, see the DEV_CLASS encoding definitions in BtIfDefinitions.h.
name	Null terminated string of type 'unsigned char' of maximum length BD_NAME_LEN (= 248). See BtIfDefiniitions.h. This is the *friendly* name of the Bluetooth device.
com_port	The Local COM port used for the connection.
state	The new state: LAP_CONNECTED if connected, *or* LAP_DISCONNECTED if not connected.

Returns: void

CDUNCLIENT

This class allows the application to establish IP access via a Dial-Up Networking link.

Before a connection can be established, the client application must obtain a Bluetooth LAN server device address using the CBtIf class for inquiry and service discovery.

This class then provides the create connection, remove connection, and state change event functions.

This class is a base class. A pure virtual method, *OnStateChange ( )*, is defined to process connection state changes. The application must provide a derived class that defines the state change method for the application.

DUN_RETURN_CODE

A common set of return codes is provided by the DUN function calls. Use enumerated type DUN_RETURN_CODE from BtIfClasses.h

Table 22: DUN_RETURN_CODE

DUN_RETURN_CODE Value	*Meaning*
SUCCESS	Operation initiated without error.
NO_BT_SERVER	COM server could not be started.
ALREADY_CONNECTED	Attempt to connect before the previous connection closed.
NOT_CONNECTED	Attempt to close an unopened connection.
NOT_ENOUGH_MEMORY	Local processor could not allocate memory for open.
INVALID_PARAMETER	One or more of function parameters are not valid.
UNKNOWN_ERROR	Any condition other than the above.
LICENSE_ERROR	License error.

CreateConnection ( )

This function requests a DUN connection to the server. Before calling this function, the application must use a CBtIf class to locate a LAN using DialUp Network service.

Prototype:

```
DUN_RETURN_CODE CreateConnection (
                                  BD_ADDR             bda,
                                  CSdpDiscoveryRec    &sdp_rec);
```

Parameters:

bda	The server's Bluetooth device address. If this parameter is NULL, error INVALID_PARAMETER is returned.
sdp_rec	The service discovery record obtained for DUN server. Not used.

Returns: SUCCESS – Everything is OK. Otherwise, see Table 22: "DUN_RETURN_CODE," on page 107.

RemoveConnection ( )

This function closes the connection.

Prototype:	`DUN_RETURN_CODE RemoveConnection ( );`
Parameters:	None
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 22: “DUN_RETURN_CODE,” on page 107.

GetConnectionStats ( )

This function retrieves current connection statistics, using the common statistics structure tBT_CONN_STATS defined in BtIfDefinitions.h. See “GetConnectionStats ( )” on page 34.

This class uses all the statistic fields in the structure.

Prototype:	`DUN_RETURN_CODE GetConnectionStats (tBT_CONN_STATS *p_conn_stats);`	
Parameters:	p_conn_stats	A pointer to the user’s connection statistics structure; see above.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 22: “DUN_RETURN_CODE,” on page 107.	

SetSecurity ( )

This function sets the authentication and encryption parameters for DUN via DialUp Network operations.

Encrypting translates data into an unreadable format using a secret key or password. Decrypting the data requires the same key or password that was used to encrypt it.

Encryption in Bluetooth for Windows is based on the same passkey or Link Key that is used for Authentication. If Authentication is not enabled, the key is not available and encryption will not take place.

To use Encryption, Authentication must be enabled.

If this function is not called the Dial-up Networking security settings in the Windows Registry will be used. These settings are used by BTExplorer and controlled from the Advanced Configuration -> Client Applications tab in the BTTray application.

Prototype:

```
void SetSecurity (
                    BOOL authentication,
                    BOOL encryption);
```

Parameters:	authentication	TRUE means use authentication procedures on future operations using this object.
	encryption	TRUE means use encryption procedures on future data transfers that use this object.
Returns:	void	

GetExtendedError ( )

This function returns the object's last-error code. The last-error code is maintained on a per-object basis. Multiple objects do not overwrite each other's last-error code.

This function is intended for obtaining extended error information when the SDK return/result codes are not sufficient. This is useful when the UNKNOWN_RETURN_ERROR code is returned from an SDK API function, or the UNKNOWN_RESULT_ERROR code is returned in an SDK callback function.

To retrieve an extended error code of a method of this class in the case where the method has failed, call *GetExtendedError ( )* directly after this method has been called. *GetExtendedError ( )* can be called in callback functions of this class as well.

Prototype:	`WBtRc GetExtendedError ( );`
Parameters:	None.
Returns:	See definition of the WBtRc enumeration in com_error.h

SetExtendedError ( )

This function sets the last-error code for the object of this class. *SetExtendedError ( )* is intended to localize the place where exactly an error occurs in the code. It can be called to reset the last-error code for an object of this class to WBT_SUCCESS (see com_error.h) before any other class method is called. After a class method has been called, *GetExtendedError ( )* can be invoked to read the error information. If *GetExtendedError ( )* reports WBT_SUCCESS, then the error occurs somewhere else in the program.

The last-error code is kept in a local variable of the object so that multiple objects of this class do not overwrite each other's values.

Prototype:	`void SetExtendedError (WBtRc code);`	
Parameters:	code	One of the values from the WBtRc enumeration defined in com_error.h
Returns:	void	

Pure virtual OnStateChange ( )

This function must be provided. It allows the application to detect when a connection is established or cleared to the DUN server.

Prototype:

```
virtual void OnStateChange (
                          BD_ADDR          bda,
                          DEV_CLASS        dev_class,
                          BD_NAME          name,
                          short            com_port,
                          DUN_STATE_CODE   state) = 0;
```

Parameters:	bda	The address of the DUN server.
	dev_class	The class of the DUN server, see the DEV_CLASS encoding definitions in BtIfDefinitions.h.
	name	Null terminated string of type 'unsigned char' of maximum length BD_NAME_LEN (= 248). See BtIfDefiniitions.h. This is the *friendly* name of the Bluetooth device.
	com_port	The Local COM port used for the connection.
	state	The new state: DUN _CONNECTED if connected, *or* DUN _DISCONNECTED if not connected.
Returns:	void	

CSPPCLIENT

This class allows a client-side application to establish an SPP connection on a Windows COM port.

A pure virtual method, *OnClientStateChange ( ),* is defined to process connection state changes. The application must provide a derived class that defines the state change method for the application.

Before a connection can be attempted, the client application must obtain the Bluetooth device address of an SPP server using the CBtIf class for inquiry and service discovery.

The application then invokes the *CreateConnection ( )* method. When a successful connection is established, *OnClientStateChange ( )* is called with the associated COM port as a parameter*.*

The application uses the COM port number to construct a Windows file name for the communication resource, e.g., "COM5:", and calls the Windows *CreateFile ( )* function. The application can then use the standard Windows serial I/O functions, *ReadFile ( )*, *WriteFile ( )*, etc. to transfer data over the Bluetooth SPP connection.

For more information on the *Serial Port Profile,* see the Bluetooth Special Interest Group, Serial Port Profile.

Configuration Notes

In the current release of Bluetooth for Windows (BTW), a COM port is predefined for SPP applications named "Bluetooth Serial Port". SDK client applications will use this COM port by default when calling *CreateConnection ( )*. Future SDK releases will support a programmatic method to create new COM ports associated with SPP Client Applications. For this SDK release, if additional COM port client connections are required, they can be defined manually using the Bluetooth Neighborhood configuration function in the *Applications* tab. For an alternative COM port client access method, see *CBtIf::CreateCOMPortAssociation ( )*.

SPP_CLIENT_RETURN_CODE

A common set of return codes is provided by the SPP function calls. Use enumerated type SPP_CLIENT_RETURN_CODE, from BtIfClasses.h.

Table 23: SPP_CLIENT_RETURN_CODE

SPP_CLIENT_RETURN_CODE Value	*Meaning*
SUCCESS	Operation initiated without error.
NO_BT_SERVER	COM server could not be started.
ALREADY_CONNECTED	Attempt to connect before the previous connection closed.
NOT_CONNECTED	Attempt to close an unopened connection.
NOT_ENOUGH_MEMORY	Local processor could not allocate memory for open.
INVALID_PARAMETER	One or more of function parameters are not valid.
UNKNOWN_ERROR	Any condition other than the above.
NO_EMPTY_PORT	There is no empty COM port available.
LICENSE_ERROR	License error.

CreateConnection ( )

This function requests an SPP connection to the server. Before calling this function, the application uses SDK class CBtIf to locate an SPP server that offers the required service.

After calling this function, the application will receive an event callback that contains the status of the connection attempt via the *OnClientStateChange ( )* function.

Note: If an application uses multiple CSppClient objects, it must serialize calls to *CreateConnection ( )* from the separate CSppClient objects if they are destined for the same Bluetooth device address, such that successive calls are not made before the preceding *OnClientStateChange ( )* event callbacks are received. This restriction only applies to calls to the same remote Bluetooth address, all other calls can be performed without serialization.

Prototype:

```
SPP_CLIENT_RETURN_CODE CreateConnection (
                                        BD_ADDR        bda,
                                        BT_CHAR        *szServiceName);
```

Parameters:	bda	The server's Bluetooth device address. If this parameter is NULL, error INVALID_PARAMETER is returned.
	szServiceName	The service name from the discovery record obtained for the SPP server.

Returns: SUCCESS – Everything is OK. Otherwise, see Table 23: "SPP_CLIENT_RETURN_CODE," on page 111.

RemoveConnection ( )

This function closes the connection, dissociating the COM port from the client application. This operation is required to free the port for other applications.

Prototype: `SPP_CLIENT_RETURN_CODE RemoveConnection ( );`

Parameters: None

Returns: SUCCESS – Everything is OK.
Otherwise, see Table 23: "SPP_CLIENT_RETURN_CODE," on page 111.

GetConnectionStats ( )

This function retrieves current connection statistics, using the common statistics structure tBT_CONN_STATS defined in BtIfDefinitions.h. See "GetConnectionStats ( )" on page 34.

This class uses all the statistic fields in the structure.

Prototype: `SPP_CLIENT_RETURN_CODE GetConnectionStats (tBT_CONN_STATS *p_conn_stats);`

Parameters:	p_conn_stats	A pointer to the user's connection statistics structure; see above.

Returns: SUCCESS – Everything is OK.
Otherwise, see Table 23: "SPP_CLIENT_RETURN_CODE," on page 111.

GetExtendedError ( )

This function returns the object's last-error code. The last-error code is maintained on a per-object basis. Multiple objects do not overwrite each other's last-error code.

This function is intended for obtaining extended error information when the SDK return/result codes are not sufficient. This is useful when the UNKNOWN_RETURN_ERROR code is returned from an SDK API function, or the UNKNOWN_RESULT_ERROR code is returned in an SDK callback function.

To retrieve an extended error code of a method of this class in the case where the method has failed, call *GetExtendedError ( )* directly after this method has been called. *GetExtendedError ( )* can be called in callback functions of this class as well.

Prototype:	`WBtRc GetExtendedError ( );`
Parameters:	None
Returns:	See definition of the WBtRc enumeration in com_error.h

SetExtendedError ( )

This function sets the last-error code for the object of this class. *SetExtendedError ( )* is intended to localize the place where exactly an error occurs in the code. It can be called to reset the last-error code for an object of this class to WBT_SUCCESS (see com_error.h) before any other class method is called. After a class method has been called, *GetExtendedError ( )* can be invoked to read the error information. If *GetExtendedError ( )* reports WBT_SUCCESS, then the error occurs somewhere else in the program.

The last-error code is kept in a local variable of the object, so that multiple objects of this class do not overwrite each other's values.

Prototype:	`void SetExtendedError (WBtRc code);`	
Parameters:	code	One of the values from the WBtRc enumeration defined in com_error.h.
Returns:	void	

Pure virtual OnClientStateChange ( )

This function must be provided. It informs the application when a connection to the SPP server has been established, or when the server has initiated a disconnect.

Note: When the client initiates a disconnect, this method is not called as a result.

When the connection is established, the COM port value should be used by the client application to open the Windows COM port.

Prototype:

```
virtual void OnClientStateChange (
                                  BD_ADDR          bda,
                                  DEV_CLASS        dev_class,
                                  BD_NAME          name,
                                  short            com_port,
                                  SPP_STATE_CODE   state) = 0;
```

Parameters:

bda	The address of the SPP server.
dev_class	The class of the SPP server, see DEV_CLASS encoding definitions in BtIfDefinitions.h.
name	The name of the connecting server device.
com_port	The local COM port assigned to the service.
state	The new state: • SPP_CONNECTED • SPP_DISCONNECTED • SPP_RFCOMM_CONNECTION_FAILED • SPP_PORT_IN_USE • SPP_PORT_NOT_CONFIGURED • SPP_SERVICE_NOT_FOUND In case of the state is not SPP_CONNECTED or SPP_DISCONNECTED, this callback will get called twice, first is with reason state defined above, and SPP_DISCONNECTED after that.

Returns: void

CreateCOMPort ( )

This function is not supported in the current SDK release.

CSPPSERVER

This class allows a server-side application to establish an SPP connection on a Windows COM port.

This class defines a pure virtual method to process connection state changes, *OnServerStateChange ( )*. The application must provide a derived class that defines the state change method for the application.

The application then invokes the *CreateConnection ( )* method to *listen* for an incoming connection request from a client. When that happens, the state change method is called with a successful connection from the client, and the application receives the local COM port associated with the requested service.

The application then uses the port number to construct a Windows file name for the communication resource, e.g., "COM3:", and calls the Windows *CreateFile ( )* function. Then the application can use the standard Windows serial I/O functions, *ReadFile ( )*, *WriteFile ( )*, etc. to transfer data over the Bluetooth SPP connection.

Configuration Notes

BTW is installed with a predefined serial service, "Bluetooth Serial Port". It is possible for a new application to use this service for SPP. This is not recommended because there may be other uses for the predefined service.

The preferred way to use CSppServer is with a new serial service, which can be created manually using the BT Configuration function, *Local Services* tab, press the *Add Serial Service* button. Then fill in the fields in the *Service Properties* dialog – Service name, COM port, and the security settings.

Future SDK releases will provide a CSppServer method to allow the application to assign a new COM port service programmatically.

SPP_SERVER_RETURN_CODE

A common set of return codes is provided by the SPP function calls. Use enumerated type SPP_SERVER_RETURN_CODE, from BtIfClasses.h.

Table 24: SPP_SERVER_RETURN_CODE

SPP_SERVER_RETURN_CODE Value	*Meaning*
SUCCESS	Operation initiated without error.
NO_BT_SERVER	COM server could not be started.
ALREADY_CONNECTED	Attempt to connect before the previous connection closed.
NOT_CONNECTED	Attempt to close an unopened connection.
NOT_ENOUGH_MEMORY	Local processor could not allocate memory for open.
NOT_SUPPORTED	The service name was not defined in the *Local Services* list.
UNKNOWN_ERROR	Any condition other than the above.
NO_EMPTY_PORT	There is no empty COM port available.
INVALID_PARAMETER	One or more function parameters are not valid.
LICENSE_ERROR	License error.

CreateConnection ( )

This function checks windows registry for the named service. If the service exists, a connection is initiated for the service. This results in the local service *listening* for an incoming client request for connection.

OnServerStateChange ( ) is called when an SPP client establishes a connection.

Prototype:	`SPP_SERVER_RETURN_CODE CreateConnection (BT_CHAR *szServiceName);`	
Parameters:	szServiceName	The service name to be used for this application.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 24: “SPP_SERVER_RETURN_CODE,” on page 115.	

RemoveConnection ( )

This function closes the connection, dissociating the COM port from the server application. This operation is required to free the port for other applications.

Prototype:	`SPP_SERVER_RETURN_CODE RemoveConnection ( );`
Parameters:	None
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 24: “SPP_SERVER_RETURN_CODE,” on page 115.

GetConnectionStats ( )

This function retrieves current connection statistics, using the common statistics structure tBT_CONN_STATS defined in BtIfDefinitions.h. See “GetConnectionStats ( )” on page 34.

This class uses all the statistic fields in the structure.

Prototype:	`SPP_SERVER_RETURN_CODE GetConnectionStats (tBT_CONN_STATS *p_conn_stats);`	
Parameters:	p_conn_stats	A pointer to the user’s connection statistics structure; see above.
Returns:	SUCCESS – Everything is OK. Otherwise, see Table 24: “SPP_SERVER_RETURN_CODE,” on page 115.	

GetExtendedError ( )

This function returns the object's last-error code. The last-error code is maintained on a per-object basis. Multiple objects do not overwrite each other's last-error code.

This function is intended for obtaining extended error information when the SDK return/result codes are not sufficient. This is useful when the UNKNOWN_RETURN_ERROR code is returned from an SDK API function or the UNKNOWN_RESULT_ERROR code is returned in an SDK callback function.

To retrieve an extended error code of a method of this class in the case where the method has failed, call *GetExtendedError ( )* directly after this method has been called. *GetExtendedError ( )* can be called in callback functions of this class as well.

Prototype:	`WBtRc GetExtendedError ( );`
Parameters:	None
Returns:	See definition of the WBtRc enumeration in com_error.h

SetExtendedError ( )

This function sets the last-error code for the object of this class. *SetExtendedError ( )* is intended to localize the place where exactly an error occurs in the code. It can be called to reset the last-error code for an object of this class to WBT_SUCCESS (see com_error.h) before any other class method is called. After a class method has been called, *GetExtendedError ( )* can be invoked to read the error information. If *GetExtendedError ( )* reports WBT_SUCCESS, then the error occurs somewhere else in the program.

The last-error code is kept in a local variable of the object so that multiple objects of this class do not overwrite each other's values.

Prototype:	`void SetExtendedError (WBtRc code);`	
Parameters:	code	One of the values from the WBtRc enumeration defined in com_error.h
Returns:	void	

Pure virtual OnServerStateChange ( )

This function must be provided. It informs the application when a connection to a SPP client has been established, or when the client has initiated a disconnect.

Note: When the server initiates a disconnect, this method is not called as a result.

When the connection event is received, the COM port value should be used by the server application to open the Windows COM port.

Prototype:

```
virtual void OnServerStateChange (
                                    BD_ADDR          bda,
                                    DEV_CLASS        dev_class,
                                    BD_NAME          name,
                                    short            com_port,
                                    SPP_STATE_CODE   state) = 0;
```

Parameters:

bda	The Bluetooth device address of the SPP server.
dev_class	The class of the SPP server, see the DEV_CLASS encoding definitions in BtIfDefinitions.h.
com_port	The local COM port assigned to the service.
state	The new state: • SPP_CONNECTED • SPP_DISCONNECTED • SPP_RFCOMM_CONNECTION_FAILED • SPP_ALLOC_SCN_FAILED • SPP_SDP_FULL In case of the state is not SPP_CONNECTED or SPP_DISCONNECTED, this callback will get called twice, first is with reason state defined above, and SPP_DISCONNECTED after that.

Returns: void

CreateCOMPort ( )

This function is not supported in the current SDK release.

COBEXHEADERS

This class is a container for OBEX message headers. Methods are provided to add and get OBEX headers, such as *Name*, *Type*, *Target*, etc.

The more complex header fields are supported by separate classes CObexAppParam, CObexAuth, and CObexUserDefined.

The remaining header fields are directly supported by methods of the CObexHeaders class.

For more information on the *Object Exchange Profile,* see the Infrared Data Association, IrDA Object Exchange Protocol (IrOBEX) with Published Errata, Version 1.2, April 1999.

SetCount ( )

This function sets the count header value. This is the number of objects to be sent.

Prototype:	`void SetCount (UINT32 count);`	
Parameters:	count	Number of objects to be sent.
Returns:	void	

DeleteCount ( )

This function deletes the header.

Prototype:	`void DeleteCount ( );`
Parameters:	None.
Returns:	void

GetCount ( )

This function returns the header value.

Prototype:	`BOOL GetCount (UINT32 *p_count);`	
Parameters:	p_count	Points to header value.
Returns:	TRUE if the header is present and value provided; FALSE, otherwise.	

SetName ( )

This function sets the name header value. This is the name of the OBEX object.

The array is copied into memory managed by the CObexHeaders object.

A NULL array pointer on input is used to construct an *empty* header. Internally, this is a header that has 0 length and a NULL array pointer. The CObexHeaders object is flagged as having this header; the header just happens to be empty. The input *length* parameter is ignored for this case and the internal length is set to 0.

Prototype:	`BOOL SetName (WCHAR *p_array);`	
Parameters:	p_array	Pointer to a null terminated string
Returns:	TRUE if successful; FALSE if sufficient memory not available to copy the header.	

DeleteName ( )

This function deletes the header.

Prototype:	`void DeleteName ( );`
Parameters:	None
Returns:	void

GetNameLength ( )

This function tests if the header is present. If so, the header length is provided. This value includes the null terminator.

Prototype:	`BOOL GetNameLength (UINT32 *p_len_incl_null);`	
Parameters:	p_len_incl_null	When the header is present, this parameter provides the number of characters allowed for the header value, including the null terminator character.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetName ( )

This function returns the header value to the caller's buffer. Caller must first call GetNameLength ( ) to determine if the header is present and to obtain the length.

Prototype:	`BOOL GetName (WCHAR *p_array);`	
Parameters:	p_array	Pointer to a character string buffer, which will receive the header value, including the null terminator character.
Returns:	TRUE if the header is present, and the header value is copied to the caller's buffer. Otherwise, FALSE. The caller must allocate and release this buffer.	

SetType ( )

This function sets the type header value.

The array is copied into memory managed by the CObexHeaders object.

A NULL array pointer on input is used to construct an *empty* header. Internally, this is a header that has 0 length and a NULL array pointer. The CObexHeaders object is flagged as having this header; the header just happens to be empty. The input *length* parameter is ignored for this case; and the internal length is set to 0.

Prototype:

```
BOOL SetType (
             UINT8     *p_array,
             UINT32    length);
```

Parameters:	p_array	Pointer to an unstructured octet array
	length	Number of characters
Returns:	TRUE if successful; FALSE if sufficient memory not available to copy the header.	

DeleteType ( )

This function deletes the header.

Prototype:	`void DeleteType ( );`
Parameters:	None
Returns:	void

GetTypeLength ( )

This function tests if the header is present. If so the header length is provided.

Prototype:	`BOOL GetTypeLength (UINT32 *p_length);`	
Parameters:	p_length	When the header is present, this parameter provides thenumber of characters allowed for the header value.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetType ( )

This function returns the header value to the caller's buffer. Caller must first call *GetTypeLength ( )* to determine if the header is present and to obtain the length.

Prototype:	`BOOL GetType (UINT8 *p_array);`	
Parameters:	p_array	Pointer to an unstructured octet array, which will receive the header value.
Returns:	TRUE if the header is present, and the header value is copied to the caller's buffer. Otherwise, FALSE. The caller must allocate and release this buffer.	

SetLength ( )

This function sets the header value. This is the length of the OBEX object.

Prototype:	`void SetLength (UINT32 length);`	
Parameters:	length	Length in bytes of the OBEX object
Returns:	None	

DeleteLength ( )

This function deletes the header.

Prototype:	`void DeleteLength ( );`
Parameters:	None
Returns:	void

GetLength ( )

This function returns the header value.

Prototype:	`BOOL GetLength (UINT32 *p_length);`	
Parameters:	p_length	Points to header value.
Returns:	TRUE if the header is present and value provided; FALSE, otherwise.	

SetTime ( )

This function sets the OBEX header parameter — time.

Prototype:	`BOOL SetTime (char *p_str_8601);`	
Parameters;	p_str_8601	Pointer to the caller's time buffer, as a NULL terminated ASCII string in ISO 8601 format. yyyymmddThhmmss for local time or yyyymmddThhmmssZ for UTC time
Returns:	TRUE if the time parameter was valid and the header was set.	

DeleteTime ( )

This function deletes the header.

Prototype:	`void DeleteTime ( );`
Parameters:	None
Returns:	void

GetTime ( )

This function returns the OBEX header parameter – length of object.

Prototype:	`BOOL GetTime (char *p_str_8601);`	
Parameters:	p_str_8601	Pointer to the caller's time buffer. The output will be formatted as an ASCII string in ISO 8601 format. yyyymmddThhmmss for local time or yyyymmddThhmmssZ for UTC time. The caller must allow 17 char buffer, which includes a null terminator.
Returns:	TRUE if the header exists and is valid and the caller's buffer has been filled; FALSE, otherwise.	

SetDescription ( )

This function sets the header value. This is an informal description of the OBEX object.

The array is copied into memory managed by the CObexHeaders object.

A NULL array pointer on input (or a string of length 0) is used to construct an *empty* header. Internally, this is a header that has 0 length and a NULL array pointer. The CObexHeaders object is flagged as having this header; the header just happens to be empty. The internal length is set to 0.

Prototype:	`BOOL SetDescription (WCHAR *p_array);`	
Parameters:	p_array	Pointer to a null terminated character string
Returns:	TRUE if successful; FALSE if sufficient memory not available to copy the header.	

DeleteDescription ( )

This function deletes the header.

Prototype:	`void DeleteDescription ( );`
Parameters:	None
Returns:	void

GetDescriptionLength ( )

This function tests if the header is present. If so, the header length is provided. This value includes the null terminator.

Prototype:	`BOOL GetDescriptionLength (UINT32 *p_len_inc_null);`	
Parameters:	p_len_incl_null	When the header is present, this parameter provides the number of characters allowed for the header value, including the null terminator character.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetDescription ( )

This function returns the header value to the caller's buffer. Caller must first call *GetDescriptionLength ( )* to determine if the header is present and to obtain the length.

Prototype:	`BOOL GetDescription (WCHAR *p_array);`	
Parameters:	p_array	Pointer to a character string buffer, which will receive the header value, including the null terminator character.
Returns:	TRUE if the header is present, and the header value is copied to the caller's buffer. Otherwise, FALSE. The caller must allocate and release this buffer.	

AddTarget ( )

This function adds a target to the OBEX header. The unstructured octet array is copied into memory managed by the CObexHeaders object.

This is the name of a service the object is being sent to. OBEX provides a series of well-known target header values. A new target can be defined as a UUID.

Targets are added up to a maximum defined in OBEX_MAX_TARGET, with value 3.

A NULL array pointer on input is used to construct an *empty* header. Internally, this is a header that has 0 length and a NULL array pointer. The CObexHeaders object is flagged as having this header; the header just happens to be empty. The input *length* parameter is ignored for this case; and the internal length is set to 0.

Prototype:

```
BOOL AddTarget (
                 UINT8     *p_array,
                 UINT32    length);
```

Parameters:	p_array	Pointer to unstructured octet array
	length	Number of characters
Returns:	TRUE if the target was successfully added; FALSE if the maximum count had already been reached, or if memory could not be allocated to copy the array.	

GetTargetCnt ( )

This function returns the count of target headers present. The maximum value supported is defined in OBEX_MAX_TARGET, with value 3.

Prototype:

```
UINT32 GetTargetCnt ( );
```

Parameters: None

Returns: Number of targets currently present in CObexHeaders object.

DeleteTarget ( )

This function deletes the indicated target header from the array of target headers. After the deletion, the count is corrected and array is compacted.

Prototype:

```
BOOL DeleteTarget (UINT16 index);
```

Parameters:	index	Selects element from array; index 0 refers to the first element.
Returns:	TRUE if the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.	

GetTargetLength ( )

This function tests if the header is present. If so the header length is provided.

Prototype:

```
BOOL GetTargetLength (
                        UINT32    *p_length,
                        UINT16    index);
```

Parameters:	p_length	When the header is present, this parameter provides the number of characters to allow for the header value.
	index	Selects element from array; index 0 refers to the first element.

Returns: TRUE if the header is present, and the header length is provided. Otherwise, FALSE.

GetTarget ( )

This function returns the selected object target to the caller's buffer. Caller must first call *GetTargetLength ( )* to determine if the header is present and to obtain the length.

Prototype:

```
BOOL GetTarget (
                    UINT8     *p_array,
                    UINT16    index);
```

Parameters:	p_array	Pointer to an unstructured octet array, which will receive the header value.
	index	Selects element from array; index 0 refers to the first element.

Returns: TRUE if the header is present, and the header value is copied to the caller's buffer. Otherwise, FALSE. The caller must allocate and release this buffer.

AddHttp ( )

This function adds a HTTP entry to the OBEX header. The unstructured octet array is copied into memory managed by the CObexHeaders object.

HTTPs are added up to a maximum defined in OBEX_MAX_HTTP, currently with value 3.

A NULL array pointer on input is used to construct an *empty* header. Internally, this is a header that has 0 length and a NULL array pointer. The CObexHeaders object is flagged as having this header; the header just happens to be empty. The input *length* parameter is ignored for this case; and the internal length is set to 0.

Prototype:

```
BOOL AddHttp (
                char      *p_array,
                UINT32    length);
```

Parameters:	p_array	Pointer to an unstructured octet array.
	length	Number of characters.

Returns: TRUE if the target was successfully added; FALSE if the maximum count had already been reached, or if memory could not be allocated to copy the array.

GetHttpCnt ( )

This function returns the OBEX header parameter – HTTP count. The maximum value supported is defined in OBEX_MAX_HTTP, with value 3.

Prototype:	`UINT32 GetHttpCnt ( );`
Parameters:	None
Returns:	Number of HTTP headers currently defined in CObexHeaders object.

DeleteHttp ( )

This function deletes the indicated HTTP header from the array of HTTP headers. After the deletion, the count is corrected and array is compacted.

Prototype:	`BOOL DeleteHttp (UINT16 index);`	
Parameters:	index	Selects element from array, index 0 refers to the first element
Returns:	TRUE if the indexed header was present and has been deleted. FALSE if the index was invalid, or the indexed header was not present.	

GetHttpLength ( )

This function tests if the header is present. If so the header length is provided.

Prototype:

```
BOOL GetHttpLength (
                 UINT32    *p_length,
                 UINT16    index);
```

Parameters:	p_length	When the header is present, this parameter provides the number of characters to allow for the header value.
	index	Selects element from array, index 0 refers to the first element.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetHttp ( )

This function returns a copy of the selected HTTP to the caller's buffer.

Prototype:

```
BOOL GetHttp (
             char      *p_array,
             UINT16    index);
```

Parameters:	p_array	Pointer to an unstructured octet array, which will receive the header value.
	index	Selects element from array; index 0 refers to the first element.
Returns:	TRUE if the header is present, and the header value is copied to the caller's buffer. Otherwise, FALSE. The caller must allocate and release this buffer.	

SetBody ( )

This function sets the body header value – this is the content of the OBEX object.

The array is copied into memory managed by the CObexHeaders object.

A NULL array pointer on input is used to construct an *empty* header. Internally, this is a header that has 0 length and a NULL array pointer. The CObexHeaders object is flagged as having this header; the header just happens to be empty. The input *length* parameter is ignored for this case and the internal length is set to 0.

If the caller's intention is to use this function for a *PUT Create Empty* OBEX call, then the body_end flag must also be set TRUE.

Prototype:

```
BOOL SetBody (
                UINT8     *p_array,
                UINT32    length,
                BOOL      body_end);
```

Parameters:	p_array	Pointer to an unstructured octet array.
	length	Number of characters.
	body_end	TRUE if this is the only headers object used to contain an OBEX object, or if this is the last of a series of headers objects which together contain an OBEX object.
Returns:	TRUE if successful; FALSE if sufficient memory not available to copy the header.	

DeleteBody ( )

This function deletes the header.

Prototype:	`void DeleteBody ( );`
Parameters:	None
Returns:	void

GetBodyLength ( )

This function tests if the header is present. If so the header length is provided.

Prototype:	`BOOL GetBodyLength (UINT32 *p_length);`	
Parameters:	p_length	When the header is present, this parameter provides the number of characters to allow for the header value.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetBody ( )

This function returns the header value to the caller's buffer. Caller must first call *GetBodyLength ( )* to determine if the header is present and to obtain the length.

Prototype:

```
BOOL GetBody (
                UINT8       *p_array,
                BOOL        *p_body_end);
```

Parameters:	p_array	Pointer to an unstructured octet array, which will receive the header value.
	p_body_end	Boolean set TRUE if this is the only headers object for an OBEX object, or if this headers object is the last of a series of headers objects which contain an OBEX object.

Returns: TRUE if the header is present, and the header value is copied to the caller's buffer. Otherwise, FALSE. The caller must allocate and release this buffer.

SetWho ( )

This function sets the object who in the OBEX header. The array is copied into memory managed by the CObexHeaders object.

The who is the peer OBEX application the object is being sent to. Typically, the who is a 128-bit UUID (represented here as an unstructured octet array) of the service which has accepted an OBEX connection.

A NULL array pointer on input is used to construct an *empty* header. Internally, this is a header that has 0 length and a NULL array pointer. The CObexHeaders object is flagged as having this header; the header just happens to be empty. The input *length* parameter is ignored for this case; and the internal length is set to 0.

Prototype:

```
BOOL SetWho (
                UINT8         *p_array,
                UINT32        length);
```

Parameters:	p_array	Pointer to an unstructured octet array.
	length	Number of characters.

Returns: TRUE if successful; FALSE if sufficient memory is not available to copy the header.

DeleteWho ( )

This function deletes the header.

Prototype: `void DeleteWho ( );`

Parameters: None

Returns: void

GetWhoLength ( )

This function tests if the header is present. If so, the header length is provided.

Prototype:	`BOOL GetWhoLength (UINT32 *p_length);`	
Parameters:	p_length	When the header is present, this parameter provides the number of characters to allow for the header value.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetWho ( )

This function returns the header value to the caller's buffer. Caller must first call *GetWhoLength ( )* to determine if the header is present and to obtain the length.

Prototype:	`BOOL GetWho (UINT8 *p_array);`	
Parameters:	p_array	Pointer to an unstructured octet array, which will receive the header value.
Returns:	TRUE if the header is present, and the header value is copied to the caller's buffer. Otherwise, FALSE. The caller must allocate and release this buffer.	

AddAppParam ( )

This function adds an application parameter entry to the OBEX header. The contents of the input parameters are copied into memory managed by the CObexHeaders object.

Application parameter entries are added up to a maximum defined in OBEX_MAX_APP_PARAM, currently with value 3.

This header is an example of a *tagged* header, which consists of a tag value and an octet array. The array can be empty (zero length and NULL pointer). But, unlike other simple octet array headers (such as Target), there is no such thing as an empty header of this type.

Prototype:

```
BOOL AddAppParam (
                        UINT8     tag,
                        UINT8     length,
                        UINT8     *p_array);
```

Parameters:	tag	Application parameter tag
	length	Length of the octet array
	p_array	points to an octet array
Returns:	TRUE if the entry was successfully added; FALSE if the maximum count had already been reached, or if memory could not be allocated to copy the array.	

GetAppParamCnt ( )

This function returns the application parameter count. The maximum value supported is defined in OBEX_MAX_APP_PARAM, with value 3.

Prototype:	`UINT32 GetAppParamCnt ( );`
Parameters:	None
Returns:	Number of application parameters currently defined .

DeleteAppParam ( )

This function deletes the indicated HTTP header from the array of HTTP headers. After the deletion the count is corrected and array is compacted.

Prototype:	`BOOL DeleteAppParam (UINT16 index);`	
Parameters:	index	Selects element from array; index 0 refers to the first element.
Returns:	TRUE if the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.	

GetAppParamLength ( )

This function tests if the header is present. If so, the header length is provided.

Prototype:

```
BOOL GetAppParamLength (
                        UINT8      *p_length,
                        UINT16     index);
```

Parameters:	p_length	When the header is present, used to provide number of octets to allow for the header value.
	index	Selects element from array; index 0 refers to the first element.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetAppParam ( )

This function returns a copy of the selected application parameter.

The caller must first call *GetAppParamLength ( )* to determine if the header is present and to obtain the length of the octet array.

Prototype:

```
BOOL GetAppParam (
                  UINT8      *p_tag,
                  UINT8      *p_array,
                  UINT16     index);
```

Parameters:	p_tag	Caller's tag value.
	p_array	Points to the caller's octet array. NULL indicates that the array is not wanted by the caller.
	index	Selects application parameter from array, index 0 refers to the first element.
Returns:	TRUE if the indexed element exists. Otherwise, FALSE.	

AddAuthChallenge ( )

This function adds an authentication challenge entry to the OBEX header. The contents of the input parameters are copied into memory managed by the CObexHeaders object.

Authentication challenge entries are added up to a maximum defined in OBEX_MAX_AUTH_CHALLENGE, currently with value 3.

This header is an example of a *tagged* header, which consists of a tag value and an octet array. The array can be empty (zero length and NULL pointer). But, unlike other simple octet array headers (such as Target), there is no such thing as an empty header of this type.

Prototype:

```
BOOL AddAuthChallenge (
                               UINT8     tag,
                               UINT8     length,
                               UINT8     *p_array);
```

Parameters:

tag	Application parameter tag.
length	Length of the octet array.
p_array	Points to an octet array.

Returns: TRUE if the entry was successfully added; FALSE if the maximum count had already been reached, or if memory could not be allocated to copy the array.

GetAuthChallengeCnt ( )

This function returns the OBEX object authentication challenge count. The maximum value supported is defined in OBEX_MAX_ AUTH_CHALLENGE, with value 3.

Prototype: `UINT32 GetAuthChallengeCnt ( );`

Parameters: None

Returns: Number of authentication challenge parameters currently defined in CObexHeaders object.

DeleteAuthChallenge ( )

This function deletes the indicated authentication challenge from the array of authentication challenge headers. After the deletion the count is corrected and array is compacted.

Prototype: `BOOL DeleteAuthChallenge (UINT16 index);`

Parameters:

index	Selects element from array, index 0 refers to the first element.

Returns: TRUE if the indexed header was present and has been deleted. FALSE if the index was invalid, or the indexed header was not present.

GetAuthChallengeLength ( )

This function tests if the header is present. If so the header length is provided.

Prototype:

```
BOOL GetAuthChallengeLength (
                           UINT8      *p_length,
                           UINT16     index);
```

Parameters:	p_length	When the header is present, used to provide number of characters to allow for the header value.
	index	Selects element from array; index 0 refers to the first element.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetAuthChallenge ( )

This function returns a copy of the selected authentication challenge.

The caller must first call *GetAuthChallengeLength ( )* to determine if the header is present and to obtain the length of the octet array.

Prototype:

```
BOOL GetAuthChallenge (
                     UINT8      *p_tag,
                     UINT8      *p_array,
                     UINT16     index);
```

Parameters:	p_tag	caller's tag value.
	p_array	points to the caller's octet array. NULL indicates that the array is not wanted by the caller.
	index	selects application parameter from array, index 0 refers to the first element.
Returns:	TRUE if the indexed element exists. Otherwise, FALSE.	

AddAuthResponse ( )

This function adds an authentication response entry to the OBEX header. The contents of the input parameters are copied into memory managed by the CObexHeaders object.

Authentication response entries are added up to a maximum defined in OBEX_MAX_AUTH_ RESPONSE, currently with value 3.

This header is an example of a *tagged* header, which consists of a tag value and an octet array. The array can be empty (zero length and NULL pointer). But, unlike other simple octet array headers (such as Target), there is no such thing as an empty header of this type.

Prototype:

```
BOOL AddAuthResponse (
                        UINT8     tag,
                        UINT8     length,
                        UINT8     *p_array);
```

Parameters:

tag	Application parameter tag.
length	Length of the octet array.
p_array	Points to an octet array.

Returns: TRUE if the entry was successfully added; FALSE if the maximum count had already been reached, or if memory could not be allocated to copy the array.

GetAuthResponseCnt ( )

This function returns the OBEX object authentication response count. The maximum value supported is defined in OBEX_MAX_AUTH_RESPONSE, with value 3.

Prototype: `UINT32 GetAuthResponseCnt ( );`

Parameters: None

Returns: Number of authentication response parameters currently defined in CObexHeaders object.

DeleteAuthResponse ( )

This function deletes the indicated authentication response from the array of authentication response headers. After the deletion the count is corrected and array is compacted.

Prototype: `BOOL DeleteAuthResponse (UINT16 index);`

Parameters:

index	Selects element from array, index 0 refers to the first element.

Returns: TRUE if the indexed header was present and has been deleted. FALSE if the index was invalid, or the indexed header was not present.

GetAuthResponseLength ( )

This function tests if the header is present. If so the header length is provided.

Prototype:

```
BOOL GetAuthResponseLength (
                            UINT8     *p_length,
                            UINT16    index);
```

Parameters:	p_length	When the header is present, this parameter provides the number of characters to allow for the header value.
	index	Selects element from array; index 0 refers to the first element.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetAuthResponse ( )

This function returns a copy of the selected authentication response.

The caller must first call *GetAuthResponseLength ( )* to determine if the header is present and to obtain the length of the octet array.

Prototype:

```
BOOL GetAuthResponse (
                      UINT8     *p_tag,
                      UINT8     *p_array,
                      UINT16    index);
```

Parameters:	p_tag	Caller's tag value
	p_array	Points to the caller's octet array. NULL indicates that the array is not wanted by the caller.
	index	Selects application parameter from array; index 0 refers to the first element.
Returns:	TRUE if the indexed element exists; otherwise, FALSE.	

SetObjectClass ( )

This function sets the object class in the OBEX header. The array is copied into memory managed by the CObexHeaders object.

This is the OBEX object class, as an unstructured octet array.

Prototype:

```
BOOL SetObjectClass (
                     UINT8     *p_array,
                     UINT32    length);
```

Parameters:	p_array	Pointer to an unstructured octet array.
	length	Number of characters.
Returns:	TRUE if successful; FALSE if sufficient memory is not available to copy the header.	

DeleteObjectClass ( )

This function deletes the header.

Prototype:	`void DeleteObjectClass ( );`
Parameters:	None
Returns:	void

GetObjectClassLength ( )

This function tests if the header is present. If so the header length is provided.

Prototype:	`BOOL GetObjectClassLength (UINT32 *p_length);`	
Parameters:	p_length	When the header is present, this parameter provides the number of characters to allow for the header value.
Returns:	TRUE if the header is present, and the header length is provided. Otherwise, FALSE.	

GetObjectClass ( )

This function returns the header value to the caller's buffer. Caller must first call *GetObjectClassLength ( )* to determine if the header is present and to obtain the length.

Prototype:	`BOOL GetObjectClass (UINT8 *p_array);`	
Parameters:	p_array	Pointer to an unstructured octet array, which will receive the header value.
Returns:	TRUE if the header is present, and the header value is copied to the caller's buffer. Otherwise,FALSE. The caller must allocate and release this buffer.	

AddUserDefined ( )

This function adds a user defined entry, as an SDK class CObexUserDefined to the OBEX header. The contents of the CObexUserDefined object are copied into memory managed by the CObexHeaders object.

Authentication response entries are added up to a maximum defined in OBEX_MAX_USER_HDR, currently with value 4.

Prototype:	`BOOL AddUserDefined (CObexUserDefined *p_user_defined);`	
Parameters:	p_user_defined	Points to a user defined object
Returns:	TRUE if the entry was successfully added; FALSE if the maximum count had already been reached, or if memory could not be allocated to copy the array.	

GetUserDefinedCnt ( )

This function returns the OBEX object user defined count. The maximum value supported is defined in OBEX_MAX_USER_HDR, with value 4.

Prototype: `UINT32 GetUserDefinedCnt ( );`

Parameters: None

Returns: Number of user defined headers currently defined in CObexHeaders object

DeleteUserDefined ( )

This function deletes the indicated *user defined* entry from the array of *user defined* headers. After the deletion the count is corrected and array is compacted.

Prototype: `BOOL DeleteUserDefined (UINT16 index);`

Parameters: index — Selects element from array, index 0 refers to the first element

Returns: TRUE if the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.

GetUserDefinedLength ( )

This function tests if the header is present. If so the header length is provided.

Prototype:

```
BOOL GetUserDefinedLength (
                              UINT16     *p_length,
                              UINT16     index);
```

Parameters:

p_length	When the header is present, used to provide number of characters to allow for the header value.
index	Selects element from array, index 0 refers to the first element.

Returns: TRUE if the header is present, and the header length is provided. Otherwise, FALSE.

GetUserDefined ( )

This function returns a copy of the selected user defined entry to the caller's buffer.

Prototype:

```
BOOL GetUserDefined (
                        CObexUserDefined    *p_user_defined,
                        UINT16              index);
```

Parameters:

p_user_defined	Pointer to caller's CObexUserDefined object
index	Selects user defined entry from array, index 0 refers to the first element.

Returns: TRUE if the indexed user defined parameter exists. Otherwise, FALSE

COBEXUSERDEFINED

This class contains a user defined header. The header identifier must be in the range 0x30–0x3F, so as not to conflict with predefined OBEX header identifiers.

SetHeader ( )

There are multiple signatures for this method, to support different data types for the header value.

Single Byte Header

Prototype:

```
BOOL SetHeader (
                 UINT8     id,
                 UINT8     byte);
```

Parameters:	id	Header identifier
	byte	Single byte value

Returns: TRUE if header value is valid; FALSE if the identifier not in the range 0x30–0x3F.

Four Byte Header

Prototype:

```
BOOL SetHeader (
                 UINT8     id,
                 UINT32    four_byte);
```

Parameters:	id	Header identifier
	four_byte	Four byte value

Returns: TRUE if header value is valid; FALSE if the identifier not in the range 0x30–0x3F.

UNICODE Text Header

Prototype:

```
BOOL SetHeader (
                 UINT8     id,
                 WCHAR     *p_text);
```

Parameters:	id	Header identifier
	p_text	Pointer to null-terminated UNICODE string

Returns: TRUE if header value is valid and string length <65536; FALSE if the identifier not in the range 0x30–0x3F or length too long

Octet Array Header

Prototype:

```
BOOL SetHeader (
                 UINT8     id,
                 UINT8     *p_array,
                 UINT16    length);
```

Parameters:	id	Header identifier
	p_array	Pointer to unstructured octet array
	length	Length of array

Returns: TRUE if header value is valid; FALSE if the identifier not in the range 0x30–3F.

GetUserType ( )

This function returns a value corresponding to user defined header types. The CObexUserDefined constructor initializes the object as a four byte type with value 0. Therefore, a valid type is always defined, even if a SetHeader ( ) call is never made.

Prototype: `UINT8 GetUserType (UINT8 *p_id);`

Parameters: p_id ID of user defined type stored here, value 0x30–0x3F.

Returns: One of the defined types:
OBEX_USER_TYPE_UNI – for NULL terminated UNICODE text
OBEX_USER_TYPE_ARRAY – for unstructured octet array
OBEX_USER_TYPE_BYTE – for the single byte
OBEX_USER_TYPE_INT – for the four byte (unsigned int) value

GetByte ( )

Prototype: `BOOL GetByte (UINT8 *p_byte);`

Parameters: p_byte Points to caller's field where value is to be copied.

Returns: TRUE if the header really is of type OBEX_USER_TYPE_BYTE; FALSE, otherwise.

GetFourByte ( )

Prototype: `BOOL GetFourByte (UINT32 *p_fourbyte);`

Parameters: p_fourbyte Points to caller's field where value is to be copied.

Returns: TRUE if the header really is of type OBEX_USER_TYPE_INT; FALSE, otherwise.

GetLength ( )

This function returns the length of the user defined parameter header.

Prototype: `UINT16 GetLength ( );`

Parameters: None

Returns: Length

GetText ( )

Prototype: `BOOL GetText (WCHAR *p_text);`

Parameters: p_text Points to caller's field where value is to be copied, including the terminating null.

Returns: TRUE if the header is of type OBEX_USER_TYPE_UNI and a pointer has been set; FALSE; otherwise.

GetOctets ( )

Prototype: `BOOL GetOctets (UINT8 *p_octet_array);`

Parameters: p_octet_array Points to caller's field where value is to be copied

Returns: TRUE if the header is of type OBEX_USER_TYPE_ARRAY and a pointer has been set; FALSE, otherwise.

COBEXCLIENT

This class allows a client-side application to establish an OBEX connection to a server. Once a connection is established the client application may send OBEX Get, SetPath, Put, Abort, and Close requests to the server. The server responds to each request with a response.

This class defines pure virtual methods to process responses to the Open and Close requests. The application must provide a derived class defining the *OnOpen ( )* and *OnClose ( )* methods.

This class defines virtual methods to process responses to the Get, Put, SetPath, and Abort requests. The application derived class should provide methods for those functions that are actually used in the application.

Before a connection can be attempted, the client application must first obtain an OBEX server Bluetooth device address using the CBtIf class for inquiry and service discovery. The application then invokes the *Open ( )* method. When the *OnOpen ( )* method is called with a successful confirmation, the application can proceed with *Get ( )*, *Put ( ), SetPath ( )* and *Abort ( ),* as appropriate for the application.

The *Close ( )* method is called to end the session.

tOBEX_ERRORS

A common set of return codes are provided by the OBEX function calls. Use enumerated type tOBEX_ERRORS, from BtIfDefinitions.h

Table 25: tOBEX_ERRORS

tOBEX_ERRORS Value	***Meaning***
OBEX_SUCCESS	Operation was successful or accepted.
OBEX_FAIL	Operation failed or was rejected.
OBEX_ERROR	Internal OBEX error.
OBEX_ERR_RESOURCES	Insufficient resources.
OBEX_ERR_NO_CB	Callback for request is missing.
OBEX_ERR_DUP_SERVER	Server for *Target* already registered with OBEX.
OBEX_ERR_RESPONSE	Peer rejected request.
OBEX_ERR_UNK_APP	Unknown Application Handle (unregistered).
OBEX_ERR_PARAM	Invalid or missing parameter value.
OBEX_ERR_CLOSED	Session is closed.
OBEX_ERR_ABORTED	Operation was aborted.
OBEX_ERR_STATE	Request is invalid for current state.
OBEX_ERR_NA	API call not allowed at this time.
OBEX_ERR_HEADER	Invalid data or header in CObexHeaders object.
OBEX_ERR_TOO_BIG	The data presented in the CObexHeaders object is larger than the maximum size allowed for the request.
OBEX_ERR_TIMEOUT	Timeout.

Open ( )

This function opens a session with the selected server.

The application must construct the CObexHeaders object as required by the application.

If this is the first time an Open was called, the application registers with the local Bluetooth stack. The CObexClient destructor unregisters.

Prototype:

```
tOBEX_ERRORS Open (
                    UINT8           scn,
                    BD_ADDR         bd_addr,
                    CObexHeaders    *p_request,
                    UINT16          mtu = OBEX_DEFAULT_MTU);
```

Parameters:

scn	Service Channel Number obtained from service discovery using CBtIf
bd_addr	Device address of the selected OBEX server
p_request	Points to OBEX headers object, which contains the headers to make up a valid request
mtu	Maximum Transmission Unit. See BtIfDefinitions.h for definition of OBEX_DEFAULT_MTU. This is the maximum size the client application is able to accept for a single OBEX packet. If the peer application in the server tries to send a packet larger than this value, the core OBEX software in both the client and server will segment the large packet as necessary so that each segment is no larger than MTU. The value OBEX_DEFAULT_MTU not only serves as a default for the *Open ( )* call, but it is in practice an absolute upper limit on the MTU value. If the client tries to use an MTU > OBEX_DEFAULT_MTU, the core OBEX logic will use OBEX_DEFAULT_MTU as the MTU. Also, OBEX_DEFAULT_MTU serves as an absolute upper limit for packets sent to the server. Even if the server allows a larger value, the WIDCOMM OBEX client will not send larger packets. An MTU value < OBEX_DEFAULT_MTU will be honored.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139.

SetPath ( )

The client application calls this function to send a formatted OBEX SetPath request to the server.

Prototype:

```
tOBEX_ERRORS SetPath (
                        CObexHeaders *p_request,
                        BOOL         backup,
                        BOOL         create);
```

Parameters:

p_request	Points to OBEX message object which should contain a valid OBEX request message.
backup	TRUE indicates that the server should set the path up one level from the current directory.
create	TRUE indicates that the server should create a new folder (if one does not already exist for the current directory).

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139.

Put ( )

The client application calls this function to send a formatted OBEX Put request to the server.

Special conventions in constructing the CObexHeaders object before calling *Put ( )*:

1. To make a simple *put* request use *SetBody ( )*, with a nonzero length. The server accepts the body header and processes it according to the application logic.
2. To request the server to create a new object having the name provided in the name header, call *SetBody ( )* with a zero-length field.
3. To request the server to delete the object with name provided by the name header, do not call *SetBody ( ).* If the CObexHeaders object might already have a body header from previous usage, just call *DeleteBody ( ).*

Prototype:

```
tOBEX_ERRORS Put (
                 CObexHeaders     *p_request,
                 BOOL             final);
```

Parameters:

p_request	Points to OBEX message object which should contain a valid OBEX request message.
final	TRUE means this Put completes sending the object.

Returns: OBEX_SUCCESS – if all was OK
Otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139.

Get ( )

The client application calls this function to send a formatted OBEX Get request to the server.

Prototype:

```
tOBEX_ERRORS Get (
                 CObexHeaders  *p_request,
                 BOOL          final);
```

Parameters:

p_request	Points to OBEX message object which should contain a valid OBEX request message.
final	TRUE means this Get completes receiving the object.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139.

Abort ( )

The client application calls this function to send a formatted OBEX Abort request to the server.

Prototype: `tOBEX_ERRORS Abort (CObexHeaders *p_request);`

Parameters:

p_request	Points to OBEX message object which should contain a valid OBEX request message.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139.

Close ( )

The client application calls this function to send a formatted OBEX Close request to the server.

Prototype:	`tOBEX_ERRORS Close (CObexHeaders *p_request);`	
Parameters:	p_request	Points to OBEX message object which should contain a valid OBEX request message.
Returns:	OBEX_SUCCESS – if all was OK. Otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139.	

Pure virtual OnOpen ( )

This derived function must be defined by the application and is called when the Open Confirmation response is received from the server.

Prototype:

```
virtual void OnOpen (
                        CObexHeaders            *p_confirm,
                        UINT16                  tx_mtu,
                        tOBEX_ERRORS            code,
                        tOBEX_RESPONSE_CODE     response) = 0;
```

Parameters:	p_confirm	Contains the server's Open Confirmation response translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	tx_mtu	This is the maximum size the server application is able to accept for a single OBEX packet. If the peer application in the client tries to send a packet larger than this value, the core OBEX software in both the client and server will segment the large packet as necessary so that each segment is no larger than MTU. In that case there will be some inefficiency on the Bluetooth connection. A good rule-of-thumb is for the client to send buffers of length tx_mtu – 6. This allows for the 6 byte overhead needed by lower protocol layers. The value OBEX_DEFAULT_MTU is an absolute upper limit on the MTU value. If the server tries to use an MTU > OBEX_DEFAULT_MTU, the core OBEX logic will use OBEX_DEFAULT_MTU as the MTU.
	code	Locally generated result code. OBEX_SUCCESS if the request succeeded; otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139 for the possible error cases.
	response	Response code sent by server; see BtIfDefinitions.h for list of values.
Returns:	void	

Pure virtual OnClose ( )

This derived function must be defined by the application and is called when the Close Confirmation response is received from the server.

Prototype:

```
virtual void OnClose (
                         CObexHeaders        *p_confirm,
                         tOBEX_ERRORS        code,
                         tOBEX_RESPONSE_CODE response) = 0;
```

Parameters:	p_confirm	Contains the server’s Close Confirmation response translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	code	Locally generated result code. OBEX_SUCCESS if the request succeeded; otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139 for the possible error cases.
	response	Response code sent by server; see BtIfDefinitions.h for list of values.
Returns:	void	

virtual OnAbort ( )

This derived function is defined by the application and is called when the Abort Confirmation response is received from the server.

Prototype:

```
virtual void OnAbort (
                         CObexHeaders        *p_confirm,
                         tOBEX_ERRORS        code,
                         tOBEX_RESPONSE_CODE response);
```

Parameters:	p_confirm	Contains the server’s Abort Confirmation response translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	code	Locally generated result code. OBEX_SUCCESS if the request succeeded; otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139 for the possible error cases.
	response	Response code sent by server; see BtIfDefinitions.h for list of values.
Returns:	void	

virtual OnPut ( )

This derived function is defined by the application and is called when the Put Confirmation response is received from the server.

Prototype:

```
virtual void OnPut (
                       CObexHeaders        *p_confirm,
                       tOBEX_ERRORS        code,
                       tOBEX_RESPONSE_CODE response);
```

Parameters:	p_confirm	Contains the server's Put Confirmation response translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	code	Locally generated result code. OBEX_SUCCESS if the request succeeded; otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139 for the possible error cases.
	response	Response code sent by server; see BtIfDefinitions.h for list of values.
Returns:	void	

virtual OnGet ( )

This derived function is defined by the application and is called when the Get Confirmation response is received from the server.

Prototype:

```
virtual void OnGet (
                   CObexHeaders            *p_confirm,
                   tOBEX_ERRORS            code,
                   BOOL                    final,
                   tOBEX_RESPONSE_CODE     response)
```

Parameters:	p_confirm	Contains the server's Get Confirmation response translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	code	Locally generated result code. OBEX_SUCCESS if the request succeeded; otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139 for the possible error cases.
	final	Set TRUE by the server when this is the final transfer for an object.
	response	Response code sent by server; see BtIfDefinitions.h for list of values.
Returns:	void	

virtual OnSetPath ( )

This derived function is defined by the application and is called when the OnSetPath Confirmation response is received from the server.

Prototype:

```
virtual void OnSetPath (
                CObexHeaders          *p_confirm,
                tOBEX_ERRORS          code,
                tOBEX_RESPONSE_CODE   response)
```

Parameters:	p_confirm	Contains the server's OnSetPath Confirmation response translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	code	Locally generated result code. OBEX_SUCCESS if the request succeeded; otherwise, see tOBEX_ERRORS Definitions in Table 25 on page 139 for the possible error cases.
	response	Response code sent by server; see BtIfDefinitions.h for list of values.
Returns:	void	

SetLinkSupervisionTimeOut ( )

This function sets the link supervision timeout value for the connected device. The timeout value is used by the master or slave Bluetooth device to monitor link loss. The same timeout value is used for both SCO and ACL connections for the device. An ACL connection has to be established before calling this function. The Zero value for timeout will disable the Link_Supervision_Timeout check for the connected device. The default timeout is 20 seconds.

Prototype: `BOOL SetLinkSupervisionTimeOut (UINT16 timeoutSlot)`

Parameters:	timeoutSlot	The timeout duration in number of slots. Each slot is .625 milliseconds. ***Note:*** Zero will disable Link_Supervision_Timeout check.
Returns:	TRUE, if successful; FALSE, otherwise.	

SetWriteTimeOut ( )

This function sets the timeout for server response to any OBEX Client write operation to the server. It can only be used after a successful *Open ( )* call returns a session handle. The new timeout setting persists for the duration of the session, or until changed by this function again. Setting timeout value 0 resets the default (60 seconds).

This function can be used in cases where long delays may be expected while the server device accepts data, if it will not respond until after it has processed a certain amount (for example, sending large files to a slow printer).

Prototype: `tOBEX_ERRORS SetWriteTimeOut (UINT16 timeout);`

Parameters:	timeout	The new timeout value, in seconds.
Returns:	OBEX_SUCCESS – if successful. Otherwise, see Table 25 on page 139.	

AUDIO CONNECTIONS

Audio connection can be established between Bluetooth devices and associated with a particular OBEX connection. The application uses the following methods to manage these connections. The reader is directed to the Audio Connections discussion in the CBtIf class description above for an overview of audio connections.

CreateAudioConnection ( )

This method is used to establish an audio connection with a remote device. The client must have already established the OBEX data connection using the *CObexClient::Open ( )* method prior to initiating the audio connection.

Once the audio connection is established, it will be associated with the OBEX connection. As such, when the OBEX connection is closed, the audio connection will automatically be closed.

The application must implement *OnAudioConnected ( )* to get the connection establishment notice.

The application may establish the audio connection in either the server mode (listen for a connection establishment from the remote device) or in the client mode (initiate a connection to the remote device).

Prototype:

```
AUDIO_RETURN_CODE CreateAudioConnection (
                                 BOOL          bIsClient,
                                 UINT16        *audioHandle);
```

Parameters:	bIsClient	TRUE if this is a client connection (initiator); otherwise, it is a server connection (listener).
	audioHandle	Handle to the opened audio connection.
Returns:	See definition of AUDIO_RETURN_CODE in Table 8 on page 35.	

RemoveAudioConnection ( )

This method is used to disconnect an audio connection. Note that usage of this method is optional. The audio connection will be disconnected automatically after the OBEX connection is closed.

Prototype: `AUDIO_RETURN_CODE RemoveAudioConnection (UINT16 audioHandle);`

Parameters:	audioHandle	Handle to an open audio connection.
Returns:	See definition of AUDIO_RETURN_CODE in Table 8 on page 35.	

virtual OnAudioConnected ( )

This is a virtual function. It is used to indicate that an audio connection has been established. Applications may provide a derived method to process the connection establishment notice.

Prototype: `virtual void OnAudioConnected (UINT16 audioHandle);`

Parameters:	audioHandle	Handle to the audio connection.
Returns:	void	

virtual OnAudioDisconnect ( )

This is a virtual function. It is used to indicate an audio connection has been disconnected. Applications may provide a derived method to process the disconnection notice.

Prototype:	`virtual void OnAudioDisconnect (UINT16 audioHandle);`	
Parameters:	audioHandle	Handle to the disconnected audio connection.
Returns:	void	

SetEscoMode ( )

This method is used to set up the negotiated parameters for SCO or eSCO, and set the default mode used for *CreateAudioConnection ( ).* It can be called only when there are no active (e)SCO links.

Prototype:

```
AUDIO_RETURN_CODE SetEScoMode (  tBTM_SCO_TYPE       sco_mode,
                                 tBTM_ESCO_PARAMS    *p_parms);
```

Parameters:	sco_mode	The desired audio connection mode: • BTM_LINK_TYPE_SCO for SCO audio connections. • BTM_LINK_TYPE_ESCO for eSCO audio connections.
	p_parms	A pointer to the (e)SCO link settings to use.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_PARAMS structure definition is:

```
typedef struct
{
   UINT32         tx_bw;
   UINT32         rx_bw;
   UINT16         max_latency;
   UINT16         voice_contfmt;
   UINT16         packet_types;
   UINT8          retrans_effort;
} tBTM_ESCO_PARAMS;
```

Where:

- tx_bw – Transmit bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- rx_bw – Receive bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- max_latency – maximum latency. This is a value in milliseconds, from 0x0004 to 0xfffe. Set to 0xffff for "don't care" setting.
- voice_contfmt – voice content format; CSVD is supported: BTM_ESCO_VOICE_SETTING
- packet_types – the packet type; EV3 is supported: BTM_ESCO_PKT_TYPES_MASK_EV3
- retrans_effort – the retransmit effort; one of:
  - BTM_ESCO_RETRANS_OFF
  - BTM_ESCO_RETRANS_POWER
  - BTM_ESCO_RETRANS_QUALITY
  - BTM_ESCO_RETRANS_DONTCARE

RegForEScoEvts ( )

This function registers an eSCO event callback with the specified object instance. It should be used to receive connection indication events and change of link parameter events.

Prototype:	`AUDIO_RETURN_CODE RegForEScoEvts ( UINT16 audioHandle, tBTM_ESCO_CBACK *p_esco_cback);`	
Parameters:	audioHandle	The handle to an open audio connection.
	p_esco_cback	A pointer to the callback function.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_CBACK definition is:

```
typedef void (tBTM_ESCO_CBACK) ( tBTM_ESCO_EVT      event,
                                 tBTM_ESCO_EVT_DATA *p_data);
```

The tBTM_ESCO_EVT definition is:

```
typedef UINT8  tBTM_ESCO_EVT;
```

The event could be one of the following values:

- BTM_ESCO_CHG_EVT – change of eSCO link parameter event.
- BTM_ESCO_CONN_REQ_EVT – connection indication event.

The tBTM_ESCO_EVT_DATA structure definition is:

```
typedef union
{
   tBTM_CHG_ESCO_EVT_DATA       chg_evt;
   tBTM_ESCO_CONN_REQ_EVT_DATA  conn_evt;
} tBTM_ESCO_EVT_DATA;

typedef struct
{
   UINT16          sco_inx;
   UINT16          rx_pkt_len;
   UINT16          tx_pkt_len;
   BD_ADDR         bd_addr;
   UINT8           hci_status;
   UINT8           tx_interval;
   UINT8           retrans_window;
} tBTM_CHG_ESCO_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- rx_pkt_len – The receive pocket length.
- tx_pkt_len – The transmit pocket length.
- bd_addr – The Bluetooth address of peer device.
- hci_status – HCI status.
- tx_interval – The transmit interval.
- retrans_window – re-transmit window.

```
typedef struct
{
   UINT16          sco_inx;
   BD_ADDR         bd_addr;
   DEV_CLASS       dev_class;
   tBTM_SCO_TYPE   link_type;
} tBTM_ESCO_CONN_REQ_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- bd_addr – The Bluetooth address the device.
- dev_class – The device class.
- link_type – The audio connection type.

ReadEScoLinkData ( )

This function retrieves current eSCO link data for the specified handle. This can be called anytime when a connection is active.

Prototype:	`AUDIO_RETURN_CODE ReadEScoLinkData ( UINT16 audioHandle, tBTM_ESCO_DATA *p_data);`	
Parameters:	audioHandle	The handle of an open audio connection.
	p_data	A pointer to the buffer where the current eSCO link settings will be stored.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_DATA structure definition is:

```
typedef struct
{
   UINT16       rx_pkt_len;
   UINT16       tx_pkt_len;
   BD_ADDR      bd_addr;
   UINT8        link_type
   UINT8        tx_interval;
   UINT8        retrans_window;
   UINT8        air_mode;
} tBTM_ESCO_DATA;
```

Where:

- rx_pkt_len – The transmit packet length.
- tx_pkt_len – The receive packet length.
- bd_addr – The device Bluetooth device address.
- link_type – the current audio connection type. It could be one of the following values:
  - BTM_LINK_TYPE_SCO
  - BTM_LINK_TYPE_ESCO
- tx_interval – The transmit interval.
- retrans_window – The retransmit window.
- air_mode – The air mode.

ChangeEscoLinkParms ( )

This function requests renegotiation of the parameters on the current eSCO link. If any of the changes are accepted by the controllers, the BTM_ESCO_CHG_EVT event is sent via the callback function registered in *RegForEScoEvts ( ),* with the current settings of the link.

Prototype:	`AUDIO_RETURN_CODE ChangeEScoLinkParms ( UINT16 audioHandle, tBTM_CHG_ESCO_PARAMS *p_parms);`	
Parameters:	audioHandle	The handle to an open audio connection.
	p_parms	A pointer to the settings that need to be changed.
Returns:	See Table 8: “Audio Return Codes,” on page 35.	

The tBTM_CHG_ESCO_PARAMS structure definition is follows:

```
typedef struct
{
   UINT16     max_latency;
   UINT16     packet_types;
   UINT8      retrans_effort;
} tBTM_CHG_ESCO_PARAMS;
```

Where:

- max_latency – maximum latency. This is a value in milliseconds.
- packet_types – packet type.
- retrans_effort – The retransmit effort.

EScoConnRsp ( )

This function should be called upon receipt of an eSCO connection request event (BTM_ESCO_CONN_REQ_EVT) to accept or reject the request.

The hci_status parameter should be [0x00] to accept, [0x0d .. 0x0f] to reject.

The p_params tBTM_ESCO_PARAMS pointer argument is used to negotiate the settings on the eSCO link. If p_parms is NULL, the values set through *SetEScoMode ( )* are used.

Prototype:

```
void EScoConnRsp (UINT16             audioHandle,
                  UINT8              hci_status,
                  tBTM_ESCO_PARAMS   *p_parms = NULL);
```

Parameters:	audioHandle	Handle to an open audio connection.
	hci_status	The HCI status. Zero is HCI success, defined as HCI_SUCCESS.
	p_parms	A pointer to the tBTM_ESCO_PARAMS to be negotiated.
Returns:	void	

COBEXSERVER

This class allows a server-side application to receive an OBEX connection from a client.

Once a connection is established the client application may send Get, Put, SetPath, Abort, and Close requests to the server. The server responds to each request with a response.

This class defines pure virtual methods to process all requests from the client. The server application must provide a derived class defining the *OnOpen ( ), OnGet ( ), OnPut ( ),* etc., as described below as pure virtual methods.

The server application must perform some preliminary functions before using the CObexServer class:

1. The server application must first use SDK class CRfCommIf to assign an SCN and a security level to the application GUID.
2. Create a service record. The SDK class CSdpService supports this function.
3. The server application must next register its service with the RFCOMM layer of the local Bluetooth stack. This is done by calling the *CObexServer::Register ( )* method, which also sets the server to a *listen* mode, listening for a client OBEX connection request.

When the *OnOpen ( )* method is called with an incoming Open request from a client, an OpenConfirm response is sent to the client.

Then the session continues, with the server responding to each *OnGet ( ), OnPut ( ), etc. or* with the corresponding confirm response.

Finally an *OnClose ( )* is called, and the server ends the session after sending a Close confirm.

The server is then available to process additional sessions.

Finally, to shut down, the server application must call *Unregister ( )* to remove the service from the local Bluetooth stack.

tOBEX_ERRORS

A common set of return codes are provided by the OBEX function calls. See Table 25: "tOBEX_ERRORS," on page 139.

Register ( )

This function registers the service with the OBEX protocol layer. This has the effect of initiating a *listen for client connection request* by the server.

The SCN parameter is required. Client and server applications will not be matched unless they specify the same scn value. Only one server application can be registered for a particular SCN.

The server application can optionally designate itself to be a *target*. Based on the target parameter the server platform can behave in the following ways:

- p_target is NULL, or p_target points to a zero length (NULL terminated) string. This server application will be connected to all incoming client requests for the scn, whether or not the client has specified targets in its open request.
- p_target points to a nonzero length (NULL terminated) string. This server application will connect with only those client requests that contain a target that matches.

An alternate method has been added to cater to an additional case where the target may be binary data, and thus not NULL terminated. The same rules above apply to that method, where a NULL p_target or a p_target with target_len 0 is used for the default server, and a non-zero target_len for a specific target server.
See *"RegisterBinaryTarget ( )"* on page 152.

The client application can send out an OBEX open request that optionally specifies none, one, or multiple targets for connection.

Clients and servers are matched up based on the target values set by the server applications and the client request. When the client requests more than one target, the targets are searched for in the order they appear in the clients headers object.

Prototype:

```
tOBEX_ERRORS Register (
                        UINT8 scn,
                        UINT8 *p_target = NULL);
```

Parameters:	scn	Service Channel Number obtained from CRfCommIf object.
	p_target	Null terminated string indicating the server target choice. See above.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

RegisterBinaryTarget ( )

This function is similar to function *Register ( )*, except that p_target points to binary data.
See "Register ( )" on page 151 above for more details.

Prototype:

```
tOBEX_ERRORS RegisterBinaryTarget (
                                   UINT8    scn,
                                   UINT8    *p_target,
                                   UINT16   target_len);
```

Parameters:	scn	Service Channel Number obtained from CRfCommIf object.
	p_target	Binary data indicating the server target.
	target_len	Binary data length.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

Unregister ( )

This function unregisters the service with the OBEX protocol layer; i.e., it dissociates the application from the OBEX protocol layer.

The CObexServer destructor will also unregister if the application fails to call *Unregister ( )* before terminating.

Prototype:

```
tOBEX_ERRORS Unregister ( );
```

Parameters: None

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

OpenCnf ( )

The server application calls this function to send a formatted OBEX confirmation message to the client, in response to the client's Open request.

Prototype:

```
tOBEX_ERRORS OpenCnf (
                    tOBEX_ERRORS            obex_errors,
                    tOBEX_RESPONSE_CODE     rsp_code,
                    CObexHeaders            *p_response,
                    UINT16                  mtu = OBEX_DEFAULT_MTU);
```

Parameters:

obex_errors	Result code from the Server application. Any code except OBEX_SUCCESS indicates a fatal error and OBEX will terminate the session.
rsp_code	Response code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_RESPONSE_CODE.
p_response	OBEX headers object containing response headers.
mtu	Maximum Transmission Unit. See BtIfDefinitions.h for definition of OBEX_DEFAULT_MTU. This is the maximum size the server application is able to accept for a single OBEX packet. If the peer application in the client tries to send a packet larger than this value, the core OBEX software in both the client and server will segment the large packet as necessary so that each segment is no larger than MTU. The value OBEX_DEFAULT_MTU not only serves as a default for the OpenCnf call, but it is in practice an absolute upper limit on the MTU value. If the server tries to use an MTU > OBEX_DEFAULT_MTU, the core OBEX logic will use OBEX_DEFAULT_MTU as the MTU. Also, OBEX_DEFAULT_MTU serves as an absolute upper limit for packets sent to the client. Even if the client allows a larger value, the WIDCOMM OBEX server will not send larger packets. An MTU value < OBEX_DEFAULT_MTU will be honored.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

SetPathCnf ( )

The server application calls this function to send a formatted OBEX confirmation message to the client, in response to the client's SetPath request.

Prototype:

```
tOBEX_ERRORS SetPathCnf (
                         tOBEX_ERRORS            obex_errors,
                         tOBEX_RESPONSE_CODE     rsp_code,
                         CObexHeaders            *p_response);
```

Parameters:

obex_errors	Result code from the Server application. Any code except OBEX_SUCCESS indicates a fatal error and OBEX will terminate the session.
rsp_code	Response code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_RESPONSE_CODE
p_response	OBEX headers object containing response headers.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

PutCnf ( )

The server application calls this function to send a formatted OBEX confirmation message to the client, in response to the client's Put request.

Prototype:

```
tOBEX_ERRORS PutCnf (
                     tOBEX_ERRORS           obex_errors,
                     tOBEX_RESPONSE_CODE    rsp_code,
                     CObexHeaders           *p_response);
```

Parameters:

obex_errors	Result code from the Server application. Any code except OBEX_SUCCESS indicates a fatal error and OBEX will terminate the session.
rsp_code	Response code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_RESPONSE_CODE.
p_response	OBEX headers object containing response headers.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

PutCreateCnf ( )

The server application calls this function to send a formatted OBEX confirmation message to the client, in response to a client's Put request, when the Put request had the special form indicating that a *Create Empty Object* was being requested. This special form of Put has no *body header* and an empty *end of body header*.

Lower levels of the OBEX protocol convert this to a normal Put Confirmation.

Prototype:

```
tOBEX_ERRORS PutCreateCnf (
                     tOBEX_ERRORS           obex_errors,
                     tOBEX_RESPONSE_CODE    rsp_code,
                     CObexHeaders           *p_response);
```

Parameters:

obex_errors	Result code from the Server application. Any code except OBEX_SUCCESS indicates a fatal error and OBEX will terminate the session.
rsp_code	Response code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_RESPONSE_CODE.
p_response	OBEX headers object containing response headers.

Returns: OBEX_SUCCESS – if all was OK.Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

PutDeleteCnf ( )

The server application calls this function to send a formatted OBEX confirmation message to the client, in response to a client's Put request, when the Put request had the special form indicating that a *Delete File Object* was being requested. This special form of Put has no *body header* and no *end of body header*.

Lower levels of the OBEX protocol convert this to a normal Put Confirmation.

Prototype:

```
tOBEX_ERRORS PutDeleteCnf (
                             tOBEX_ERRORS            obex_errors,
                             tOBEX_RESPONSE_CODE     rsp_code,
                             CObexHeaders            *p_response);
```

Parameters:

obex_errors	Result code from the Server application. Any code except OBEX_SUCCESS indicates a fatal error and OBEX will terminate the session.
rsp_code	Response code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_RESPONSE_CODE.
p_response	OBEX headers object containing response headers.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

GetCnf ( )

The server application calls this function to send a formatted OBEX confirmation message to the client, in response to the client's Get request.

Prototype:

```
tOBEX_ERRORS GetCnf (
                       tOBEX_ERRORS            obex_errors,
                       tOBEX_RESPONSE_CODE     rsp_code,
                       BOOL                    final,
                       CObexHeaders            *p_response);
```

Parameters:

obex_errors	Result code from the Server application. Any code except OBEX_SUCCESS indicates a fatal error and OBEX will terminate the session.
rsp_code	Response code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_RESPONSE_CODE.
final	TRUE if this completes the request.
p_response	OBEX headers object containing response headers.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: "tOBEX_ERRORS," on page 139.

AbortCnf ( )

The server application calls this function to send a formatted OBEX confirmation message to the client, in response to the client’s Abort request.

Prototype:

```
tOBEX_ERRORS AbortCnf (
                         tOBEX_ERRORS            obex_errors,
                         tOBEX_RESPONSE_CODE     rsp_code,
                         CObexHeaders            *p_response);
```

Parameters:	obex_errors	Result code from the Server application. Any code except OBEX_SUCCESS indicates a fatal error and OBEX will terminate the session.
	rsp_code	Response code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_RESPONSE_CODE.
	p_response	OBEX headers object containing response headers.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: “tOBEX_ERRORS,” on page 139.

CloseCnf ( )

The server application calls this function to send a formatted OBEX confirmation message to the client, in response to the client’s Close request.

Prototype:

```
tOBEX_ERRORS CloseCnf (
                         tOBEX_ERRORS            obex_errors,
                         tOBEX_RESPONSE_CODE     rsp_code,
                         CObexHeaders            *p_response);
```

Parameters:	obex_errors	Result code from the Server application. Any code except OBEX_SUCCESS indicates a fatal error and OBEX will terminate the session.
	rsp_code	Response code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_RESPONSE_CODE.
	p_response	OBEX headers object containing response headers.

Returns: OBEX_SUCCESS – if all was OK.
Otherwise, see Table 25: “tOBEX_ERRORS,” on page 139.

GetRemoteBDAddr ( )

The server application calls this function to find the Bluetooth Device Address of the remote client device that initiated the connection.

Prototype: `void GetRemoteBDAddr (BD_ADDR_PTR p_bd_addr);`

Parameters:	p_bd_addr	Pointer to BD_ADDR into which the remote client’s BD Address will be returned.

Returns: void

SwitchRole ( )

The application uses this method to request that the device switch role to Master or Slave. If the application desires to accept multiple simultaneous connections, it must execute this command to request that the device switch role to Master.

Prototype:	`BOOL SwitchRole (MASTER_SLAVE_ROLE new_role);`	
Parameters:	new_role	The role to which the local device should switch. Valid values are NEW_MASTER or NEW_SLAVE.
Returns:	TRUE if successful.	

Pure virtual OnOpenInd ( )

This derived function must be defined by the application and is called when the Open request is received from the client.

Prototype:	`virtual void OnOpenInd (CObexHeaders *p_request) = 0;`	
Parameters:	p_request	Contains the server's Open request. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
Returns:	void	

Pure virtual OnSetPathInd ( )

This derived function must be defined by the application and is called when the OnSetPath request is received from the client.

Prototype:

```
virtual void OnSetPathInd (
                           CObexHeaders     *p_request,
                           BOOL             backup,
                           BOOL             create) = 0;
```

Parameters:	p_request	Contains the server's OnSetPath request translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	backup	TRUE means the server is requested to change the current path up one level in the hierarchy.
	create	TRUE means the server is requested to create a new folder with the object's name, if one does not already exist.
Returns:	void	

Pure virtual OnPutInd ( )

This derived function must be defined by the application and is called when the Put request is received from the client.

Prototype:
```
virtual void OnPutInd (
                        CObexHeaders     *p_request,
                        BOOL             final) = 0;
```

Parameters:	p_request	Contains the server's Put request. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	final	TRUE means this is the last Put request for the object.
Returns:	void	

Pure virtual OnPutCreateInd ( )

This derived function must be defined by the application and is called when a special form of Put request is received from the client — the special form indicating that a *Create Empty Object* was being requested. This special form of Put has no *body header* and an empty *end of body header.*

Prototype:
```
virtual void OnPutCreateInd (CObexHeaders *p_request) = 0;
```

Parameters:	p_request	Contains the server's Put request. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
Returns:	void	

Pure virtual OnPutDeleteInd ( )

This derived function must be defined by the application and is called when a special form of Put request is received from the client — the special form indicating that a *Create Empty Object* was being requested. This special form of Put has no *body header* and no *end of body header*.

Prototype:
```
virtual void OnPutDeleteInd (CObexHeaders *p_request) = 0;
```

Parameters:	p_request	Contains the server's Put request translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
Returns:	void	

Pure virtual OnGetInd ( )

This derived function must be defined by the application and is called when the Get request is received from the client.

Prototype:
```
virtual void OnGetInd (
                          CObexHeaders *p_request,
                          BOOL final) = 0;
```

Parameters:	p_request	Contains the server's Get request translated into an OBEX message object. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
	final	TRUE means this is the last Get request for the object.
Returns:	void	

Pure virtual OnAbortInd ( )

This derived function must be defined by the application and is called when the Abort request is received from the client.

Prototype: `virtual void OnAbortInd (CObexHeaders *p_request) = 0;`

Parameters:	p_request	Contains the server's Abort request. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
Returns:	void	

Pure virtual OnCloseInd ( )

This derived function must be defined by the application and is called when the Close request is received from the client.

Prototype: `virtual void OnCloseInd (CObexHeaders *p_request) = 0;`

Parameters:	p_request	Contains the server's Close request. ***Note:*** This object will be deallocated after the application returns. So the application must extract all required information before returning.
Returns:	void	

SetLinkSupervisionTimeOut ( )

This function sets the link supervision timeout value for the connected device. The timeout value is used by the master or slave Bluetooth device to monitor link loss. The same timeout value is used for both SCO and ACL connections for the device. An ACL connection has to be established before calling this function. The Zero value for timeout will disable the Link_Supervision_Timeout check for the connected device. The default timeout is 20 seconds.

Prototype: `BOOL SetLinkSupervisionTimeOut (UINT16 timeout);`

Parameters:	timeoutSlot	The timeout duration in number of slots. Each slot is .625 milliseconds. ***Note:*** Zero will disable Link_Supervision_Timeout check.
Returns:	TRUE, if successful. FALSE, otherwise.	

Audio Connections

Audio connections can be established between Bluetooth devices and associated with a particular OBEX connection. The application uses the following methods to manage these connections. The reader is directed to the Audio Connections discussion in the CBtIf class description above for an overview of audio connections.

CreateAudioConnection ( )

This method is used to establish an audio connection with a remote device. It listens for an incoming audio connection. Prior to executing this method, the *CObexServer::Register ( )* method must have already be executed to begin listening for an OBEX data connection prior to executing the *CObexServer::CreateAudioConnection ( )* method to begin listening for an audio connection.

Once the audio connection is established, it will be associated with the OBEX connection. As such, when the OBEX connection is closed, the audio connection will automatically be closed.

The application must implement *OnAudioConnected ( )* to get the connection establishment notice.

The application may establish the audio connection in either the server mode (listen for a connection establishment from the remote device) or in the client mode (initiate a connection to the remote device).

There are multiple signatures for this method. The orignal (Basic) version was supplemented with the Enhanced version, which allows a server application to specify a particular Bluetooth device address. The server will then only accept connections from that specific address.

Basic Version

Prototype:

```
AUDIO_RETURN_CODE CreateAudioConnection (
                                 BOOL          bIsClient,
                                 UINT16        *audioHandle);
```

Parameters:	bIsClient	TRUE if this is a client connection (initiator); otherwise, it is a server connection (listener).
	audioHandle	A pointer to where the handle for the new audio connection should be stored.
Returns:	See definition in Table 8: "Audio Return Codes," on page 35	

Enhanced Version

Prototype:

```
AUDIO_RETURN_CODE CreateAudioConnection (
                                    BOOL      bIsClient,
                                    UINT16    *audioHandle,
                                    BD_ADDR   bda);
```

Parameters:	bIsClient	TRUE if this is a client connection (initiator); otherwise, it is a server connection (listener).
	audioHandle	Pointer to where the handle for the new audio connection should be stored.
	bda	Specific remote device allowed to make connections (server-only). 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF indicates a connection will be accepted from any device.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

RemoveAudioConnection ( )

This method is used to disconnect an audio connection. Note that usage of this method is optional. The audio connection will be disconnected automatically after the OBEX connection is closed.

Prototype:	`AUDIO_RETURN_CODE RemoveAudioConnection (UINT16 audioHandle);`	
Parameters:	audioHandle	Handle to an open audio connection.
Returns:	See definition in Table 8: "Audio Return Codes," on page 35.	

virtual OnAudioConnected ( )

This is a virtual function. It is used to indicate that an audio connection has been established. Applications may provide a derived method to process the connection establishment notice.

Prototype:	`virtual void OnAudioConnected (UINT16 audioHandle);`	
Parameters:	audioHandle	Handle to the audio connection.
Returns:	void	

virtual OnAudioDisconnect ( )

This is a virtual function. It is used to indicate an audio connection has been disconnected. Applications may provide a derived method to process the disconnection notice.

Prototype:	`virtual void OnAudioDisconnect (UINT16 audioHandle);`	
Parameters:	audioHandle	Handle to the disconnected audio connection.
Returns:	void	

SetEscoMode ( )

This method is used to set up the negotiated parameters for SCO or eSCO, and set the default mode used for *CreateAudioConnection ( )*. It can be called only when there are no active (e)SCO links.

Prototype:

```
AUDIO_RETURN_CODE SetEScoMode (  tBTM_SCO_TYPE      sco_mode,
                                 tBTM_ESCO_PARAMS   *p_parms);
```

Parameters:	sco_mode	The desired audio connection mode: • BTM_LINK_TYPE_SCO for SCO audio connections. • BTM_LINK_TYPE_ESCO for eSCO audio connections.
	p_parms	A pointer to the (e)SCO link settings to use.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_PARAMS structure definition is:

```
typedef struct
{
   UINT32          tx_bw;
   UINT32          rx_bw;
   UINT16          max_latency;
   UINT16          voice_contfmt;
   UINT16          packet_types;
   UINT8           retrans_effort;
} tBTM_ESCO_PARAMS;
```

Where:

- tx_bw – Transmit bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- rx_bw – Receive bandwidth in octets per second. 64 kbits/sec data rate is supported: BTM_64KBITS_RATE.
- max_latency – maximum latency. This is a value in milliseconds, from 0x0004 to 0xfffe. Set to 0xffff for "don't care" setting.
- voice_contfmt – voice content format; CSVD is supported: BTM_ESCO_VOICE_SETTING
- packet_types – the packet type; EV3 is supported: BTM_ESCO_PKT_TYPES_MASK_EV3
- retrans_effort – the retransmit effort; one of:
  - BTM_ESCO_RETRANS_OFF
  - BTM_ESCO_RETRANS_POWER
  - BTM_ESCO_RETRANS_QUALITY
  - BTM_ESCO_RETRANS_DONTCARE

RegForEScoEvts ( )

This function registers an eSCO event callback with the specified object instance. It should be used to receive connection indication events and change of link parameter events.

Prototype:	`AUDIO_RETURN_CODE RegForEScoEvts ( UINT16 audioHandle, tBTM_ESCO_CBACK *p_esco_cback)`	
Parameters:	audioHandle	The handle to an open audio connection.
	p_esco_cback	A pointer to the callback function.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_CBACK definition is:

```
typedef void (tBTM_ESCO_CBACK) ( tBTM_ESCO_EVT      event,
                                 tBTM_ESCO_EVT_DATA *p_data);
```

The tBTM_ESCO_EVT definition is:

```
typedef UINT8  tBTM_ESCO_EVT;
```

The event could be one of the following values:

- BTM_ESCO_CHG_EVT – change of eSCO link parameter event.
- BTM_ESCO_CONN_REQ_EVT – connection indication event.

The tBTM_ESCO_EVT_DATA structure definition is:

```
typedef union
{
   tBTM_CHG_ESCO_EVT_DATA       chg_evt;
   tBTM_ESCO_CONN_REQ_EVT_DATA  conn_evt;
} tBTM_ESCO_EVT_DATA;

typedef struct
{
   UINT16         sco_inx;
   UINT16         rx_pkt_len;
   UINT16         tx_pkt_len;
   BD_ADDR        bd_addr;
   UINT8          hci_status;
   UINT8          tx_interval;
   UINT8          retrans_window;
} tBTM_CHG_ESCO_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- rx_pkt_len – The receive pocket length.
- tx_pkt_len – The transmit pocket length.
- bd_addr – The Bluetooth address of peer device.
- hci_status – HCI status.
- tx_interval – The transmit interval.
- retrans_window – re-transmit window.

```
typedef struct
{
  UINT16          sco_inx;
  BD_ADDR         bd_addr;
  DEV_CLASS       dev_class;
  tBTM_SCO_TYPE   link_type;
} tBTM_ESCO_CONN_REQ_EVT_DATA;
```

Where:

- sco_idx – The current SCO index.
- bd_addr – The Bluetooth address the device.
- dev_class – The device class.
- link_type – The audio connection type.

ReadEScoLinkData ( )

This function retrieves current eSCO link data for the specified handle. This can be called anytime when a connection is active.

Prototype:	`AUDIO_RETURN_CODE ReadEScoLinkData ( UINT16 audioHandle, tBTM_ESCO_DATA *p_data);`	
Parameters:	audioHandle	The handle of an open audio connection.
	p_data	A pointer to the buffer where the current eSCO link settings will be stored.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_ESCO_DATA structure definition is:

```
typedef struct
{
   UINT16       rx_pkt_len;
   UINT16       tx_pkt_len;
   BD_ADDR      bd_addr;
   UINT8        link_type
   UINT8        tx_interval;
   UINT8        retrans_window;
   UINT8        air_mode;
} tBTM_ESCO_DATA;
```

Where:

- rx_pkt_len – The transmit packet length.
- tx_pkt_len – The receive packet length.
- bd_addr – The device Bluetooth device address.
- link_type – the current audio connection type. It could be one of the following values:
  - BTM_LINK_TYPE_SCO
  - BTM_LINK_TYPE_ESCO
- tx_interval – The transmit interval.
- retrans_window – The retransmit window.
- air_mode – The air mode.

ChangeEscoLinkParms ( )

This function requests renegotiation of the parameters on the current eSCO link. If any of the changes are accepted by the controllers, the BTM_ESCO_CHG_EVT event is sent via the callback function registered in *RegForEScoEvts ( ),* with the current settings of the link.

Prototype:	`AUDIO_RETURN_CODE ChangeEScoLinkParms ( UINT16 audioHandle, tBTM_CHG_ESCO_PARAMS *p_parms);`	
Parameters:	audioHandle	The handle to an open audio connection.
	p_parms	A pointer to the settings that need to be changed.
Returns:	See Table 8: "Audio Return Codes," on page 35.	

The tBTM_CHG_ESCO_PARAMS structure definition is follows:

```
typedef struct
{
    UINT16      max_latency;
    UINT16      packet_types;
    UINT8       retrans_effort;
} tBTM_CHG_ESCO_PARAMS;
```

Where:

- max_latency – maximum latency. This is a value in milliseconds.
- packet_types – packet type.
- retrans_effort – The retransmit effort.

EScoConnRsp ( )

This function should be called upon receipt of an eSCO connection request event (BTM_ESCO_CONN_REQ_EVT) to accept or reject the request.

The hci_status parameter should be [0x00] to accept, [0x0d .. 0x0f] to reject.

The p_params tBTM_ESCO_PARAMS pointer argument is used to negotiate the settings on the eSCO link. If p_parms is NULL, the values set through *SetEScoMode ( )* are used.

Prototype:

```
void EScoConnRsp (UINT16            audioHandle,
                  UINT8             hci_status,
                  tBTM_ESCO_PARAMS  *p_parms = NULL);
```

Parameters:	audioHandle	Handle to an open audio connection.
	hci_status	The HCI status. Zero is HCI success, defined as HCI_SUCCESS.
	p_parms	A pointer to the tBTM_ESCO_PARAMS to be negotiated.
Returns:	void	

CPRINTCLIENT

This class allows the application to print files on Bluetooth enabled printers or a Bluetooth printer server.

Before a connection can be established, the client application must obtain a Bluetooth printer server device address using the CBtIf class for inquiry and service discovery.

This class then provides the start, cancel, get state, get bytes sent, and state change event functions.

ePRINT_PROFILE

Table 26 defines the profiles supported by the SDK.

Table 26: ePRINT_PROFILE

ePRINT_PROFILE Value	*Meaning*
PRINT_PROFILE_BPP	Basic Printing Profile
PRINT_PROFILE_HCRP	Hardcopy Cable Replacement Profile
PRINT_PROFILE_SPP	Serial Port Profile

ePRINT_STATE

Table 27 defines the current state of the print client.

Table 27: ePRINT_STATE

ePRINT_STATE Value	*Meaning*
PRINT_STATE_IDLE	Client in idle state
PRINT_STATE_CONNECTING	Connecting to printer
PRINT_STATE_PRINTING	Printing
PRINT_STATE_FLOW_CONTROLLED	Waiting for printer
PRINT_STATE_DISCONNECTING	Disconnecting from the printer
PRINT_STATE_DONE	Done printing

ePRINT_ERROR

Table 28 defines error codes returned by the print client.

Table 28: ePRINT_ERROR

ePRINT_ERROR Value	*Meaning*
Generic To All Profiles	
PRINT_RC_OK	Operation was successful or accepted.
PRINT_RC_FILE_PRINTED_OK	The file was printed successfully.
PRINT_RC_FILE_NOT_FOUND	The file to be printed could not be found.
PRINT_RC_FILE_READ_ERROR	Read file error.
PRINT_RC_ALREADY_PRINTING	The file is already printing.
PRINT_RC_UNKNOWN_PROFILE	Profile passed is unknown.
PRINT_RC_SERVICE_NOT_FOUND	Service not found.
PRINT_RC_SECURITY_ERROR	Security setting error.
PRINT_RC_CONNECT_ERROR	Connecting error.
PRINT_RC_WRITE_ERROR	Write error.
PRINT_RC_REMOTE_DISCONNECTED	Remote device disconnected.
PRINT_RC_INVALID_PARAM	Invalid or missing parameter value.
BPP-Specific Errors	
PRINT_RC_BPP_SCN_NOT_FOUND	BPP SCN not found.
PRINT_RC_BPP_SCN_NOT_ASSIGNED	BPP SCN not assigned.
PRINT_RC_BPP_OBEX_ABORTED	OBEX Operation was aborted.
PRINT_RC_BPP_OBEX_MISMATCH	OBEX mismatch.
HCRP-Specific Errors	
PRINT_RC_HCRP_CTL_PSM_NOT_FOUND	HCRP control PSM not found.
PRINT_RC_HCRP_DATA_PSM_NOT_FOUND	HCRP data PSM not found.
SPP-Specific Errors	
PRINT_RC_SPP_SCN_NOT_FOUND	SPP SCN not found.

BTPRINTSTRUCT

Table 29 defines the structure for using CPrintClass.

Table 29: BTPRINTSTRUCT

BTPRINTSTRUCT Type		***Meaning***
DWORD	dwSize	Must be size of (BTPRINTSTRUCT).
UINT	mask;	0 or BPSF_TYPE. If mask is set to BPSF_TYPE, then pszType is used. Otherwise TYPE_STRING_TEXT is used.
LPCSTR	pszType;	File data type to print. This is for BPP only.

Start ( )

This function sends printing requests to the printer server. Before calling this function the application must use a CBtIf class inquiry and discovery functions to locate a service offering PRINT_PROFILE_BPP, PRINT_PROFILE_HCRP or PRINT_PROFILE_SPP.

Prototype:
```
ePRINT_ERROR Start (
                        BD_ADDR          pBDA,
                        ePRINT_PROFILE   eProfile,
                        LPCSTR           pszFile,
                        BTPRINTSTRUCT    *pBtPrintStruct = NULL);
```

Parameters:	pBDA	The Bluetooth address of the printer device.
	eProfile	The profile to use to start printing.
	pszFile	File name to print.
	pBtPrintStruct	Default to NULL. If pBtPrintStruct is not NULL, struct member pszType is used

Returns: PRINT_RC_OK – Everything is OK.
Otherwise, see Table 28: “ePRINT_ERROR,” on page 166.

Cancel ( )

This function cancels a current printing request to the printer server.

Prototype: `void Cancel ( )`
Parameters: None.
Returns: void

virtual OnStateChange ( )

This callback function allows the application to detect the status of the print client.

Prototype: `virtual void OnStateChange (ePRINT_STATE state);`
Parameters: state See Table 27: “ePRINT_STATE,” on page 166.
Returns: void

GetState ( )

This function gets the print client state.

Prototype:	`ePRINT_STATE GetState ( );`
Parameters:	None
Returns:	ePRINT_STATE. See Table 27: "ePRINT_STATE," on page 166.

GetBytesSent ( )

This function returns the number of bytes sent to the printer server.

Prototype:	`UINT GetBytesSent ( );`
Parameters:	None
Returns:	The number of bytes sent to the printer server.

GetBytesTotal ( )

This function returns the total number of bytes in the print file.

Prototype:	`UINT GetBytesTotal ( );`
Parameters:	None
Returns:	The number of bytes sent to the printer server.

GetLastError ( )

This function returns the print client's last-error code value.

Prototype:	`ePRINT_ERROR GetLastError (BT_CHAR **pDescr);`	
Parameters:	pDescr	Pointer to a BT_CHAR array, set by the class to point to a string with textual error description. May contain more information usable in interpreting the ePRINT_ERROR return code.
Returns:	ePRINT_ERROR. See Table 28: "ePRINT_ERROR," on page 166.	

CHEADPHONECLIENT

This class allows the application to establish a connection to Bluetooth stereo headphones or any other device that offers the Bluetooth Advanced Audio Distribution Profile service (A2DP).

Before a connection can be established, the client application must obtain a Bluetooth headphone device address using the CBtIf class for inquiry and service discovery.

The CHeadphoneClient class then provides the functions to create and close a headphone connection, process connection state change events, set security, gather statistics, and process errors. To receive the state change event, the application must first register a callback.

HEADPHONE_RETURN_CODE

A common set of return codes is provided by the headphone function calls defined in the enumerated type `HEADPHONE_RETURN_CODE` from BtIfClasses.h.

Table 30: Headphone Return Codes

Value	***Meaning***
SUCCESS	Operation initiated without error.
NO_BT_SERVER	COM server could not be started.
ALREADY_CONNECTED	Attempt to connect before previous connection closed.
NOT_CONNECTED	Attempt to close unopened connection.
NOT_ENOUGH_MEMORY	Local processor could not allocate memory for open.
INVALID_PARAMETER	One or more function parameters are invalid.
UNKNOWN_ERROR	Any condition other than those defined here.
LICENSE_ERROR	Invalid license
DEVICE_BUSY	Device busy
SERVICE_NOT_FOUND	Service not found
BTM_WRONG_MODE	Device disabled or device is not up

HEADPHONE_STATUS

A status code is returned in the status change event callback function. The status code is defined by the enumerated type `HEADPHONE_STATUS` from BtIfClasses.h.

Table 31: Headphone Status Values

Value	*Meaning*
HEADPHONE_CONNECTED	Device is connected.
HEADPHONE_LOCAL_DISCONNECT	The connection is closed by local device.
HEADPHONE_REMOTE_DISCONNECT	The connection is closed by remote device.
HEADPHONE_DEVICE_NOT_AUTHORIZED	The device is not authorized.
HEADPHONE_NO_STREAM_FOUND	No stream found
HEADPHONE_REMOTE_SUSPENDED	Remote suspend the streaming
HEADPHONE_INCOMING_STREAM	Incoming stream
HEADPHONE_STREAMING	Streaming data
HEADPHONE_STOPPED	Stop streaming

CONNECTHEADPHONE ( )

This function creates a connection to a headphone. Before calling this function, the application must use the CBtIf class to locate a device that offers the headphone service. To get connection status, the application needs to register the callback by calling function *RegStatusChangeCB ( )*.

Prototype:

```
HEADPHONE_RETURN_CODE ConnectHeadphone (
                                       BD_ADDR   bda,
                                       LPCSTR    szServiceName);
```

Parameters:	bda	The server's Bluetooth device address. If this parameter is NULL, an INVALID_PARAMETER error is returned.
	szServiceName	The server's service name, default to NULL.
Returns:	SUCCESS	Everything is okay
	Otherwise	See Table 30: "Headphone Return Codes," on page 169

DISCONNECTHEADPHONE ( )

This function closes the connection.

Prototype:

```
HEADPHONE_RETURN_CODE DisconnectHeadphone ( long hHandle);
```

Parameters:	hHandle	The connection handle. This value is returned as a parameter of the status change event callback function.
Returns:	SUCCESS	Everything is okay
	Otherwise	See Table 30: "Headphone Return Codes," on page 169

SETSECURITY ( )

This function sets the authentication and encryption parameters for a headphone connection.

Encryption translates data into an unreadable format using a secret key generated during the authentication process. Decrypting the data requires the same key that was used to encrypt it.

Encryption in Bluetooth for Windows is based on the same passkey or Link Key that is used for Authentication. If Authentication is not enabled, the key is not available and encryption will not take place.

To use Encryption, Authentication must be enabled.

If this function is not called, the headphone security setting in the Windows Registry will be used. Those settings are used by BTExplorer and controlled from the Advanced Configuration -> Client Applications tab in the BTTray application.

Prototype:

```
void SetSecurity (  BOOL  authentication,
                    BOOL  encryption);
```

Parameters:	authentication	TRUE means use authentication procedures on future operations using this object.
	encryption	TRUE means use encryption procedures on future data transfers that use this object.
Returns:	void	

GETEXTENDEDERROR ( )

This function returns the object's last-error code. The last-error code is maintained on a per-object basis. Multiple objects do not overwrite each other's last-error code.

This function is intended to obtain extended error information when the SDK return/result codes are non-specific. This is useful when the UNKNOWN_ERROR code is returned from an SDK API function.

To retrieve an extended error code for a method of this class in the case where the method has failed, call *GetExtendedError ( )* directly after the method has been called. *GetExtendedError ( )* can be called from callback functions of this class as well.

Prototype: `WBtRc GetExtendedError ( );`

Parameters: None

Returns: See the definition of the WBtRc enumeration in com_error.h.

SetExtendedError ( )

This function sets the last-error code for the object of this class. *SetExtendedError ( )* is intended to exactly identify the place where an error occurs in the code. It can be called to reset the last-error code to WBT_SUCCESS (see com_error.h) for an object of this class before any other class method is called. After a class method has been called, *GetExtendedError ( )* can be invoked to read the error information. If *GetExtendedError ( )* reports WBT_SUCCESS, then the error occurs somewhere else in the program.

The last-error code is kept in a local variable of the object so that multiple objects of this class do not overwrite each other's values.

Prototype:	`void SetExtendedError ( WBtRc code);`	
Parameters:	code	One of the values from the WBtRc enumeration defined in com_error.h
Returns:	void	

GetConnectionStats ( )

This function retrieves current connection statistics, using the common statistics structure tBT_CONN_STATS defined in BtIfDefinitions.h. See "GetConnectionStats ( )" on page 34.

This class uses all the statistic fields in the structure.

Prototype:	`HEADPHONE_RETURN_CODE GetConnectionStats ( tBT_CONN_STATS *p_conn_stats);`	
Parameters:	p_conn_stats	A pointer to the user's connection statistics structure, see above.
Returns:	SUCCESS	Everything is okay.
	Otherwise	See "HEADPHONE_RETURN_CODE" on page 169

RegStatusChangeCB ( )

This function is used to register a callback to receive events when a connection to the headphone server is established or cleared. This function needs to be called before calling *ConnectHeadphone ( )***.**

Prototype:

```
HEADPHONE_RETURN_CODE RegStatusChangeCB (
                 tOnHAGConnectionStatusChangedCallback      pOnHAGStatus,
                 void                                       *userData);
```

Parameters:	pOnHAGStatus	The pointer to the callback function.
	userData	This is a pointer to user-defined data that will be passed back to the user in the callback function.
Returns:	SUCCESS	Everything is okay.
	Otherwise	See "HEADPHONE_RETURN_CODE" on page 169

The callback function prototype is defined in the header file BtIfClasses.h as follows:

Prototype:

```
typedef void (*tOnHAGConnectionStatusChangedCallback) (
                                                        void       *userData,
                                                        BD_ADDR    bda,
                                                        DEV_CLASS  dev_class,
                                                        BD_NAME    bd_name,
                                                        long       lHandle,
                                                        long       lStatus);
```

Parameters:	userData	The pointer to the user-defined data that was passed to the RegStatusChangeCB function.
	bda	The Bluetooth device address
	dev_class	Device class.
	bd_name	The Bluetooth device name.
	lHandle	The connection handle. Need to use this handle to disconnect headphone.
	lStatus	Connection status. See "HEADPHONE_STATUS" on page 170
Returns:	void	

Section 6: References

1. Infrared Data Association, "Serial Infrared Link Access Protocol (IrLAP)", Version 1.0, June 23, 1994.
2. Infrared Data Association, IrDA Object Exchange Protocol (IrOBEX) with Published Errata, Version 1.2, April 1999.
3. Bluetooth Special Interest Group Specification website: http://www.bluetooth.org/spec.

Broadcom Corporation

5300 California Avenue
P.O. Box 57013
Irvine, CA 92617
Phone: 949-926-5000
Fax: 949-926-5203