Bluetooth for Windows
DK API Reference Guide

March 15, 2002

Document Number: BTW-SDK-DOCS-010308-1256

Version: 1.2

Confidential and Proprietary Information

Wireless Internet and Data Communication

9645 Scranton Road, Suite 205
San Diego, CA 92121
Phone: (858) 453-8400
Fax: (858) 453-5735
Email
Technical Support: support@widcomm.com
Information: info@widcomm.com

LICENSED SOFTWARE

Warning

Copyright law and international treaties protect this software and accompanying documentation. Unauthorized reproduction or distribution of this software, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.

Use of this software is governed by the terms of the end user license agreement that accompanies or is included with such software.

Unless otherwise noted in the end user license agreement, or herein, no part of the documentation accompanying this software, whether provided in printed or electronic form may be reproduced in any form, or stored in a database or retrieval system, or transmitted in any form or by any means, or used to make any derivative work (such as translation, transformation, or adaptation) without the express, prior written consent of WIDCOMM.

Copyright and Trademark Notices

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes WIDCOMM's Bluetooth™ for Windows Software Development Kit. The DK supports developers of custom Bluetooth applications with protocol-layer direct access to:

- L2CAP
- RFCOMM
- OPP
- FTP
- SDP
- SPP
- LAP
- OBEX

The DK consists of:

- A Dynamic Link Library (DLL).
- C++ header files.
- Source files for sample applications.
- This document, which provides usage and interface details for the DLL.

The operational context for the DK is a standard Bluetooth PC platform on which WIDCOMM BTW software has been installed.

Custom applications developed using the DK can be run concurrently with the standard WIDCOMM BT Neighborhood application and profiles.

NOTE: The DK software was developed and tested with Microsoft Visual C++ 6.0. Future releases will be tested on other development environments.

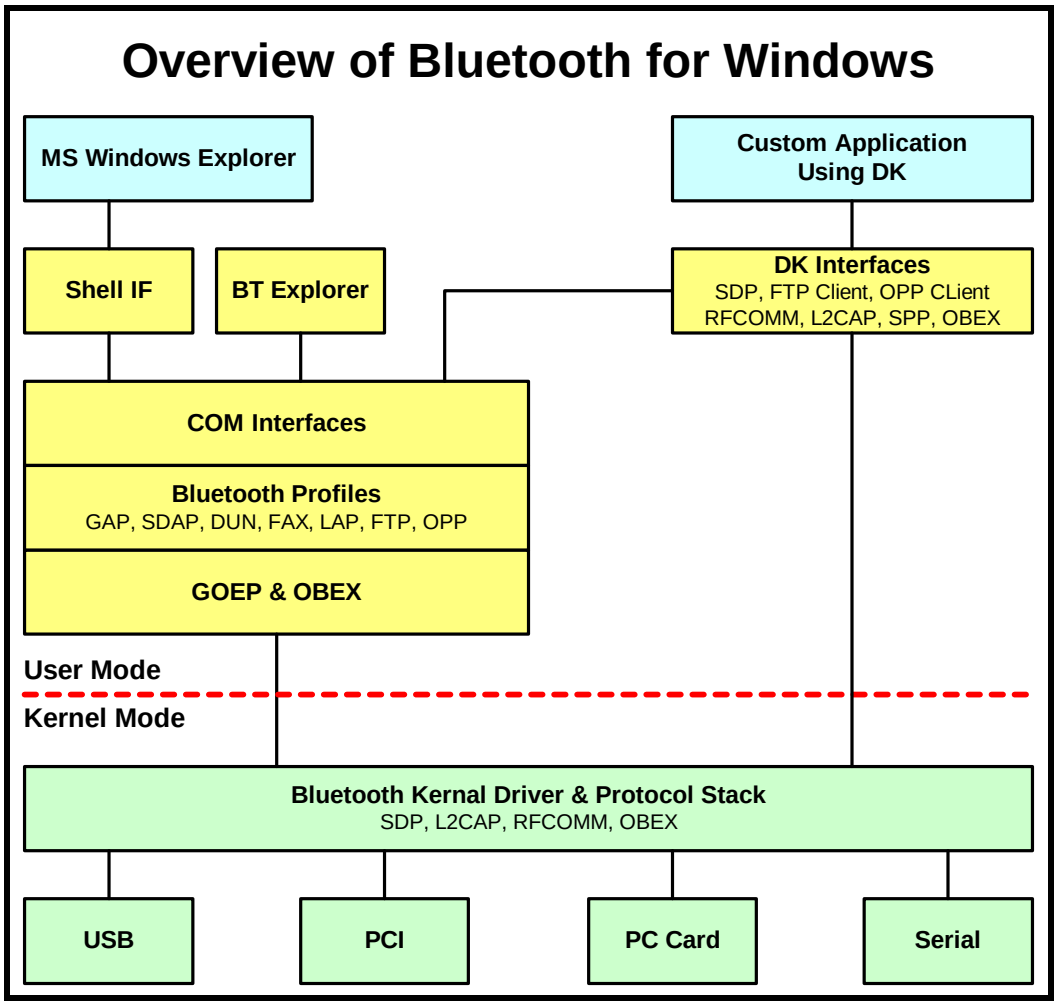
Functionally, the DK presents a C++ interface to the *Inquiry* and *Discovery* functions of the SDP layer.

DK functions allow access to the discovery database and to the "attribute" values contained there. The custom application can add to and delete from the SDP service database.

For the SDP, OPP, FTP, L2CAP, and RFCOMM layers, C++ classes are provided for all detailed functions exposed by the WIDCOMM Bluetooth stack implementation.

Figure 1 is an overview of the DK in the Windows Bluetooth context.

Figure 1: Block diagram/overview of the Bluetooth for Windows DK



2 System Requirements

In order to develop software using the WIDCOMM Bluetooth API, you will need the following hardware and software:

- Windows 98 SE / Me / 2000 (must have system administrator privileges under 2000)
- The evaluation or retail version of the BTW software from WIDCOMM, Inc.
- Microsoft Visual C++ 6.0 or Visual Studio 6.0 with service pack 5 installed
- 64 MB memory
- 50 MB free hard disk space (after VC++ or Visual Studio is installed)
- A USB port (a hub may be required)
- A USB Bluetooth dongle
- A CD-ROM drive for installation of the software and drivers

3 Implementation

The DK consists of a DLL, a library file and two header files.

- *WidcommSdk.dll* should be placed in a directory that is in the DLL search path, typically the working directory for the executable or the Windows system directory.
- Applications should be linked with *WidcommSdk.lib*.
- *BtIfDefinitions.h* is a header file that defines constants and structures used by the DK and applications.
- *BtIfClasses.h* is a header file that defines the classes offered by the DK.

Table 1: Classes offered in the DK.

Classes Offered In The DK	
DK Class	**Function**
CBtIf	Provides interface level management functions, e.g., methods for doing inquiry and service discovery.
CL2CapIf	Interfaces to 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 to RFCOMM for Service Channel Number (SCN) allocation and security settings.
CRfCommConn	Controls RFCOMM connections.
CFtpClient	Provides the client-side interface for FTP.
COppClient	Provides the client-side interface for OPP.
CLapClient	Provides the client-side interface for LAN access using PPP.
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

4 DK Classes

4.1 VIRTUAL FUNCTIONS

The DK 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 defines derived functions that substitute for the virtual functions in CRfCommPort.

The *OnDataReceived()* function is called to pass an incoming data packet to the application.

The *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 DK 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()*.

4.2 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 hex 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 DK 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 DK. The new value may then be passed to the DK 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

// {CE37CA6E-288A-409a-9796-191882EE44FC}

static GUID <<name>> =

{ 0xce37ca6e, 0x288a, 0x409a, { 0x97, 0x96, 0x19, 0x18, 0x82, 0xee, 0x44, 0xfc } };

can be cut and pasted from the guidgen.exe output window. You just substitute your variable name for '<<name>>'

The DK 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 CBtIfClasses.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;
```

4.3 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. When side A, say, 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 occurs below the application level.

Applications can take the default or set a non-default 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 non-default MTU might be that the application writer knows that the hardware platform has very limited memory.

4.4 DESTRUCTORS FOR DK CLASSES

All of the DK classes have non-trivial destructors. Applications which instantiate DK classes, or application classes derived from DK classes, from the heap must delete these objects to prevent the application from leaking memory.

4.5 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.

4.6 EXAMPLES (RFCOMM)

4.6.1 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:
   - 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.
   - 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.
   - Processing is application dependent; data is sent using the *CRfCommPort::Write()* functions and the derived function *CRfCommPort::OnDataReceived()* is called when data is received.
   - 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.

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

5 Class Descriptions and Usage

5.1 CBtIf

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

An object of this class must be instantiated before any other DK 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).

This class defines pure virtual methods *CBtIf::OnDeviceResponded()* and *CBtIf::OnDiscoveryComplete()*. The application must provide a derived class that provides these functions.

This class also defines a virtual method, *CBtIf::OnInquiryComplete()*, that the application can define, when useful.

A set of standard GUID values (Table 2) for Bluetooth service classes is provided as public data members for this class.

Table 2: Standard GUIDs for Service Classes.

Standard GUIDs for Service Classes
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_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

5.1.1 StartInquiry()

Starts the Bluetooth device inquiry procedure.

Because the Bluetooth stack is multi-user, 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.

Table 3: StartInquiry()

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

5.1.2 StopInquiry()

Stops a running inquiry.

Table 4: StopInquiry()

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

5.1.3 pure virtual OnDeviceResponded()

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.

Table 5: pure virtual OnDeviceResponded()

pure virtual OnDeviceResponded()		

Prototype	`virtual void OnDeviceResponded (`
	`BD_ADDR bda,`
	`DEV_CLASS dev_class,`
	`BD_NAME bd_name,`
	`BOOL b_connected)= 0;`	

Parameters	bda	The address of the responding device.
	dev_class	The class of the responding device, see BtIfDefinitions.h.
	bd_name	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.
	b_connected	TRUE if the responding device is currently connected to the local device.
Returns	void	

5.1.4 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.

Table 6: virtual OnInquiryComplete()

virtual OnInquiryComplete()		

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

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

5.1.5 StartDiscovery()

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()*.

Table 7: StartDiscovery()

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.	

5.1.6 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.

Table 8:SwitchRole ()

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.

5.1.7 pure 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()*.

Table 9: pure virtual OnDiscoveryComplete()

pure virtual OnDiscoveryComplete()	
Prototype	`virtual void OnDiscoveryComplete () = 0;`

Parameters	None
Returns	void

5.1.8 ReadDiscoveryRecords()

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 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().*

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

Table 10: ReadDiscoveryRecords()

ReadDiscoveryRecords()		

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 thecapacity for *max_size* entries of class CsdpDiscoveryRec.
	p_guid_filter	An optional pointer to a GUID filter. If this is set, only record(s) which have a service class ID matching this GUID are returned (typically only one record will match).
Returns	The number of discovery records read.	

5.1.9 Bond()

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

A temporary connection is set up by the stack server to authenticate the remote device. If the authentication is successful, the devices are bonded.

Once bonded, future connections for services will automatically be authenticated by the stack server security logic, using the link key, without intervention by the application.

Table 11: Bond()

Bond()		
Prototype	`BOND_RETURN_CODE Bond(BD_ADDR bda, LPTSTR pin_code);`	
Parameters	bda	The BD address of the device to bond with
	pin_code	the PIN code to use for the bonding. This is an array of UINT8, null terminated. A code with the length >= PIN_CODE_LEN (=16) is invalid.
Returns	Values are: SUCCESS,	

	ALREADY_BONDED - BAD_PARAMETER – PIN code too long FAIL – if timeout or rejection by other device

5.1.10 BondQuery()

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

Table 12: BondQuery()

BondQuery()		
Prototype	`BOOL BondQuery(BD_ADDR bda);`	
Parameters	bda	The BD address of the device to bond with
Returns	TRUE if the devices are already paired; FALSE otherwise.	

5.1.11 UnBond()

Deletes the bond between the devices.

Table 13: UnBond()

BondQuery()		
Prototype	`BOOL BondQuery(BD_ADDR bda);`	
Parameters	bda	The BD address of the device to delete the bond with
Returns	TRUE if the devices are now unbonded; FALSE otherwise.	

5.2 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.

5.2.1 AssignPsmValue()

Used to assign a PSM value to the interface. This function 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.

Table 14: AssignPsmValue()

AssignPsmValue()		
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 non-zero 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. *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.*	

5.2.2 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()*

Table 15: Register()

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

5.2.3 Deregister()

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

Table 16: Deregister()

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

5.2.4 GetPsm()

Returns the PSM value assigned to the interface.

Table 17: GetPsm()

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

5.2.5 SetSecurityLevel()

Sets the security level for all connections. 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.

Table 18: SetSecurityLevel()

SetSecurityLevel()		

Prototype	`BOOL SetSecurityLevel (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, can not be set by it's self, has to use it with BTM_SEC_IN_AUTHENTICATE) • BTM_SEC_OUT_AUTHENTICATE (used by client side) • BTM_SEC_OUT_ENCRYPT (used by client side, can not be set by it's self, has to use it with BTM_SEC_OUT_AUTHENTICATE) See BtIfDefinitions.h. Note: When multiple values other than BTM_SEC_NONE are used, they must be bit-Ored in the security_level field. BTM_SEC_IN_ENCRYPT /BTM_SEC_OUT_EN
	is_server	TRUE if the local device is a server, FALSE if a client.
Returns	TRUE if operation was successful. FALSE otherwise.	

5.2.6 RegisterAppService ()

Allows the client side of an L2Cap connection to add the service to the Windows Registry. If the guid is already in the applications registry no further action is taken. If not, a new registry entry is created containing the service name and the guid, and the security parameters for an application.

This function must be called on the client side before discovery is initiated. Otherwise discovery will not report the service to the application.

Table 19: RegisterAppService ()

RegisterAppService ()		
Prototype	`BOOL RegisterAppService(char *p_service_name, GUID guid);`	
Parameters	p_service_name	The name of the service.
	GUID	The globally unique identifier for the service.
Returns	TRUE if operation was successful; FALSE if the new entry cannot be saved in the registry.	

5.3 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 must provide a derived class that defines real methods for these virtual methods.

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.

5.3.1 Listen()

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.

Table 20: Listen()

Listen()		
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. *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.*	

5.3.2 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 non-default MTU (Maximum Transmission Unit). The MTU sets an upper limit for a single packet at the Bluetooth protocol layer being used. When side A, say, 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 occurs 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.

Table 21: Accept()

Accept()

Prototype	`BOOL Accept (` `  UINT16 desired_mtu = L2CAP_DEFAULT_MTU);`	
Parameter	desired_mtu	An optional parameter that can be passed if the application wants a non-default MTU.
Returns	This method will fail (return FALSE) only if the connection object has not received an incoming connection indication.	

5.3.3 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.

Table 22: Reject()

Reject()		
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 in incoming connection indication.	

5.3.4 Connect()

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

The client may specify a non-default MTU (Maximum Transmission Unit). The MTU sets an upper limit for a single packet at the Bluetooth protocol layer being used. When side A, say, 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 occurs below the application level.

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

Table 23: Connect()

Connect()		

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 non-default MTU.

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

5.3.5 Reconfigure()

Either client or server applications call this function to reconfigure an existing connection.

Typically, this would be to change the MTU size The MTU sets an upper limit for a single packet at the Bluetooth protocol layer being used. When side A, say, 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 occurs below the application level.

Table 24: Reconfigure()

Reconfigure()		
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.	

5.3.6 Disconnect()

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

Table 25: Disconnect()

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

5.3.7 Write()

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.

Table 26: Write()

Write()		

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.	

5.3.8 GetConnectionStats()

Retrieves current connection statistics

5.3.8.1 Connection Statistics

These are defined in the structure tBT_CONN_STATS, in header file BtIfDefinitions.h:

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.
- 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

Table 27: GetConnectionStats ()

GetConnectionStats ()		
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.	

5.3.9 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.

Table 28:SwitchRole ()

SwitchRole ()	
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.

5.3.10 virtual OnIncomingConnection()

Server applications must 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.

`Table 29: virtual OnIncomingConnection()`

virtual OnIncomingConnection()

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

5.3.11 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.

Table 30: virtual OnConnectPendingReceived()

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

5.3.12 pure virtual OnConnected()

This is a pure virtual function; all applications must provide a derived method to process the connection.

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

Table 31: pure virtual OnConnected()

pure virtual OnConnected()	
Prototype	`virtual void OnConnected() = 0;`
Parameters	None
Returns	void

5.3.13 pure virtual OnDataReceived()

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

Table 32: pure virtual OnDataReceived()

pure virtual OnDataReceived()		
Prototype	`virtual void OnDataReceived (void *p_data, UINT16 length) = 0;`	
Parameters	p_data	A pointer to the received data.
	length	The length of the received data.
Returns	void	

5.3.14 pure virtual OnCongestionStatus()

This is a pure virtual function; all applications must provide a derived method to process changes in the congestion status.

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.

Table 33: pure virtual OnCongestionStatus()

pure virtual OnCongestionStatus()		
Prototype	`virtual void OnCongestionStatus ( BOOL is_congested) = 0;`	
Parameter	is_congested	Set TRUE if congested, else FALSE if not.
Returns	void	

5.3.15 pure virtual OnRemoteDisconnected()

This is a pure virtual function; all applications must provide a derived method to process disconnects from the remote device.

Table 34: pure virtual OnRemoteDisconnected()

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

5.4 CSDPSERVICE

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

All methods return an enumerated type *SDP_RETURN_CODE*.

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).

Table 35 SDP_RETURN_CODE

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	1 – Attempt to add a string or sequence attribute that is too large, greater than 255 characters. 2 – Attempt to add more than 32 attributes.

5.4.1 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.

Table 36: AddServiceClassIdList()

AddServiceClassIdList()		
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)
	p_service_guids	A pointer to a list of *num_guids* GUIDs.
Returns	See definition of SDP_RETURN_CODE above.	

5.4.2 AddServiceName()

Adds a service name attribute to the service record.

Table 37: AddServiceName()

AddServiceName()		
Prototype	`SDP_RETURN_CODE AddServiceName ( 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.
Returns	See definition of SDP_RETURN_CODE above.	

5.4.3 AddProfileDescriptorList()

Adds a profile descriptor list attribute to the service record.

Table 38: AddProfileDescriptorList()

AddProfileDescriptorList()		
Prototype	`SDP_RETURN_CODE AddProfileDescriptorList ( GUID *p_profile_guid, UINT16 version);`	
Parameters	p_profile_guid	A pointer to the GUID of the profile.
	version	The profile version.
Returns	See definition of SDP_RETURN_CODE above.	

5.4.4 AddL2CapProtocolDescriptor()

Adds a protocol descriptor list attribute to the service record for an L2CAP-based service.

Table 39: AddL2CAPProtocolDescriptor()

AddL2CapProtocolDescriptor()		
Prototype	`SDP_RETURN_CODE AddL2CapProtocolDescriptor ( UINT16 psm);`	
Parameters	psm	The PSM the service is using.
Returns	See definition of SDP_RETURN_CODE above.	

5.4.5 AddRFCommProtocolDescriptor()

Adds a protocol descriptor list attribute to the service record for an RFCOMM-based service.

Table 40: AddRFCommProtocolDescriptor()

AddRFCommProtocolDescriptor()		
Prototype	`SDP_RETURN_CODE AddRFCommProtocolDescriptor ( UINT8 scn);`	
Parameters	scn	The RFCOMM SCN for the service.
Returns	See definition of SDP_RETURN_CODE above.	

5.4.6 AddProtocolList()

Adds a protocol descriptor list attribute to the service record. This function should only be needed if the L2CAP or RFCOMM specific 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;
```

Table 41: AddProtocolList()

AddProtocolList()		
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.
	p_elem_list	A pointer to the protocol list to add.
Returns	See definition of SDP_RETURN_CODE above.	

5.4.7 AddLanguageBaseAttrIDList()

Add a language-base attribute to the service record.

Table 42: AddLanguageBaseAttrIDList()

AddLanguageBaseAttrIDList()		
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 definition of SDP_RETURN_CODE above.	

5.4.8 MakePublicBrowseable()

Adds the browse group list attribute to the record, with the "public browse root (0x1002)" UUID.

Table 43: MakePublicBrowseable()

MakePublicBrowseable()	
Prototype	`SDP_RETURN_CODE MakePublicBrowseable (void);`
Parameter	None
Returns	See definition of SDP_RETURN_CODE above.

5.4.9 SetAvailability()

Sets the availability attribute in the service record.

Table 44: SetAvailability()

SetAvailability()		
Prototype	`SDP_RETURN_CODE SetAvailability (UINT8 availability);`	
Parameters	availability	The service availability: 0 = unavailable 255 = fully available.
Returns	See definition of SDP_RETURN_CODE above.	

5.4.10 AddAttribute()

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 the table of data types below.

Note that numeric data types, UINT_DESC_TYPE and TWO_COMP_INT_DESC_TYPE , should be in Big Endian order. So, 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.

Table 45: AddAttribute()

AddAttribute()		

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 below and BtIfDefinitions.h. Note: The value ATTR_ID_SERVICE_CLASS_ID_LIST may not be used with this function. There is a special DK 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.
	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 below and BtIfDefinitions.h.
	attr_len	The length of the attribute.
	p_val	A pointer to the attribute value.
Returns	See definition of SDP_RETURN_CODE above.	

Table 46 Service Record Attribute IDs

Attribute Code - See the Bluetooth Core Spec 1.1, Part E Section 5.1 for precise definitions of the service attribute codes
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_ICON10
ATTR_ID_ICON_URL
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_SUPPORTED_FORMATS_LIST
ATTR_ID_FAX_CLASS_2_0_SUPPORT
ATTR_ID_FAX_CLASS_2_SUPPORT
ATTR_ID_AUDIO_FEEDBACK_SUPPORT

Table 47 Service Record Attribute Types

Attribute Type	
Attribute Type Code	**Meaning**
NULL_DESC_TYPE	Null
UINT_DESC_TYPE	Unsigned Integer
TWO_COMP_INT_DESC_TYPE	Signed 2's 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

5.4.11 DeleteAttribute()

Deletes an attribute from the service record.

Table 48: DeleteAttribute()

DeleteAttribute()		
Prototype	`SDP_RETURN_CODE DeleteAttribute (UINT16 attr_id);`	
Parameters	attr_id	The ID of the attribute.
Returns	*SDP_OK*	Everything OK.
	SDP_INVALID_PARAMETERS	The attribute ID was not found.
	SDP_INVALID_RECORD	The record was not created or did not have a service class ID list attribute.

5.5 CSDPDISCOVERYREC

When an application reads the services from a remote device, 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 'char'
- *m_service_guid* is the service GUID, the service GUID, of type 'GUID'

5.5.1 FindRFCommScn()

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

Table 49: FindRFCommScn()

FindRFCommScn()		
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.	

5.5.2 FindL2CapPsm()

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

Table 50: FindL2CAPPsm()

FindL2CapPsm()		
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.	

5.5.3 FindProtocolListElem()

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.

Table 51: FindProtocolListElem()

FindProtocolListElem()		

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.
Returns	TRUE if the list element was found. FALSE otherwise.	

5.5.4 FindProfileVersion()

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

Table 52: FindProfileVersion()

FindProfileVersion()		
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.	

5.5.5 FindAttribute()

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.

Table 53: FindAttribute()

FindAttribute()		
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 (=256) bytes long. Nested sequences up to 5 levels deep are supported.
Returns	TRUE if the attribute was found. FALSE otherwise.	

5.6 CRFCOMMIF

Controls Service Channel Number (SCN) allocation for the DK RFCOMM applications.

5.6.1 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 when the CL2CommIf object destructor is executed.

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

Table 54: AssignScnValue()

AssignScnValue()		
Prototype	`BOOL AssignScnValue (GUID *p_service_guid,` `UINT8 scn = 0);`	
Parameters	p_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.
Returns	TRUE if a SCN was assigned. FALSE otherwise. *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.*	

5.6.2 GetScn()

Returns the SCN value in use.

Table 55: GetScn()

GetScn()	
Prototype	`UINT8 GetScn();`
Parameters	None
Returns	The SCN assigned to the interface. Zero (0) if an SCN is not assigned).

5.6.3 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.

Table 56: SetSecurityLevel()

SetSecurityLevel()		

Prototype	`BOOL SetSecurityLevel (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 client side) • BTM_SEC_OUT_AUTHENTICATE (used by client side) • BTM_SEC_OUT_ENCRYPT (used by client side) 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.	

5.6.4 RegisterAppService ()

Allows the client side of an RFComm connection to add the service to the Windows Registry. If the guid is already in the applications registry no further action is taken. If not, a new registry entry is created containing the service name and the guid, and the security parameters for an application.

This function must be called on the client side before discovery is initiated. Otherwise discovery will not report the service to the application.

Table 57: RegisterAppService ()

RegisterAppService ()		
Prototype	`BOOL RegisterAppService(char *p_service_name, GUID guid);`	
Parameters	p_service_name	The name of the service.
	GUID	The globally unique identifier for the service.
Returns	TRUE if operation was successful; FALSE if the new entry cannot be saved in the registry.	

5.7 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 must provide a derived class that defines real methods for these virtual methods.

Most methods return an enumerated type *PORT_RETURN_CODE*.

5.7.1 PORT_RETURN_CODE

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

Table 58: PORT_RETURN_CODE

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.
NOT_OPENED	The function was called before the connection opened or after it closed.
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.
UNKNOWN_ERROR	Any condition other than the above.

5.7.2 OpenServer()

Opens a server connection for an RFCOMM serial port and listens for a connection attempt.

Table 59: OpenServer()

OpenServer()		

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 non-default MTU.
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.	

5.7.3 OpenClient ()

Opens a client connection for an RFCOMM serial port and initiates the connection.

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

Table 60: OpenClient ()

OpenClient ()		

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

Parameters	scn	The Service Channel Number obtained from the service discovery procedure.
	remote_bd_addr	The BD address of the server, obtained from the service discovery procedure.
	desired_mtu	Optional: can be passed if the application wants a non-default MTU.
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.	

5.7.4 Close()

Closes the connection

Table 61: Close()

Close()	
Prototype	`PORT_RETURN_CODE Close(void)`
Parameters	void
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.

5.7.5 SetFlowEnabled()

Enables/disables flow control. Signals the remote device to stop/resume sending data packets.

Table 62: SetFlowEnabled()

SetFlowEnabled()		
Prototype	`PORT_RETURN_CODE SetFlowEnabled` `(BOOL enabled)`	
Parameters	enabled	Set TRUE/FALSE to enable/disable incoming data packets.
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.	

5.7.6 IsConnected()

Tests to see if the connection has been established.

Table 63: IsConnected()

IsConnected()		
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.	

5.7.7 SetModemSignal ()

Sets the modem control signals.

Table 64: SetModemSignal ()

SetModemSignal ()

Prototype	`PORT_RETURN_CODE SetModemSignal ( UINT8 signal)`	
Parameters	signal	One of the following defined values from BtIfDefinitions.h: • PORT_SET_DTRDSR • PORT_CLR_DTRDSR • PORT_SET_CTSRTS • PORT_CLR_CTSRTS • PORT_SET_RI PORT_CLR_RI • PORT_SET_DCD • PORT_CLR_DCD • PORT_SET_BREAK • PORT_CLR_BREAK
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.	

5.7.8 GetModemStatus()

Reads the modem status.

Table 65: GetModemStatus()

GetModemStatus()		
Prototype	`PORT_RETURN_CODE GetModemStatus ( UINT8 *p_signal)`	
Parameters	signal	One or more of the following flags from BtIfDefinitions.h: • PORT_DTRDSR_ON • PORT_CTSRTS_ON PORT_RING_ON • PORT_DCD_ON.
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.	

5.7.9 SendError()

Sends a specific error code to the remote device.

Table 66: SendError()

SendError()		
Prototype	`PORT_RETURN_CODE SendError ( UINT8 errors)`	
Parameters	errors	One or more of the following flags from BtIfDefinitions.h: • PORT_ERR_BREAK • PORT_ERR_OVERRUN • PORT_ERR_FRAME • PORT_ERR_RXOVER • PORT_ERR_TXFULL.
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.	

5.7.10 Purge()

Discards all the data from the output or input queues of the specified connection.

Table 67: Purge()

Purge()		
Prototype	`PORT_RETURN_CODE Purge (` `UINT8 purge_flags)`	
Parameters	purge_flags	One or both of the following flags from BtIfDefinitions.h: • PORT_PURGE_TXCLEAR • PORT_PURGE_RXCLEAR.
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.	

5.7.11 Write()

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

Table 68: Write()

Write()		

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.
	len_to_write	The number of characters to write.
	p_len_written	The number of characters actually written, returned to the caller.
Returns	SUCCESS Everything OK. Otherwise, see definition of PORT_RETURN_CODE above.	

5.7.12 GetConnectionStats()

Retrieves current connection statistics

5.7.12.1 Connection Statistics

These are defined in the structure tBT_CONN_STATS, in header file BtIfDefinitions.h:

```
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.
- 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

Table 69: GetConnectionStats ()

GetConnectionStats ()		
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
Returns	CRfCommPort::NOT_OPENED if a connection attempt has not even been initiated. CRfCommPort::SUCCESS otherwise.	

5.7.13 pure virtual OnDataReceived()

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

Table 70: pure virtual OnDataReceived()

pure virtual OnDataReceived()		
Prototype	`virtual void OnDataReceived (void *p_data, UINT16 len) = 0;`	
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	

5.7.14 virtual OnEventReceived()

The client and server must 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.

Table 71: virtual OnEventReceived()

virtual OnEventReceived()		
Prototype	`virtual void OnEventReceived (UINT32 event_code);`	
Parameters	event_code	One of the port events from BtIfDefinitions.h. • PORT_EV_RXFLAG • PORT_EV_TXEMPTY • PORT_EV_CTS • PORT_EV_DSR • PORT_EV_RLSD • PORT_EV_BREAK • PORT_EV_ERR • PORT_EV_RING • PORT_EV_CTSS • PORT_EV_DSRS • PORT_EV_RLSDS • PORT_EV_OVERRUN • PORT_EV_TXCHAR

		• PORT_EV_CONNECTED • PORT_EV_CONNECT_ERR • PORT_EV_FC • PORT_EV_FCS.
Returns	Void	

5.8 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.

5.8.1 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 72: FTP_RETURN_CODE

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.
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.
UNKNOWN_RETURN_ERROR	Any condition other than the above.

5.8.2 OpenConnection()

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

Table 73: OpenConnection()

OpenConnection()		
Prototype	`FTP_RETURN_CODE OpenConnection (BD_ADDR bda, CSdpDiscoveryRec & sdp_rec);`	
Parameters	bda	The FTP server's BT device address, from Service Discovery process.
	sdp_rec	The service discovery record.
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.	

5.8.3 CloseConnection()

Closes the connection. The *OnCloseResponse()* function will be called with the server's response.

Table 74: CloseConnection()

CloseConnection()	
Prototype	`FTP_RETURN_CODE CFtpClient:: CloseConnection();`
Parameters	None
Returns	SUCCESS Everything OK. Otherwise, see the definition of FTP_RETURN_CODE.

5.8.4 PutFile()

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

Table 75: PutFile()

PutFile()		
Prototype	`FTP_RETURN_CODE CFtpClient:: PutFile( WCHAR * local_file_name);`	
Parameters	local_file_name	The name of the file to be sent to server's current folder.
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.	

5.8.5 GetFile()

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

Table 76: GetFile()

GetFile()		
Prototype	`FTP_RETURN_CODE CFtpClient:: GetFile( WCHAR * remote_file_name, WCHAR * local_folder);`	
Parameters	remote_file_name	The name of the source file on the server.
	local_folder	The name of the local folder on the client.
Returns	SUCCESS Everything OK. Otherwise, see the definition of FTP_RETURN_CODE.	

5.8.6 FolderListing()

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.

Table 77: FolderListing()

FolderListing()	
Prototype	`FTP_RETURN_CODE CFtpClient:: FolderListing();`
Parameters	None
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.

5.8.7 ChangeFolder()

Requests that the server make a sub-folder that is in the current folder the new 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.

Table 78: ChangeFolder()

ChangeFolder()		
Prototype	`FTP_RETURN_CODE CFtpClient:: ChangeFolder( WCHAR *sz_folder);`	
Parameters	sz_folder	The name of the new server folder.
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.	

5.8.8 DeleteFile()

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.

Table 79: DeleteFile()

DeleteFile()		
Prototype	`FTP_RETURN_CODE CFtpClient:: DeleteFile( WCHAR *sz_file);`	
Parameters	sz_file	The name of the file or folder to be deleted.
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.	

5.8.9 Abort()

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

Table 80: Abort()

Abort()	
Prototype	`FTP_RETURN_CODE CFtpClient:: Abort();`
Parameters	None
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.

5.8.10 Parent()

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.

Table 81: Parent()

Parent()	
Prototype	`FTP_RETURN_CODE CftpClient:: Parent();`
Parameters	Void
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.

5.8.11 Root()

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.

Table 82: Root()

Root()	
Prototype	`FTP_RETURN_CODE CftpClient:: Root();`
Parameters	Void
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.

5.8.12 CreateEmpty()

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.

Table 83: CreateEmpty()

CreateEmpty()		
Prototype	`FTP_RETURN_CODE CFtpClient:: CreateEmpty( WCHAR *sz_file );`	
Parameters	sz_file	The name of the file to be created.
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.	

5.8.13 CreateFolder()

Requests that the server create a new folder, as a sub folder 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.

Table 84: CreateFolder()

CreateFolder()		
Prototype	`FTP_RETURN_CODE CFtpClient:: CreateFolder( WCHAR *sz_folder );`	
Parameters	sz_folder	The name of the new folder.
Returns	SUCCESS Everything OK. Otherwise, see definition of FTP_RETURN_CODE.	

5.8.14 SetSecurity()

Sets authentication and encryption parameters for OBEX FTP connections.

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 Configuration tab in the BTTray application.

Table 85: SetSecurity()

SetSecurity()		
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	

5.8.15 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 86: FTP_RESULT_CODE

FTP_RESULT_CODE	
FTP_RESULT_CODE Value	**Meaning**
COMPLETED	Operation completed without error.
BAD_ADDR	Bad Bluetooth device address.
BAD_STATE	Could not handle the request in present state.
FILE_EXISTS	File already exists.
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_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.
NOT_INITIALIZED	Not initialized.
PARAM	Bad parameter.
PERMISSIONS	Prohibited by file permissions.
SHARING	File is shared.
RESOURCES	File system resource limit has been reached - may be file handles, disk space, etc.
UNKNOWN_RESULT_ERROR	Any condition other than the above.

5.8.16 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.

Table 87: virtual OnProgress()

virtual void OnProgress()		
Prototype	`virtual void OnProgress(FTP_RESULT_CODE result_code, WCHAR * name, long current, long total);`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
	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, the file length.
Returns	void	

5.8.17 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, DK provides default response handlers, which take no action.

Table 88: virtual OnOpenResponse()

virtual void OnOpenResponse()		
Prototype	`virtual void OnOpenResponse( FTP_RESULT_CODE result_code);`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
Returns	void	

5.8.18 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, DK provides default response handlers, which take no action.

Table 89: virtual OnCloseResponse()

virtual void OnCloseResponse()		
Prototype	`virtual void OnCloseResponse( FTP_RESULT_CODE result_code);`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
Returns	void	

5.8.19 virtual OnPutResponse()

If the "put" function is used client applications must 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.

Table 90: virtual OnPutResponse()

virtual void OnPutResponse()		
Prototype	`virtual void OnPutResponse( FTP_RESULT_CODE result_code, WCHAR * name);`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
	name	The file name.
Returns	void	

5.8.20 virtual OnGetResponse()

If the "get" function is used client applications must 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.

Table 91: virtual OnGetResponse()

virtual void OnGetResponse()		
Prototype	`virtual void OnGetResponse( FTP_RESULT_CODE result_code, WCHAR * name);`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
	name	The file name.
Returns	void	

5.8.21 virtual OnCreateResponse()

If one of the create functions (*CreateEmpty* or *CreateFolder*) is to be used, the client application must 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.

Table 92: virtual OnCreateResponse()

virtual void OnCreateResponse()		
Prototype	`virtual void OnCreateResponse( FTP_RESULT_CODE result_code, WCHAR * name);`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
	name	The created file name, in response to a CreateEmpty call; or an empty string in response to a CreateFolder call.
Returns	void	

5.8.22 virtual OnDeleteResponse()

If the delete function DeleteFile is to be used, the client application must 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.

Table 93: virtual OnDeleteResponse()

virtual void OnDeleteResponse()		
Prototype	`virtual void OnDeleteResponse( FTP_RESULT_CODE result_code, WCHAR * name);`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
	name	The file name or the folder name deleted
Returns	void	

5.8.23 virtual OnChangeFolderResponse()

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

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

Table 94: virtual OnChangeFolderResponse()

virtual void OnChangeFolderResponse()		
Prototype	`virtual void OnChangeFolderResponse( FTP_RESULT_CODE result_code, tFtpFolder folder_type, WCHAR * folder_name) ;`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
	folder_type	A value from enumerated type tFtpFolder – one of the values: • FTP_ROOT_FOLDER • FTP_PARENT_FOLDER • FTP_SUBFOLDER, in BtIfClasses.h.
	folder_name	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	

5.8.24 virtual OnFolderListingResponse()

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 must 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 folderlisting 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
- Date last modified
- File size
- Permissions characters.

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

Table 95: virtual OnFolderListingResponse()

virtual void OnFolderListingResponse()		

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

Parameters	result_code	See the definition of FTP_RESULT_CODE.
	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 if entries (files/folders) in the array.
Returns	Void	

5.8.25 virtual OnXmlFolderListingResponse()

If the FolderListing() function is to be used, the client application must 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 folderlisting 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 with the following contents:

```
<?xml version="1.0"?>
<!DOCTYPE folder-listing SYSTEM "obex-folder-listing.dtd">
<folder-listing version="1.0">
<file name="BT DK 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>
```

Table 96: virtual OnXmlFolderListingResponse()

virtual void OnXmlFolderListingResponse()	
Prototype	`void OnXmlFolderListingResponse(`

		`FTP_RESULT_CODE result_code,`
		`WCHAR *pfolder_listing,`
		`long folder_length );`

Parameters	result_code	See the definition of FTP_RESULT_CODE.
	p_folder_listing	A pointer to a NULL-terminated character string, formatted in XML, which contains the folder listing information. See BtIfDefinitions.h.
	folder_length	The number of characters in the string.
Returns	void	

5.8.26 virtual OnAbortResponse()

If the “abort” function is used client applications must 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.

Table 97: virtual OnAbortResponse()

virtual void OnAbortResponse()		
Prototype	`virtual void OnAbortResponse(` `FTP_RESULT_CODE result_code);`	
Parameters	result_code	See the definition of FTP_RESULT_CODE.
Returns	void	

5.9 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.

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' box must also be checked.

5.9.1 OPP_RETURN_CODE

A common set of return codes is provided by the direct OPP function calls, push, pull, exchange, and abort. Use enumerated type OPP_RETURN_CODE, from BtIfClasses.h.

Table 98: OPP_RETURN_CODE

OPP_RETURN_CODE	
OPP_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.
OPP_ERROR	OPP-specific error.
ABORT_INVALID	Abort not valid, no operation is in progress.
UNKNOWN_ERROR	Any condition other than the above.

5.9.2 Push()

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

Table 99: Push()

Push()		

Prototype	`OPP_RETURN_CODE COppClient:: Push(`
	`BD_ADDR bda,`
	`WCHAR *psz_path_name,`
	`CsdpDiscoveryRec & sdp_rec);`	

Parameters	bda	The Bluetooth device address of the server.
	psz_path_name	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.
	sdp_rec	The OPP Service discovery record obtained from the OPP server.
Returns	*OPP_RETURN_SUCCESS* Everything OK. Otherwise, see definition of OPP_RETURN_CODE.	

5.9.3 Pull()

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.

Table 100: Pull()

Pull()		

Prototype	`OPP_RETURN_CODE COppClient::Pull(`
	`BD_ADDR bda,`
	`WCHAR *psz_folder_name,`
	`CsdpDiscoveryRec & sdp_rec);`	

Parameters	bda	The Bluetooth device address of the server.
	psz_folder_name	The local folder to receive the object. The must be an absolute path name.
	sdp_rec	The OPP Service discovery record obtained from the OPP server.
Returns	*OPP_RETURN_SUCCESS* Everything OK. Otherwise, see definition of OPP_RETURN_CODE.	

5.9.4 Exchange()

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

Table 101: Exchange()

Exchange()		

Prototype	`OPP_RETURN_CODE COppClient::Exchange(`
	`BD_ADDR bda,`
	`WCHAR *psz_name,`
	`WCHAR *psz_folder,`
	`CSdpDiscoveryRec & sdp_rec);`	

Parameters	bda	The Bluetooth device address of the server.
	psz_name	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.
	psz_folder	The client folder to receive the server's business card—a null terminated string specifying an absolute path..
	sdp_rec	The OPP Service discovery record obtained from the OPP server.
Returns	*OPP_RETURN_SUCCESS* Everything OK. Otherwise, see definition of OPP_RETURN_CODE.	

5.9.5 Abort()

Requests that the current operation be aborted.

Table 102: Abort()

Abort()	
Prototype	`OPP_RETURN_CODE COppClient::Abort();`
Parameters	None
Returns	*OPP_RETURN_SUCCESS* Everything OK. Otherwise, see definition of OPP_RETURN_CODE.

5.9.6 SetSecurity()

Sets the authentication and encryption parameters for OBEX OPP operations.

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

Table 103: SetSecurity()

SetSecurity()		
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	

5.9.7 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 104: OPP_RESULT_CODE

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 thepresent 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 theconnection.
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.

5.9.8 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.

Table 105: virtual OnProgress()

virtual void OnProgress()		

Prototype	`virtual void OnProgress(BD_ADDR bda,`
	`WCHAR * name,`
	`OPP_RESULT_CODE result_code,`
	`long current,`
	`long total);`	

Parameters	bda	The Bluetooth device address of the server.
	name	The file/folder name being transferred.
	result_code.	See the definition of OPP_RESULT_CODE.
	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	

5.9.9 virtual OnPushResponse()

If the "push" function is used client applications must 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.

Table 106: virtual OnPushResponse()

virtual void OnPushResponse()		

Prototype	`virtual void OnPushResponse(`
	`OPP_RESULT_CODE result_code,`
	`BD_ADDR bda,`
	`WCHAR * file_name);`	

Parameters	result_code	See the definition of OPP_RESULT_CODE.
	bda	The address of the remote device.
	file_name	The name of the file being pushed.
Returns	void	

5.9.10 virtual OnPullResponse()

If the "pull" function is used client applications must 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.

Table 107: virtual OnPullResponse()

virtual void OnPullResponse()		

Prototype	`virtual void OnPullResponse(`
	`OPP_RESULT_CODE result_code,`
	`BD_ADDR bda,`
	`WCHAR * file_name);`	

Parameters	result_code	See the definition of OPP_RESULT_CODE.

	bda	The address of the remote device.
	file_name	The name of the file being pulled.
Returns	void	

5.9.11 virtual OnExchangeResponse()

If the “exchange” function is used client applications must 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.

Table 108: virtual OnExchangeResponse()

virtual void OnExchangeResponse()		

Prototype	`virtual void OnExchangeResponse(`
	`  OPP_RESULT_CODE result_code,`
	`  BD_ADDR bda,`
	`  WCHAR * file_name);`	

Parameters	result_code	See the definition of OPP_RESULT_CODE.
	bda	The address of the remote device.
	file_name	The name of the file being transferred to server.
Returns	void	

5.9.12 virtual OnAbortResponse()

When the “abort” function is used client applications must 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.

Table 109: virtual OnAbortResponse()

virtual void OnAbortResponse()		
Prototype	`virtual void OnAbortResponse (` `  OPP_RESULT_CODE result_code);`	
Parameters	result_code	See the definition of OPP_RESULT_CODE.
Returns	void	

5.10 CLAPCLIENT

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 then provides the create connection, remove connection, and state change event functions.

5.10.1 LAP_RETURN_CODE

A common set of return codes is provided by the LAP function calls. Use enumerated type LAP_RETURN_CODE, from BtIfClasses.h

Table 110: LAP_RETURN_CODE

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.
UNKNOWN_ERROR	Any condition other than the above.

5.10.2 CreateConnection()

Requests a LAP connection to the server. Before calling this function the application must use a CBtIf class to locate a LAN using PPP service.

Table 111: CreateConnection()

CreateConnection()		

Prototype	`LAP_RETURN_CODE CreateConnection(`
	`BD_ADDR bda,`
	`CSdpDiscoveryRec & sdp_rec);`	

Parameters	bda	The server's Bluetooth device address
	sdp_rec	The service discovery record obtained for LAP server.
Returns	SUCCESS Everything OK. Otherwise, see definition of LAP_RETURN_CODE.	

5.10.3 RemoveConnection()

Closes the connection.

Table 112: RemoveConnection()

RemoveConnection()	
Prototype	`LAP_RETURN_CODE RemoveConnection();`
Parameters	None
Returns	SUCCESS Everything OK. Otherwise, see definition of LAP_RETURN_CODE.

5.10.4 SetSecurity()

Sets the authentication and encryption parameters for LAP, via PPP operations.

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

Table 113: SetSecurity()

SetSecurity()		
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	

5.10.5 pure virtual OnStateChange()

Allows the application to detect when a connection is established or cleared to the LAP server.

Table 114: pure virtual OnStateChange()

pure virtual void OnStateChange()		

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 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	

5.11 CDUNCLIENT

Allows the application to establish IP access, via a DialUp Network 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.

5.11.1 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 115:DUN_RETURN_CODE

DUN_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.
UNKNOWN_ERROR	Any condition other than the above.

5.11.2 CreateConnection()

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.

Table 116: CreateConnection()

CreateConnection()		
Prototype	`DUN_RETURN_CODE CreateConnection( BD_ADDR bda, CSdpDiscoveryRec & sdp_rec);`	
Parameters	bda	The server's Bluetooth device address
	sdp_rec	The service discovery record obtained for DUN server.
Returns	SUCCESS Everything OK. Otherwise, see definition of DUN_RETURN_CODE.	

5.11.3 RemoveConnection()

Closes the connection.

Table 117: RemoveConnection()

RemoveConnection()	
Prototype	`DUN_RETURN_CODE RemoveConnection();`
Parameters	None
Returns	SUCCESS Everything OK. Otherwise, see definition of DUN_RETURN_CODE.

5.11.4 SetSecurity()

Sets the authentication and encryption parameters for DUN, via DialUp Network operations.

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

Table 118: SetSecurity()

SetSecurity()		
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	

5.11.5 pure virtual OnStateChange()

Allows the application to detect when a connection is established or cleared to the DUN server.

Table 119: pure virtual OnStateChange()

pure virtual void OnStateChange()		

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 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	

5.12 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.

5.12.1 Configuration Notes

In this release of the DK a COM port is pre-defined for SPP application "Generic Serial". The user may select "Generic Serial" but if the application requires a different service name an additional serial port must be defined manually using the Bluetooth Neighborhood configuration function in the "Applications" tab.

Future DK releases will provide a CSppClient method to allow the application to assign a new COM port service programmatically.

5.12.2 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 120: SPP_CLIENT_RETURN_CODE

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.
UNKNOWN_ERROR	Any condition other than the above.

5.12.3 CreateConnection()

Requests an SPP connection to the server. Before calling this function the application uses DK class CBtIf to locate an SPP server that offers the required service.

Table 121: CreateConnection()

CreateConnection()		

Prototype	`SPP_CLIENT_RETURN_CODE CreateConnection(`
	`  BD_ADDR  bda,`
	`  LPCTSTR  szServiceName);`	

Parameters	Bda	The server's Bluetooth device address

	SzServiceName	The service name from the discovery record obtained for the SPP server.
Returns	SUCCESS Everything OK. Otherwise, see definition of SPP_CLIENT_RETURN_CODE.	

5.12.4 RemoveConnection()

Closes the connection, dissociating the COM port from the client application. This operation is required to free the port for other applications.

Table 122: RemoveConnection()

RemoveConnection()	
Prototype	`SPP_CLIENT_RETURN_CODE RemoveConnection();`
Parameters	None
Returns	SUCCESS Everything OK. Otherwise, see definition of SPP_CLIENT_RETURN_CODE.

5.12.5 pure virtual OnClientStateChange()

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 received the COM port value should be used by the client application to open the Windows COM port.

Table 123: pure virtual OnClientStateChange()

pure virtual void OnClientStateChange()		

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 the 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 or SPP_DISCONNECTED.
Returns	Void	

5.13 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.

5.13.1 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 create 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 DK releases will provide a CSppServer method to allow the application to assign a new COM port service programmatically.

Whether the predefined "Bluetooth Serial Port" or a new service name is used, the service must be configured as "Manual". This is done by clearing the "Startup" box for the service in the Bluetooth Neighborhood configuration function. If Bluetooth Neighborhood shows that the service is started, the service must be stopped before *CreateConnection() can be called.*

5.13.2 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 124: SPP_SERVER_RETURN_CODE

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_SUPPORTED	The service name was not defined in the "Local Services" list or the service was defined but was not "Manual".
NOT_ENOUGH_MEMORY	Local processor could not allocate memory for open.
UNKNOWN_ERROR	Any condition other than the above.

5.13.3 CreateConnection()

Checks Windows registry for the named service. If the service exists and is configured as "manual" in the Bluetooth Neighborhood, 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.

If the service is configured as "automatic" in the Bluetooth Neighborhood, a NOT_SUPPORTED error is returned.

Table 125: CreateConnection()

CreateConnection()		
Prototype	`SPP_SERVER_RETURN_CODE CreateConnection( LPCTSTR szServiceName);`	
Parameters	SzServiceName	The service name to be used for this application.
Returns	SUCCESS Everything OK. Otherwise, see definition of SPP_SERVER_RETURN_CODE.	

5.13.4 RemoveConnection()

Closes the connection, dissociating the COM port from the server application. This operation is required to free the port for other applications

Table 126: RemoveConnection()

RemoveConnection()	
Prototype	`SPP_SERVER_RETURN_CODE RemoveConnection();`
Parameters	None
Returns	SUCCESS Everything OK. Otherwise, see definition of SPP_SERVER_RETURN_CODE.

5.13.5 pure virtual OnServerStateChange()

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.

Table 127: pure virtual OnServerStateChange()

pure virtual void OnServerStateChange()		
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 BtIfDefinitions.h.
	Com_port	The local COM port assigned to the service.
	State	The new state: SPP_CONNECTED Or SPP_DISCONNECTED.
Returns	Void	

5.14 CLASS 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.

5.14.1 CObexHeaders Default Constructor

The default constructor creates an OBEX headers object with all header s null and with OBEX object length 0.

Table 128: CObexHeaders ()

CObexHeaders ()	
Prototype	CObexHeaders ();
Parameters	None
Returns	-

5.14.2 SetCount()

Sets the count header value. This is the number of objects to be sent.

Table 129: SetCount ()

SetCount ()	
Prototype	Void SetCount(UINT32 count);
Parameters	*count* – number of objects to be sent
Returns	None

5.14.3 DeleteCount()

Deletes the header.

Table 130: DeleteCount ()

DeleteCount ()	
Prototype	void DeleteCount();
Parameters	None
Returns	Void

5.14.4 GetCount()

Returns the header value.

Table 131: GetCount ()

GetCount ()	
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

5.14.5 SetName()

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.

Table 132: SetName ()

SetName ()	
Prototype	BOOL SetName (char * 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.

5.14.6 DeleteName()

Deletes the header.

Table 133: DeleteName ()

DeleteName ()	
Prototype	void DeleteName();
Parameters	None
Returns	void

5.14.7 GetNameLength()

Tests if the header is present. If so the header length is provided. . This value includes the null terminator.

Table 134: GetNameLength ()

GetNameLength ()	
Prototype	BOOL GetNameLength (UINT32 * p_len_incl_null);
Parameters	*p_len_incl_null* – When the header is present, used to provide number of characters to allow for the header value, including the null terminator character
Returns	TRUE if the header is present, and the header length is provided. Otherwise FALSE.

5.14.8 GetName()

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.

Table 135: GetName ()

GetName ()	
Prototype	BOOL GetName (char * 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.

5.14.9 SetType()

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.

Table 136: SetType ()

SetType ()

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.

5.14.10 DeleteType()

Deletes the header.

Table 137: DeleteType ()

DeleteType ()	
Prototype	void DeleteType();
Parameters	None
Returns	void

5.14.11 GetTypeLength()

Tests if the header is present. If so the header length is provided.

Table 138: GetTypeLength ()

GetTypeLength ()	
Prototype	BOOL GetTypeLength (UINT32 * p_length);
Parameters	*p_length* – When the header is present, used to provide number of characters to allow for the header value
Returns	TRUE if the header is present, and the header length is provided. Otherwise FALSE.

5.14.12 GetType()

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.

Table 139: GetType ()

GetType ()	
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.

5.14.13 SetLength()

Sets the header value. This is the length of the OBEX object..

Table 140: SetLength ()

SetLength ()	
Prototype	void SetLength (UINT32 length);
Parameters	*length* – length in bytes of the OBEX object
Returns	None

5.14.14 DeleteLength()

Deletes the header.

Table 138: DeleteLength ()

DeleteLength ()	
Prototype	void DeleteLength ();
Parameters	None

Returns	void

5.14.15 GetLength()

Returns the header value.

Table 141: GetLength ()

GetLength ()	
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

5.14.16 SetTime()

Sets the OBEX header parameter - time.

Table 142: SetTime ()

SetTime ()	
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

5.14.17 DeleteTime()

Deletes the header.

Table 143: DeleteTime ()

DeleteTime ()	
Prototype	void DeleteTime ();
Parameters	None
Returns	void

5.14.18 GetTime()

Returns the OBEX header parameter – length of object.

Table 144: GetTime ()

GetTime ()	
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

5.14.19 SetDescription()

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. For this case; and the internal length is set to 0.

Table 145: SetDescription ()

SetDescription ()	
Prototype	BOOL SetDescription (UINT8 * 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.

5.14.20 DeleteDescription()

Deletes the header.

Table 146: DeleteDescription ()

DeleteDescription ()	
Prototype	void DeleteDescription();
Parameters	None
Returns	void

5.14.21 GetDescriptionLength()

Tests if the header is present. If so the header length is provided. This value includes the null terminator.

Table 147: GetDescriptionLength ()

GetDescriptionLength ()	
Prototype	BOOL GetDescriptionLength (UINT32 * p_length);
Parameters	*p_len_incl_null* – When the header is present, used to provide number of characters to allow for the header value, including the null terminator character
Returns	TRUE if the header is present, and the header length is provided. Otherwise FALSE.

5.14.22 GetDescription()

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.

Table 148: GetDescription ()

GetDescription ()	
Prototype	BOOL GetDescription (char * 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.

5.14.23 AddTarget()

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.

Table 149: AddTarget ()

AddTarget ()

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.

5.14.24 GetTargetCnt()

Returns the count of target headers present. The maximum value supported is defined in OBEX_MAX_TARGET, with value 3.

Table 150: GetTargetCnt ()

GetTargetCnt ()	
Prototype	UINT32 GetTargetCnt ();
Parameters	None
Returns	Number of targets currently present in CObexHeaders object

5.14.25 DeleteTarget()

Deletes the indicated target header from the array of target headers. After the deletion the count is corrected and array is compacted.

Table 151: DeleteTarget ()

DeleteTarget ()	
Prototype	BOOL DeleteTarget (UINT16 index);
Parameters	*index* – selects element from array, index 0 refers to the first element
Returns	TRUE is the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.

5.14.26 GetTargetLength()

Tests if the header is present. If so the header length is provided.

Table 152: GetTargetLength ()

GetTargetLength ()	
Prototype	BOOL GetTargetLength (UINT32 * 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.

5.14.27 GetTarget()

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.

Table 153: GetTarget ()

GetTarget ()	
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.

5.14.28 AddHttp()

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 1.

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.

Table 154: AddHttp ()

AddHttp ()	
Prototype	BOOL AddHttp (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.

5.14.29 GetHttpCnt()

Returns the OBEX header parameter – HTTP count. The maximum value supported is defined in OBEX_MAX_HTTP, with value 1.

Table 155: GetHttpCnt ()

GetHttpCnt ()	
Prototype	UINT32 GetHttpCnt ();
Parameters	None
Returns	Number of HTTP headers currently defined in CObexHeaders object

5.14.30 DeleteHttp ()

Deletes the indicated HTTP header from the array of HTTP headers. After the deletion the count is corrected and array is compacted.

Table 156: DeleteHttp ()

DeleteHttp ()	
Prototype	BOOL DeleteHttp (UINT16 index);
Parameters	*index* – selects element from array, index 0 refers to the first element
Returns	TRUE is the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.

5.14.31 GetHttpLength()

Tests if the header is present. If so the header length is provided.

Table 157: GetHttpLength ()

GetHttpLength ()	
Prototype	BOOL GetHttpLength (UINT32 * 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.

5.14.32 GetHttp()

Returns a copy of the selected HTTP to the caller's buffer.

Table 158: GetHttp ()

GetHttp ()	
Prototype	BOOL GetHttp (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.

5.14.33 SetBody()

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.

Table 159: SetBody ()

SetBody ()	
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 of 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.

5.14.34 DeleteBody()

Deletes the header.

Table 160: DeleteBody ()

DeleteBody ()	
Prototype	void DeleteBody();
Parameters	None
Returns	Void

5.14.35 GetBodyLength()

Tests if the header is present. If so the header length is provided.

Table 161: GetBodyLength ()

GetBodyLength ()	
Prototype	BOOL GetBodyLength (UINT32 * p_length);
Parameters	*p_length* – When the header is present, used to provide number of characters to allow for the header value
Returns	TRUE if the header is present, and the header length is provided. Otherwise FALSE.

5.14.36 GetBody()

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.

Table 162: GetBody ()

GetBody ()	
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.

5.14.37 SetWho()

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.

Table 163: SetWho ()

SetWho ()	
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.

5.14.38 DeleteWho()

Deletes the header.

Table 164: DeleteWho ()

DeleteWho ()	
Prototype	void DeleteWho();
Parameters	None
Returns	Void

5.14.39 GetWhoLength()

Tests if the header is present. If so the header length is provided.

Table 165: GetWhoLength ()

GetWhoLength ()	
Prototype	BOOL GetWhoLength (UINT32 * p_length);
Parameters	*p_length* – When the header is present, used to provide number of characters to allow for the header value
Returns	TRUE if the header is present, and the header length is provided. Otherwise FALSE.

5.14.40 GetWho()

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.

Table 166: GetWho ()

GetWho ()	
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.

5.14.41 AddAppParam()

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.

Table 167: AddAppParam ()

AddAppParam ()	
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.

5.14.42 GetAppParamCnt()

Returns the application parameter count. The maximum value supported is defined in OBEX_MAX_APP_PARAM, with value 3.

Table 168: GetAppParamCnt ()

GetAppParamCnt ()	
Prototype	UINT32 GetAppParamCnt ();
Parameters	None
Returns	Number of application parameters currently defined

5.14.43 DeleteAppParam ()

Deletes the indicated HTTP header from the array of HTTP headers. After the deletion the count is corrected and array is compacted.

Table 169: DeleteAppParam ()

DeleteAppParam ()	
Prototype	BOOL DeleteAppParam(UINT16 index);
Parameters	*index* – selects element from array, index 0 refers to the first element
Returns	TRUE is the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.

5.14.44 GetAppParamLength()

Tests if the header is present. If so the header length is provided.

Table 170: GetAppParamLength ()

GetAppParamLength ()	
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.

5.14.45 GetAppParam()

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.

Table 171: GetAppParam ()

GetAppParam ()	
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

5.14.46 AddAuthChallenge()

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.

Table 172: AddAuthChallenge ()

AddAuthChallenge ()	
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.

5.14.47 GetAuthChallengeCnt()

Returns the OBEX object authentication challenge count. The maximum value supported is defined in OBEX_MAX_ AUTH_CHALLENGE, with value 3.

Table 173: GetAuthChallengeCnt ()

GetAuthChallengeCnt ()	
Prototype	UINT32 GetAuthChallengeCnt ();

Parameters	None
Returns	Number of authentication challenge parameters currently defined in CObexHeaders object

5.14.48 DeleteAuthChallenge ()

Deletes the indicated authentication challenge from the array of authentication challenge headers. After the deletion the count is corrected and array is compacted.

Table 174: DeleteAuthChallenge ()

DeleteAuthChallenge ()	
Prototype	BOOL DeleteAuthChallenge (UINT16 index);
Parameters	*index* – selects element from array, index 0 refers to the first element
Returns	TRUE is the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.

5.14.49 GetAuthChallengeLength()

Tests if the header is present. If so the header length is provided.

Table 175: GetAuthChallengeLength ()

GetAuthChallengeLength ()	
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.

5.14.50 GetAuthChallenge()

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.

Table 176: GetAuthChallenge ()

GetAuthChallenge ()	
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

5.14.51 AddAuthResponse()

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 challenge 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.

Table 177: AddAuthResponse ()

AddAuthResponse ()	
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.

5.14.52 GetAuthResponseCnt()

Returns the OBEX object authentication response count. The maximum value supported is defined in OBEX_MAX_AUTH_RESPONSE, with value 3.

Table 178: GetAuthResponseCnt ()

GetAuthResponseCnt ()	
Prototype	UINT32 GetAuthResponseCnt ();
Parameters	None
Returns	Number of authentication response parameters currently defined in CObexHeaders object

5.14.53 DeleteAuthResponse ()

Deletes the indicated authentication response from the array of authentication response headers. After the deletion the count is corrected and array is compacted.

Table 179: DeleteAuthResponse()

DeleteAuthResponse ()	
Prototype	BOOL DeleteAuthResponse (UINT16 index);
Parameters	*index* – selects element from array, index 0 refers to the first element
Returns	TRUE is the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.

5.14.54 GetAuthResponseLength()

Tests if the header is present. If so the header length is provided.

Table 180: GetAuthResponseLength ()

GetAuthResponseLength ()	
Prototype	BOOL GetAuthResponseLength (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.

5.14.55 GetAuthResponse()

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.

Table 181: GetAuthResponse ()

GetAuthResponse ()	
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

5.14.56 SetObjectClass

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.

Table 182: SetObjectClass ()

SetObjectClass ()	
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.

5.14.57 DeleteObjectClass ()

Deletes the header.

Table 183: DeleteObjectClass ()

DeleteObjectClass ()	
Prototype	void DeleteObjectClass ();
Parameters	None
Returns	Void

5.14.58 GetObjectClassLength()

Tests if the header is present. If so the header length is provided.

Table 184: GetObjectClassLength ()

GetObjectClassLength ()	
Prototype	BOOL GetObjectClassLength (UINT32 * p_length);
Parameters	*p_length* – When the header is present, used to provide number of characters to allow for the header value
Returns	TRUE if the header is present, and the header length is provided. Otherwise FALSE.

5.14.59 GetObjectClass ()

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.

Table 185: GetObjectClass ()

GetObjectClass ()	
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.

5.14.60 AddUserDefined

Adds a user defined entry, as an DK class CObexUserDefined to the OBEX header. The contents of the CObexUserDefined object is 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 1.

Table 186: AddUserDefined ()

AddUserDefined ()	
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.

5.14.61 GetUserDefinedCnt()

Returns the OBEX object user defined count. The maximum value supported is defined in OBEX_MAX_ USER_HDR, with value 1.

Table 187: GetUserDefinedCnt ()

GetUserDefinedCnt ()	
Prototype	UINT32 GetUserDefinedCnt ();
Parameters	None
Returns	Number of user defined parameters currently defined in CObexHeaders object

5.14.62 DeleteUserDefined ()

Deletes the indicated 'user defined' entry from the array of 'user defined' headers. After the deletion the count is corrected and array is compacted.

Table 188: DeleteUserDefined ()

DeleteUserDefined ()	
Prototype	BOOL DeleteUserDefined (UINT16 index);
Parameters	*index* – selects element from array, index 0 refers to the first element
Returns	TRUE is the indexed header was present and has been deleted. FALSE if the index was invalid or the indexed header was not present.

5.14.63 GetUserDefinedLength()

Tests if the header is present. If so the header length is provided.

Table 189: GetUserDefinedLength ()

GetUserDefinedLength ()	
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.

5.14.64 GetUserDefined()

Returns a copy of the selected user defined entry to the caller's buffer.

Table 190: GetUserDefined ()

GetUserDefined ()	
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

5.15 CLASS COBEXUSERDEFINED

This class contains a user defined header. The header identifier must be in the range 0x30-3f, so as not to conflict with pre-defined OBEX header identifiers.

5.15.1 CObexUserDefined Default Constructor

The default constructor creates an object with identifier 0x30, and a 1-byte user type with byte value 0.

Table 191: CobexUserDefined ()

CobexUserDefined ()	
Prototype	CObexUserDefined ();
Parameters	None
Returns	-

5.15.2 SetHeader()

There are multiple signatures for this method, to support different data types for the header value.

5.15.2.1 Single Byte Header

Table 192: SetHeader ()

SetHeader ()	
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 0x30-3f

5.15.2.2 Four Byte Header

Table 193: SetHeader ()

SetHeader ()	
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 0x30-3f

5.15.2.3 ASCII Text Header

Table 194: SetHeader ()

SetHeader ()	
Prototype	BOOL SetHeader (UINT8 id, char * p_text);
Parameters	*id* – header identifier *p_text* – pointer to null-terminated ASCII string
Returns	TRUE if header value is valid and string length <65536; FALSE if the identifier not in the range 0x30-3f or length too long

5.15.2.4 Octet Array Header

Table 195: SetHeader ()

SetHeader ()	
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

5.15.3 GetUserType()

Returns a value corresponding of of the user defined header types. A valid type is always defined, even if a SetHeader() call is never made, because the CObexUserDefined constructor initializes the object as a four byte type with value 0.

Table 196: GetUserType ()

GetUserType ()	
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_TEXT – for ASCII text OBEX_USER_TYPE_DATA – for unstructured octet array OBEX_USER_TYPE_BYTE - for the single byte OBEX_USER_TYPE_INT – for the four byte (unsigned int) value

5.15.4 GetByte()

Table 197: GetByte ()

GetByte ()	
Prototype	BOOL GetByte (UINT8 *p_byte);
Parameters	*p_byte* – points to caller's field where value is to be copied
Returns	TRUE is the header really is of type OBEX_USER_TYPE_BYTE; FALSE otherwise

5.15.5 GetFourByte()

Table 198: GetFourByte ()

GetFourByte ()	
Prototype	BOOL GetFourByte (UINT32 *p_fourbyte);
Parameters	*p_fourbyte* - points to caller's field where value is to be copied
Returns	TRUE is the header really is of type OBEX_USER_TYPE_INT; FALSE otherwise

5.15.6 GetLength()

Returns the length of the user defined parameter header.

Table 199: GetLength ()

GetLength ()	
Prototype	UINT16 GetLength ();
Parameters	None
Returns	Length

5.15.7 GetText()

Table 200: GetText ()

GetText ()	
Prototype	BOOL GetText (char *p_text);
Parameters	*p_text* - points to caller's field where value is to be copied, including the terminating null.
Returns	TRUE is the header really is of type OBEX_USER_TYPE_TEXT, and a pointer has been set; FALSE otherwise

5.15.8 GetOctets()

Table 201: GetOctets ()

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 is the header really is of type OBEX_USER_TYPE_DATA, and a pointer has been set; FALSE otherwise

5.16 CLASS 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 Get, 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. 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.

5.16.1 tOBEX_ERRORS

A common set of return codes are provided by the OBEX function calls. Use enumerated type tOBEX_ERRORS, from BtIfDefinitions.h

Table 202: tOBEX_ERRORS

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 register 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

5.16.2 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.

Table 203: Open()

Open ()	
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 definition of tOBEX_ERRORS

5.16.3 SetPath()

The client application calls this function to send a formatted OBEX SetPath request to the server.

Table 204: SetPath()

SetPath ()	
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 definition of tOBEX_ERRORS

5.16.4 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 non-zero 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()*.

Table 205: Put()

Put ()	
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 definition of tOBEX_ERRORS

5.16.5 Get()

The client application calls this function to send a formatted OBEX Get request to the server.

Table 206: Get()

Get ()	
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 definition of tOBEX_ERRORS

5.16.6 Abort()

The client application calls this function to send a formatted OBEX Abort request to the server.

Table 207: Abort()

Abort ()	
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 definition of tOBEX_ERRORS

5.16.7 Close()

The client application calls this function to send a formatted OBEX Close request to the server.

Table 208: Close()

Close ()	
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 definition of tOBEX_ERRORS

5.16.8 pure virtual OnOpen()

This derived function is defined by the application and is called when the Open Confirmation response is received from the server.

Table 209: pure virtual OnOpen()

pure virtual OnOpen ()	
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. If the client exceeds this size OBEX will automatically segment the buffer. 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 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. *code* – response code - OBEX_SUCCESS if the request succeeded, otherwise see tOBEX_ERRORS in BtIfDefinitions.h for the possible error cases. *response* – response code sent by server, see BtIfDefinitions.h for list of values
Returns	Void

5.16.9 pure virtual OnClose()

This derived function is defined by the application and is called when the Close Confirmation response is received from the server.

Table 210: pure virtual OnClose()

pure virtual OnClose ()	
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* – response code - OBEX_SUCCESS if the request succeeded, otherwise see tOBEX_ERRORS in BtIfDefinitions.h for the possible error cases. *response* – response code sent by server, see BtIfDefinitions.h for list of values
Returns	Void

5.16.10 virtual OnAbort()

This derived function is defined by the application and is called when the Abort Confirmation response is received from the server.

Table 211: virtual OnAbort()

virtual OnAbort ()	
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* – response code - OBEX_SUCCESS if the request succeeded, otherwise see tOBEX_ERRORS in BtIfDefinitions.h for the possible error cases.

	response – response code sent by server, see BtIfDefinitions.h for list of values
Returns	Void

5.16.11 virtual OnPut()

This derived function is defined by the application and is called when the Put Confirmation response is received from the server.

Table 212: virtual OnPut()

virtual OnPut ()	
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* – response code - OBEX_SUCCESS if the request succeeded, otherwise see tOBEX_ERRORS in BtIfDefinitions.h for the possible error cases. *response* – response code sent by server, see BtIfDefinitions.h for list of values
Returns	Void

5.16.12 virtual OnGet()

This derived function is defined by the application and is called when the Get Confirmation response is received from the server.

Table 213: virtual OnGet()

virtual OnGet ()	
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* – response code - OBEX_SUCCESS if the request succeeded, otherwise see tOBEX_ERRORS in BtIfDefinitions.h for the possible error cases. *final* – set TRUE by the server when this is the last final transfer for an object *response* – response code sent by server, see BtIfDefinitions.h for list of values
Returns	Void

5.16.13 virtual OnSetPath()

This derived function is defined by the application and is called when the OnSetPath Confirmation response is received from the server.

Table 214: virtual OnSetPath()

virtual OnSetPath ()	
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* – response code - OBEX_SUCCESS if the request succeeded, otherwise see tOBEX_ERRORS in BtIfDefinitions.h for the possible error cases. *response* – response code sent by server, see BtIfDefinitions.h for list of values
Returns	Void

5.17 CLASS 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 DK class CRfCommIf to assign an SCN and a security level to the application GUID.
2. Create a service record. The DK 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 ort* 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.

5.17.1 tOBEX_ERRORS

A common set of return codes are provided by the OBEX function calls. Use enumerated type tOBEX_ERRORS, from BtIfDefinitions.h

Table 215: tOBEX_ERRORS

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 register 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

5.17.2 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.

The server application can optionally designate itself to be a 'target'. Based on the target parameter the server platform can permit multiple server applications:

- p_target is NULL. This will be the only server for the scn. No other server will be allowed to register. Also, 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 zero length string. This server application will be the 'default server' for the scn. Other server applications can also register as long as they provide a unique target (non zero length). The default server will receive connection from clients which specify no targets, or which specify targets not provided by other server applications.
- p_target points to a non-zero length string. This server application can be one of many registered for the scn, as long as the target string is unique. This server application will connect with only those client requests that contain a target that matches.

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.

Table 216: Register ()

Register()	
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 definition of tOBEX_ERRORS

5.17.3 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.

Table 217: Unregister ()

Unregister ()	
Prototype	tOBEX_ERRORS Unregister();
Parameters	None
Returns	OBEX_SUCCESS – if all was OK Otherwise, see definition of tOBEX_ERRORS

5.17.4 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.

Table 218: OpenCnf ()

OpenCnf ()	
Prototype	tOBEX_ERRORS OpenCnf(tOBEX_ERRORS obex_errors, tOBEX_RESPONSE_CODE rsp_code, CObexHeaders * p_response, UINT16 mtu= OBEX_DEFAULT_MTU);
Parameters	*obex_errors* - Error code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_ERRORS rsp_code –Response code. 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 definition of tOBEX_ERRORS

5.17.5 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.

Table 219: SetPathCnf ()

SetPathCnf ()	
Prototype	tOBEX_ERRORS SetPathCnf (tOBEX_ERRORS obex_errors, tOBEX_RESPONSE_CODE rsp_code, CObexHeaders * p_response);
Parameters	*obex_errors* - Error code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_ERRORS rsp_code –Response code. 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 definition of tOBEX_ERRORS

5.17.6 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.

Table 220: PutCnf ()

PutCnf ()	
Prototype	tOBEX_ERRORS PutCnf (tOBEX_ERRORS obex_errors, tOBEX_RESPONSE_CODE rsp_code, CObexHeaders * p_response);

Parameters	*obex_errors* - Error code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_ERRORS rsp_code –Response code. 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 definition of tOBEX_ERRORS

5.17.7 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.

Table 221: PutCreateCnf ()

PutCreateCnf ()	
Prototype	tOBEX_ERRORS PutCreateCnf (tOBEX_ERRORS obex_errors, tOBEX_RESPONSE_CODE rsp_code, CObexHeaders * p_response);
Parameters	*obex_errors* - Error code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_ERRORS rsp_code –Response code. 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 definition of tOBEX_ERRORS

5.17.8 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.

Table 222: PutDeleteCnf ()

PutDeleteCnf ()	
Prototype	tOBEX_ERRORS PutDeleteCnf (tOBEX_ERRORS obex_errors, tOBEX_RESPONSE_CODE rsp_code, CObexHeaders * p_response);
Parameters	*obex_errors* - Error code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_ERRORS rsp_code –Response code. 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 definition of tOBEX_ERRORS

5.17.9 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.

Table 223: GetCnf ()

GetCnf ()	
Prototype	tOBEX_ERRORS GetCnf (tOBEX_ERRORS obex_errors,

	tOBEX_RESPONSE_CODE rsp_code, BOOL final, CObexHeaders * p_response);
Parameters	*obex_errors* - Error code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_ERRORS rsp_code –Response code. 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 definition of tOBEX_ERRORS

5.17.10 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.

Table 224: AbortCnf ()

AbortCnf ()	
Prototype	tOBEX_ERRORS AbortCnf (tOBEX_ERRORS obex_errors, tOBEX_RESPONSE_CODE rsp_code, CObexHeaders * p_response);
Parameters	*obex_errors* - Error code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_ERRORS rsp_code –Response code. 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 definition of tOBEX_ERRORS

5.17.11 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.

Table 225: CloseCnf ()

CloseCnf ()	
Prototype	tOBEX_ERRORS CloseCnf (tOBEX_ERRORS obex_errors, tOBEX_RESPONSE_CODE rsp_code, CObexHeaders * p_response);
Parameters	*obex_errors* - Error code to be sent back to the client. See BtIfDefinitions.h for definition of tOBEX_ERRORS rsp_code –Response code. 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 definition of tOBEX_ERRORS

5.17.12 GetRemoteBDAddr()

The server application calls this function to find the Bluetooth Device Address of the remote client device that initiated the connection.

Table 226: GetRemoteBDAddr ()

GetRemoteBDAddr ()	
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

5.17.13 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.

Table 227:SwitchRole ()

SwitchRole ()	
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.

5.17.14 pure virtual OnOpenInd()

This derived function is defined by the application and is called when the Open request is received from the client.

Table 228: OnOpenInd ()

OnOpenInd ()	
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

5.17.15 pure virtual OnSetPathInd()

This derived function is defined by the application and is called when the OnSetPath request is received from the client.

Table 229: pure virtual OnSetPathInd()

pure virtual OnSetPathInd ()	
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 the server is requested to create a new folder with the object's name, if one does not already exist.
Returns	void

5.17.16 pure virtual OnPutInd()

This derived function is defined by the application and is called when the Put request is received from the client.

Table 230: pure virtual OnPutInd()

pure virtual OnPutInd ()	
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*

5.17.17 pure virtual OnPutCreateInd()

This derived function is 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'.

Table 231: pure virtual OnPutCreateInd ()

pure virtual OnPutCreateInd ()	
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

5.17.18 pure virtual OnPutDeleteInd()

This derived function is 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'.

Table 232: pure virtual OnPutDeleteInd ()

pure virtual OnPutDeleteInd ()	
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

5.17.19 pure virtual OnGetInd()

This derived function is defined by the application and is called when the Get request is received from the client.

Table 233: pure virtual OnGetInd ()

pure virtual OnGetInd ()	
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

5.17.20 pure virtual OnAbortInd()

This derived function is defined by the application and is called when the Abort request is received from the client.

Table 234: pure virtual OnAbortInd ()

pure virtual OnAbortInd ()	
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

5.17.21 pure virtual OnCloseInd()

This derived function is defined by the application and is called when the Close request is received from the client.

Table 235: pure virtual OnCloseInd ()

pure virtual OnCloseInd ()	
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

6 Sample Applications

6.1 USING DK CLASSES IN AN APPLICATION

Sample applications are provided in the form of Microsoft Visual Studio projects.

Two Bluetooth-enabled PCs are required to run the sample applications.

These sample applications are built using MFC dialog applications. The primary application class is based on the CDialog class.

In addition, the DK classes CRfCommPort, CL2CapConn, CFtpClient, COppClient, and CLapClient, CSppClient, CSppServer, CObexClient, and CObexServer are base classes containing virtual methods that are provided by the applications. These virtual methods serve as event handlers for the application, for events such as file transfer complete, progress, etc.

NOTE: Virtual functions are called on an independent thread; operations performed within the virtual functions must be made thread safe.

The sample application dialog classes (CBlueChatDlg, CBlueTimeDlg, CBlueClientDlg) use multiple inheritance from CDialog and whichever DK base classes are needed for the application. This approach makes it convenient for the application to define derived methods for the DK virtual methods. These derived methods have access to the dialog class services and can easily share data with the main sample dialog object.

6.2 OTHER APPROACHES

Other approaches are possible for using the DK classes. Suppose an application developer wants to use the DK base class CRfCommPort, but does not want the application's main dialog module to be based on CRfCommPort. The application may need to support two RFCOMM connections, for example.

In this case the developer could define a derived class, CRfCommPortDerived, which supplies all the virtual methods for using the DK base class CRFCommPort. Then the application's main program, or main dialog module, would define two instance members, such as:

CRfCommPortDerived connection_a;

CRfCommPortDerived connection_b;

The developer would need to supply methods for CRfCommPortDerived that allow the main module to exchange data and status with connection_a and connection_b.

How this would be done would depend on the application requirements.

Class CRfCommPortDerived could be defined with a public method that returned current status to the caller. This would allow the main module to poll the CRfCommPortDerived object, passing commands and receiving status.

If polling by the main module was not desireable, the CRfCommPortDerived constructor could be defined to allow the main module to pass a 'this' pointer. Then the CRfCommPortDerived object could call functions in the main module in an event-driven manner.

6.3 BLUETIME, AN L2CAP TIME MONITOR

This application creates a client-server L2CAP connection. Once per second each side sends its local time to the other side.

BlueChat is installed on two platforms. When they are executed, one plays the part of client the other the server, as selected by the user.

Both sides display the current local time and the latest time sent from the other side.

The user establishes the connection; the server creates a new service and then waits for a client to connect. The client performs a device inquiry and service discovery and then the two sides connect and communicate.

To run the BlueTime sample application:

1. Start Microsoft Visual C++.
2. Open the workspace named "BlueTime.dsw" in the `\Sdk\Samples\BlueTime` directory.
3. Build the application.
4. Run `BlueTime.exe`

6.3.1 BlueTime Functionality

Figure 2: BlueTime Main Window at Start

At startup a decision to run the application as a server or a client must be made.

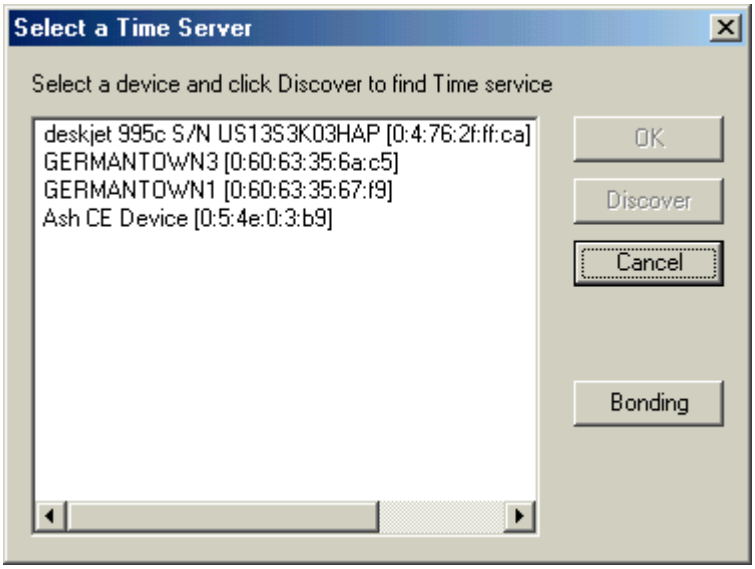
Server mode:

- Select the **Act as Time Server** check box.
- Click the **Start Session** Button. BlueTime starts a session and:
  - Adds a Service Record using SDP.
  - Opens an L2CAP server connection.
  - Waits for a client connection.

Client mode:

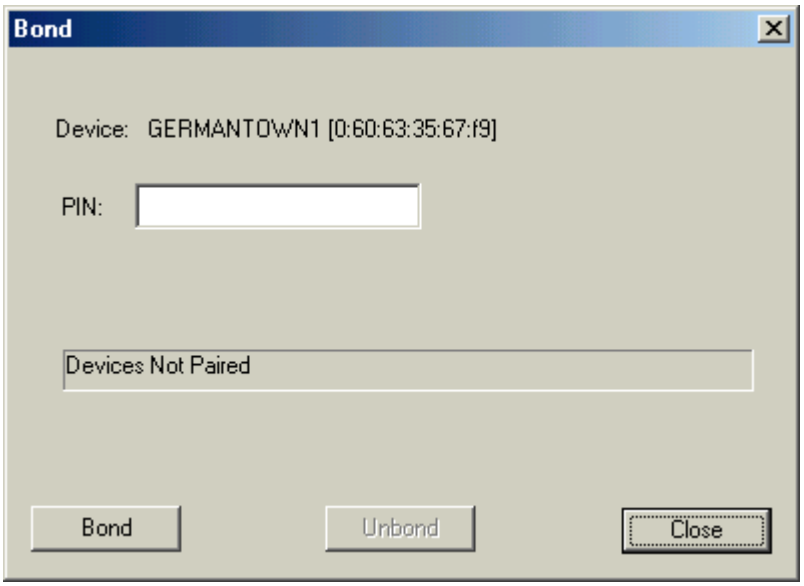
The user is presented with a dialog box so that a Time Server can be selected. The server list is populated automatically with the Bluetooth devices found using device Inquiry.

Figure 3: BlueTime Choose Server Dialog Window



The user may select a device to perform a bonding operation, by pressing the **Bonding** button. The result is a dialog window indicating if the selected device is currently paired with the local device or not. See below:

Figure 4: Bonding Dialog Window



If the devices are already paired an **Unbond** button lets the user delete the bond. If the devices are not already bonded the user may enter a PIN code, then press the **Bond** button, to initiate the bonding procedure. The BlueTime application will wait until the bonding is resolved, either by approval from the selected server, rejection by the server, or timeout. The result will be reported on the bonding dialog window.

In the displayed list, those devices which offer the BlueTime server will be indicated with a suffix ("<-Time Server") to the device name.

- Select a "Time Server" host from the list.
- Click **Discover** to send a discovery request to the Time Server.
- Click the **Okay** button. The dialog window closes and an L2CAP client connection is opened with the Time Server host.

BlueTime is ready to communicate.

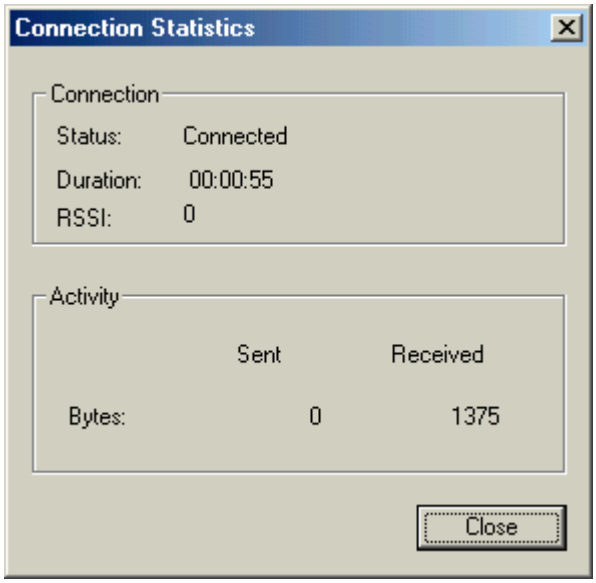
Both Client and Server Mode:

To exit the application, click the **Close** button.

Either mode may use the 'Security' button to select one or more security features for connections. The choices are: 'No Security', 'Authorize Incoming', Authenticate Incoming', Encrypt Incoming', 'Authenticate Outgoing', and 'Encrypt Outgoing'.

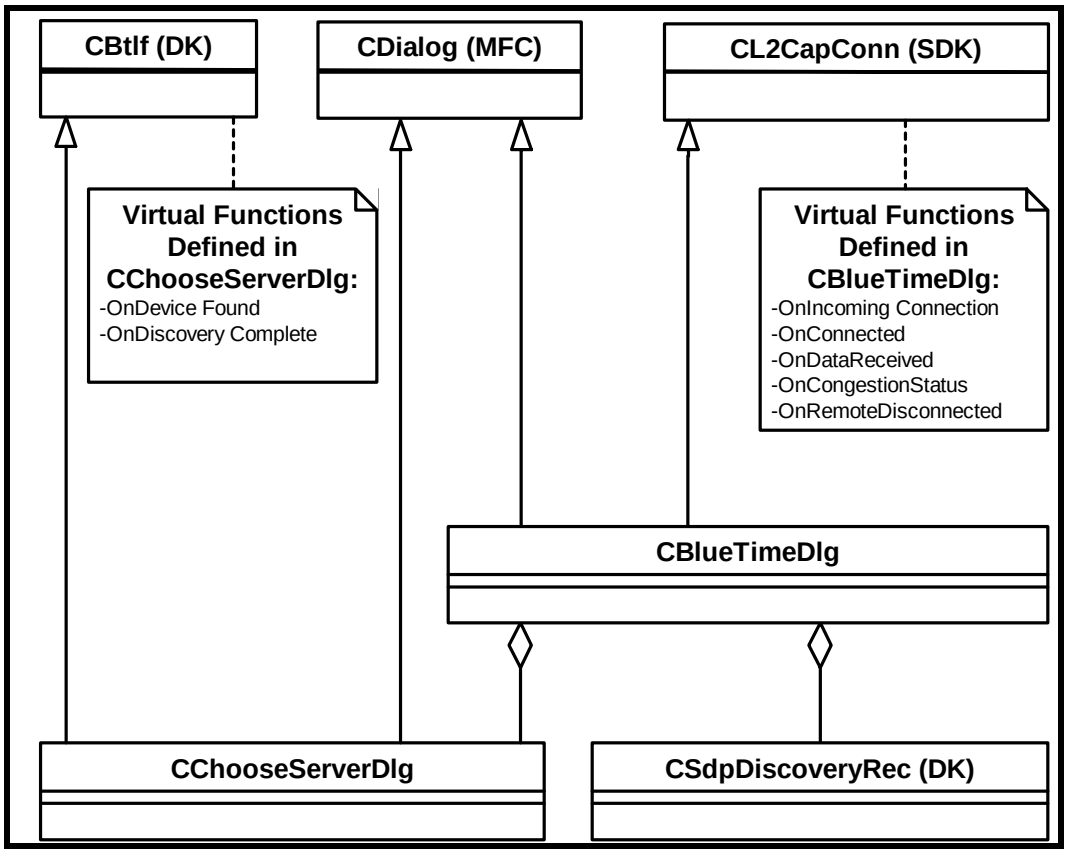
The **Conn Stats** button brings up a modeless window that shows the current connection statistics, updated once per second. See CL2CapConn::GetConnectionStats() for details.

Figure 5: BlueTime Connection Stats Window



6.3.2 Key Class Descriptions

Figure 6: BlueTime—the relationships between the DK classes and the primary application classes, CBlueTimeDlg and CChooseServerDlg.



CBlueTimeDlg–implements the dialog functionality for the main chat window.

This module demonstrates the following features of the DK:

- Create SDP Service records.
- Open a client or server L2CAP connection.
- Set the security level.
- Check the connection.
- Write data to the port.
- Close the connection.

This is a typical MFC dialog class, derived on the CDialog base class.

For this application this class also inherits from the DK class CL2CapConn. The virtual methods from CL2CapConn may then be incorporated within CBlueTimeDlg for convenient access to common data members.

CChooseServerDlg–implements the dialog functionality for choosing a Time Server.

This module demonstrates the following features of the DK:

- Use the BT API class.
- Perform a device inquiry.
- Perform a service discovery.
- Read discovery records.

6.4 BLUECHAT, AN RFCOMM CHAT APPLICATION

This application creates an RFCOMM connection and permits the client and server to "chat." BlueChat is installed on two platforms. When they are executed, one plays the part of client the other the server, as selected by the user.

The GUI allows modem control signals to be set/read and error conditions to be sent to the other side.

- The user establishes the connection with one PC as the server and the other as the client. The server PC creates a new service, and then waits for the client to connect.
- The client performs a device inquiry and service discovery.
- The two sides connect and communicate.

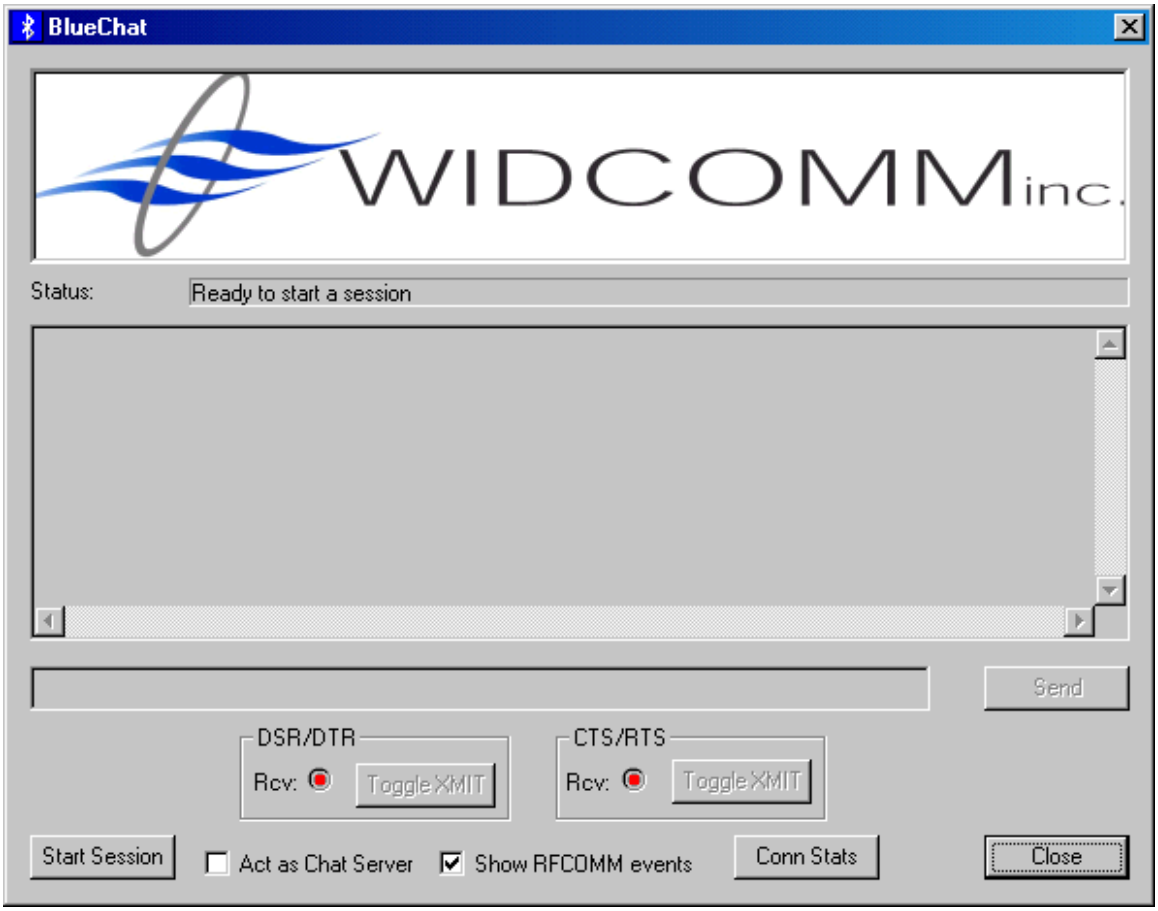
To run the BlueChat sample application:

1. Start Microsoft Visual C++.
2. Open the workspace named "BlueChat.dsw" in the `\Sdk\Samples\BlueChat` directory.
3. Build the application.
4. Run `BlueChat.exe`.

6.4.1 BlueChat Functionality

The BlueChat main window is shown below:

Figure 7: BlueChat Main Window at Start



At startup a decision to run the application as a server or a client must be made.

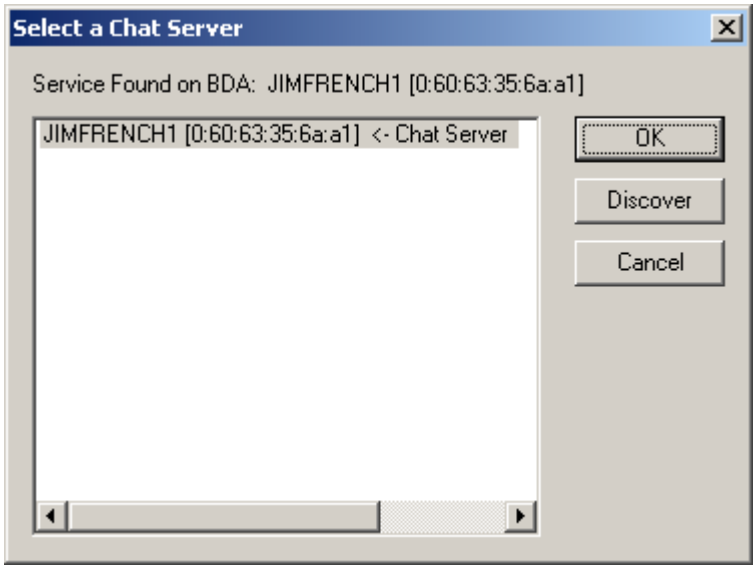
Server mode:

- Select the **Act as Chat Server** check box.
- Click the **Start Session** button. BlueChat starts a session and:
  - Adds a Service Record using DK class CSdpService in function CBlueChatDlg::DoCreateServiceRecord().
  - Opens an RFCOMM server port using OpenServer(), a method inherited from DK class CRfCommPort.
  - Waits for a client connection.

Client mode:

- Click the **Start Session** button. BlueChat starts a client session.
  - The list of servers is populated automatically with the Bluetooth devices found using device Inquiry.
  - The user is presented with a dialog box so that a Chat Server can be selected, as shown below.

Figure 8: BlueChat Choose Server Dialog Window



- Select the Chat Server host from the list.
- Click **Discover** to send a discovery request to the Chat Server.
- Click the **OK** button. The dialog window closes and an RFCOMM client connection is opened with the Chat Server.

BlueChat is ready to communicate; type a message in the edit field and click the **Send** button.

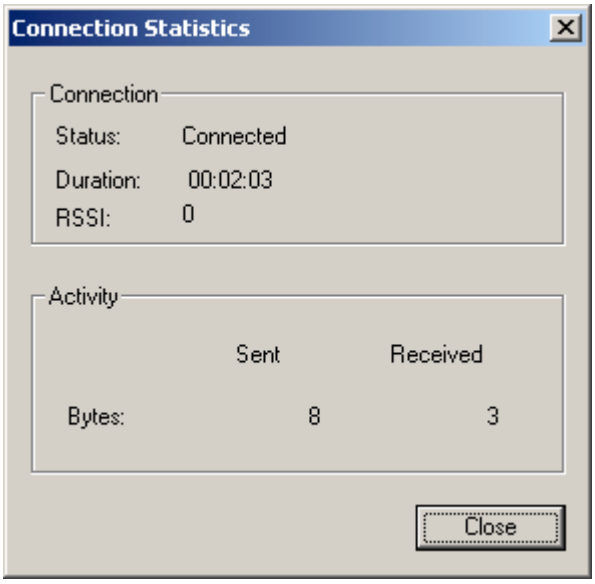
Both Client and Server mode:

Sent and received messages are echoed in the log window. RFCOMM events are also shown in the log window. Clear the **Show RFCOMM Events** check box to disable the display of events.

Status lights indicate control signals sent by the remote chat application. The user can transmit a control signal by clicking the appropriate button.

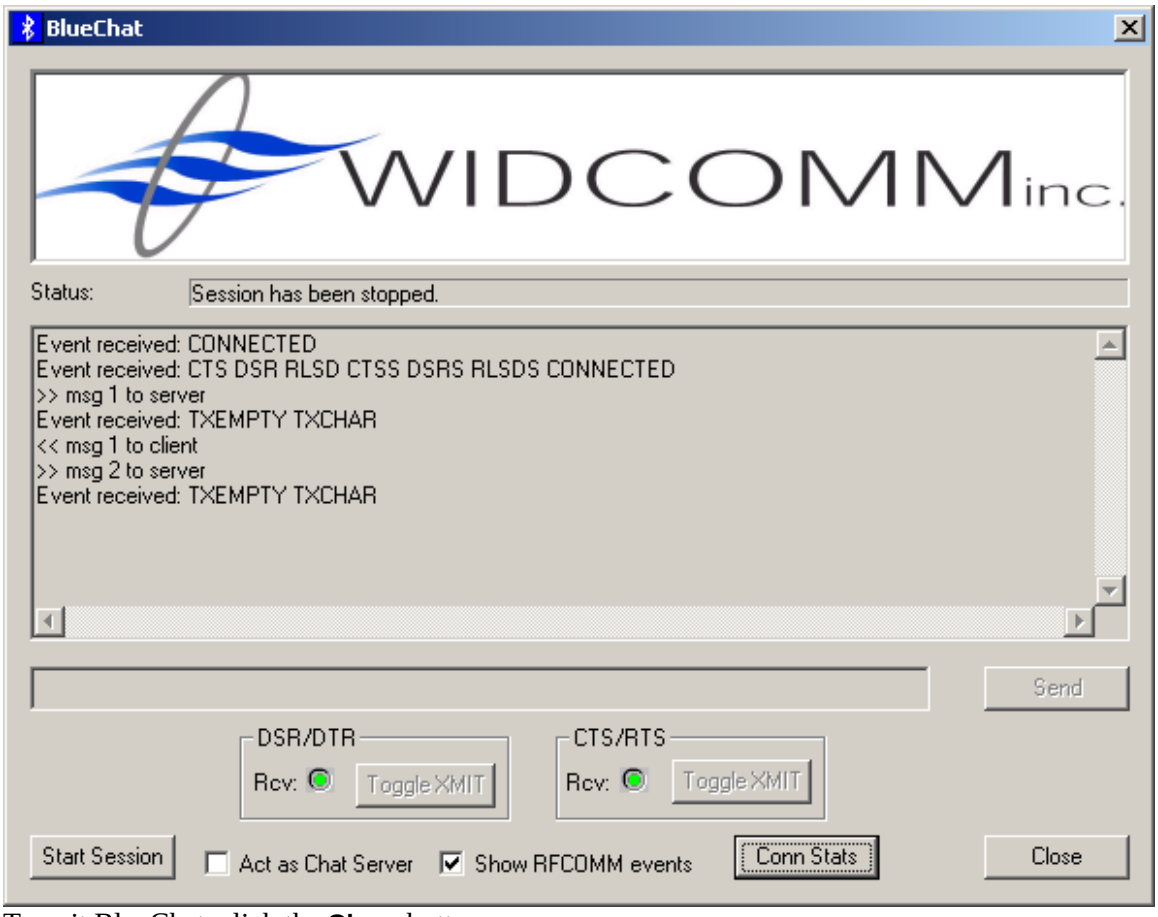
The **Conn Stats** button brings up a modeless window that shows the current connection statistics, updated once per second. See CRfCommPort::GetConnectionStats() for details.

Figure 9: BlueChat Connection Stats Window



After a brief chat session with a server, the main window looks as shown below:

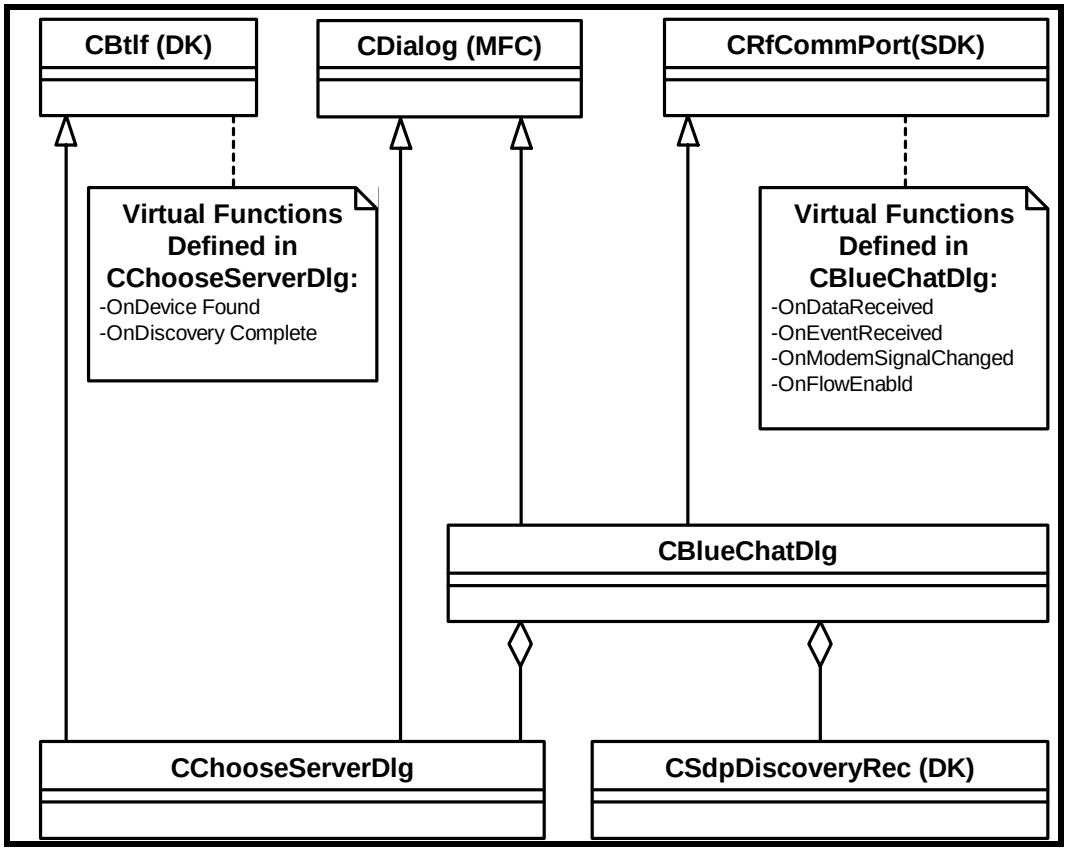
Figure 10: BlueChat Main Window on Client after a Chat Session



To exit BlueChat, click the **Close** button.

6.4.2 Key Class Descriptions

Figure 11: BlueChat—the relationships between the DK classes and the primary application classes, CBlueChatDlg and CChooseServerDlg.



CBlueChatDlg–implements the dialog functionality for the main chat window.

This module demonstrates the following features of the DK:

- Create SDP Service records.
- Open a client or server RFCOMM port.
- Set the security level.
- Check the port connection.
- Write data to the port.
- Get modem status.
- Transmit modem signals.
- Close the RFCOMM port.

This is a typical MFC dialog class, derived on the CDialog base class.

For this application this class also inherits from the DK class CRfCommPort. The virtual methods from CRfCommPort may then be incorporated within CBlueChatDlg for convenient access to common data members.

CChooseServerDlg–implements the dialog functionality for choosing a Chat Server. This module demonstrates the following features of the DK:

- Use the BT API class.
- Perform a device inquiry.
- Perform a service discovery.
- Read discovery records.

6.5 BLUECLIENT, AN FTP OPP APPLICATION

BlueClient provides client access to the FTP and OPP servers in the Bluetooth neighborhood. Connections are setup as needed and client functions are under direct control of the user.

BlueClient performs an inquiry at startup to determine the available Bluetooth devices the neighborhood. The user chooses a server and then requests the discovery of FTP and OPP services, such as file get, file put, object push, etc.

BlueClient has the look-and-feel of the WIDCOMM BT Neighborhood; Bluetooth devices, services, folder names, and file names are displayed in a “tree” format.

To run the BlueClient sample application:

1. Start Microsoft Visual C++.
2. Open the workspace named “BlueClient.dsw” in the `\Sdk\Samples\BlueClient` directory.
3. Build the application.
4. Run `BlueClient.exe`.

6.5.1 BlueClient Functionality

BlueClient runs only as a client. But, another platform is needed to provide the standard profile FTP and OPP services requested by BlueClient.

Figure 12: BlueClient Main Window at start

Most actions are taken in the main dialog window. Right-click the “Neighborhood” entry at the root of the tree to refresh the list of BT server devices.

Under the root there is a branch for each server device. Right-click a device to a request a discovery for it, or “properties”, to show the device properties.

Services (FTP and OPP) are sub-branches of the tree under the device.

Right-click a service to see the available functions or expand the branch to view a sub list of files, for example for FTP. Right-click a specific file to “put” the file to the server.

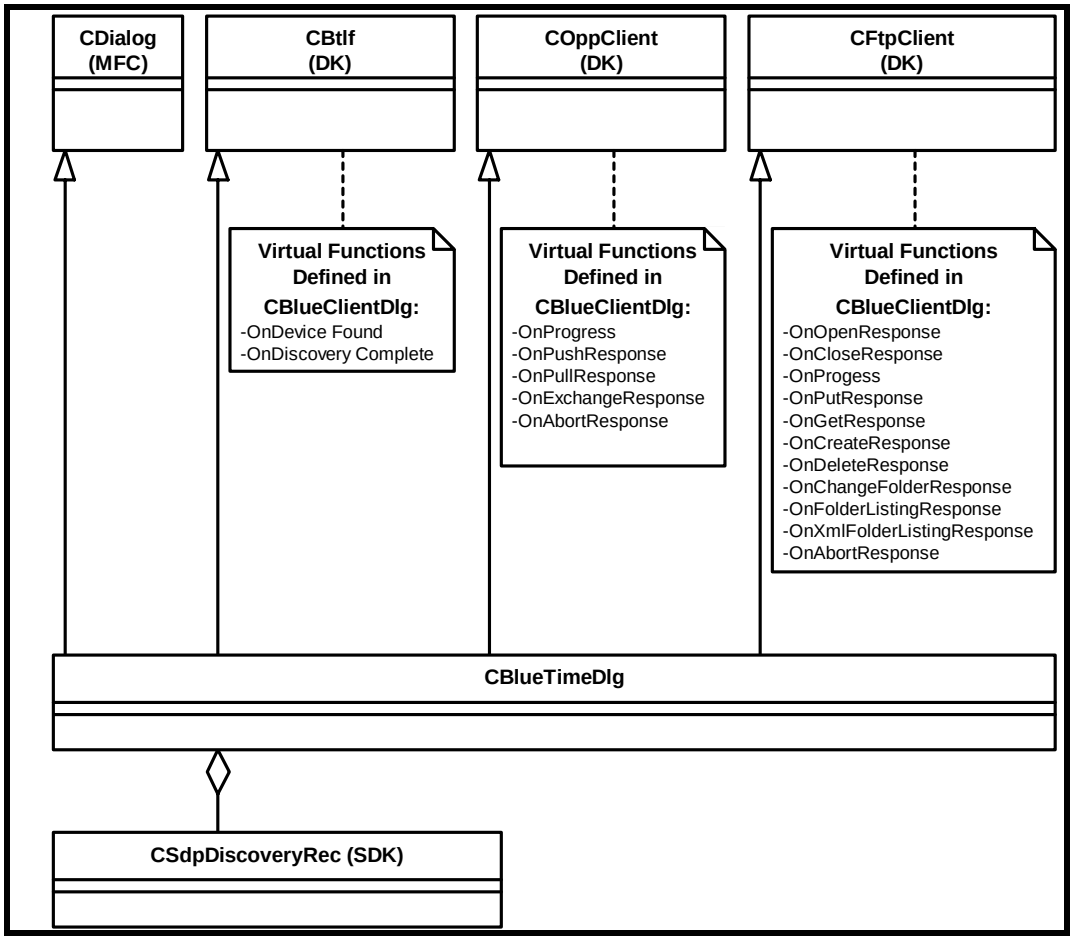
Figure 13: BlueClientWindow, showing an FTP Folder Listing

The status bar at the base of the main dialog window shows the operation in progress, and the result. For file transfers the status bar displays the of progress of the transfer in the format of “x” of “y” bytes transferred.

Subsidiary dialog windows are used, as necessary, to select files to be transferred, show the properties of servers, to select a file to delete, etc.

6.5.2 Key Class Descriptions

Figure 14: BlueClient—the relationships between the DK classes and the primary application class, CBlueClientDlg.



CBlueClientDlg–implements the dialog functionality for the main BlueClient window. This module demonstrates the following features of the DK:

- Start device inquiry and present the available servers in the tree.
- Allow right-click menus to refresh and expand the device services.
- Allow right-click menus to perform specific FTP and OPP operations.
- Present progress and status for the operations requested.

This is a typical MFC dialog class, inheriting from the CDialog base class.

For this application this class also inherits from the DK classes CFtpClient and COppClient. The virtual methods from these base classes may then be incorporated within CBlueClientDlg for convenient access to common data members.

FTPCreate–implements the dialog functionality to obtain a name for a new file under FTP.

CbtDeviceProps–displays the properties of a selected server device. The user-friendly name, the device's Bluetooth address, and the connection state are displayed.

6.6 BLUECOMCHAT, A COM PORT CHAT APPLICATION

This application creates a serial connection over a Windows COM port that has been pre-assigned to a Bluetooth service. The client and server can then "chat."

BlueComChat is installed on two platforms. When they are executed, one plays the part of client the other the server, as selected by the user.

6.6.1 Building the Application

To run the BlueComChat sample application:

- Start Microsoft Visual C++.
- Open the workspace named "BlueComChat.dsw" in the `\Sdk\Samples\BlueComChat` directory.
- Build the application.
- Run `BlueComChat.exe`.

6.6.2 Application Functionality

6.6.2.1 Server Configuration

The sample application BlueComChat expects to use a new service , "Widcomm DK COM Serial Port".

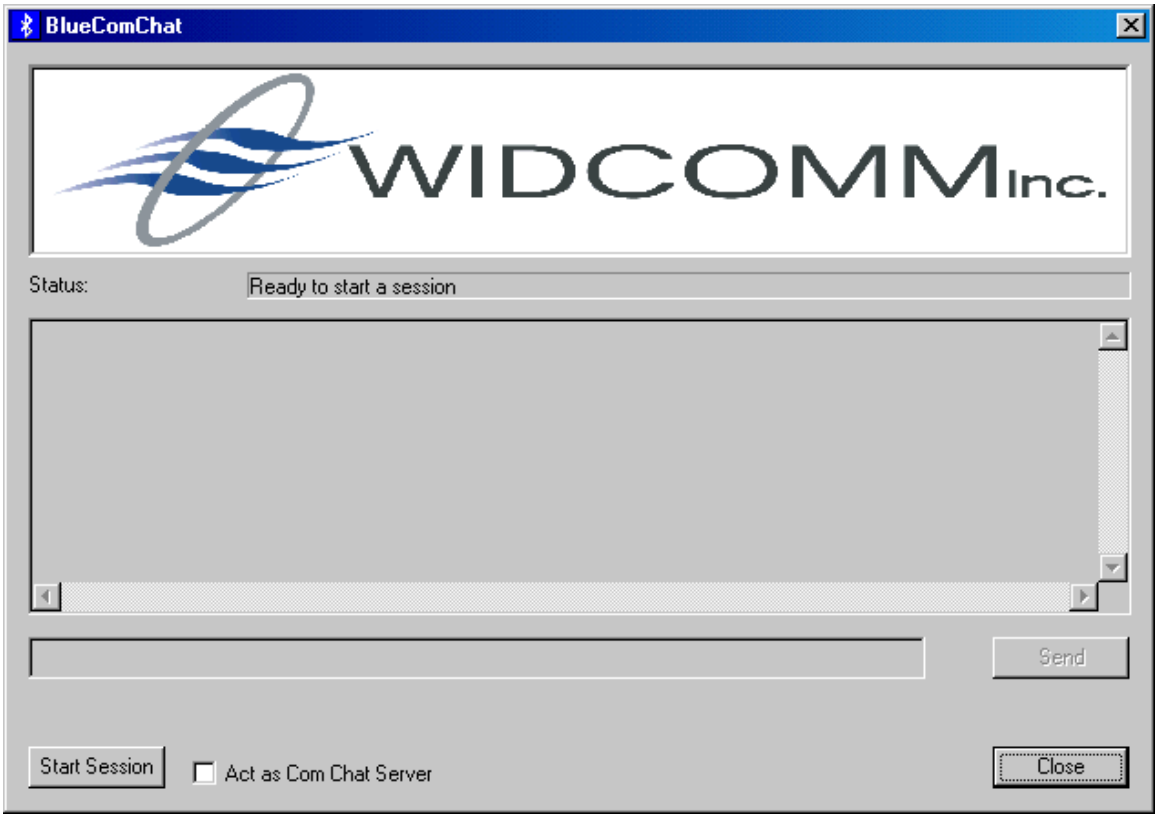
This service must be manually defined on the intended server platform. This is done in the BT Configuration function, 'Local Services' tab, pressing the 'Add Serial Service' button. In the 'Service Properties' dialog, the name entered must be: Widcomm DK COM Serial Port. The default COM port assignment may be accepted, or a different one entered.

The desired security options may be set. The 'Startup' checkbox should not be checked. This means the service will not be started automatically on system startup. As a result the server side application totally controls when the service is started and stopped. No manual intervention is required after the one-time setup

6.6.2.2 Operation

At application startup a decision to run the application as a server or client must be made.

Figure 15: BlueComChatMain Window at Start



On the server side, the one-time configuration procedure must have been performed.

Server mode:

- Select the **Act as Com Chat Server** check box.
- Click the **Start Session** button. BlueComChat starts a chat session and:
  - Initiates an SPP server connection for the service name defined in m_serviceNameForServer in the application source file BlueComChatDlg.cpp (the DK is installed with the service name “Widcomm DK COM Serial Port”).
  - Waits for a client connection.

Client mode:

- Performs device inquiry and service discovery. Servers that offer the serial port service class (GUID equal to CBtIf::guid_SERVCLASS_SERIAL_PORT) are displayed on the client user interface. For each discovered service, the server device name and the service name are displayed.
- The user selects a service to initiate the SPP connection.

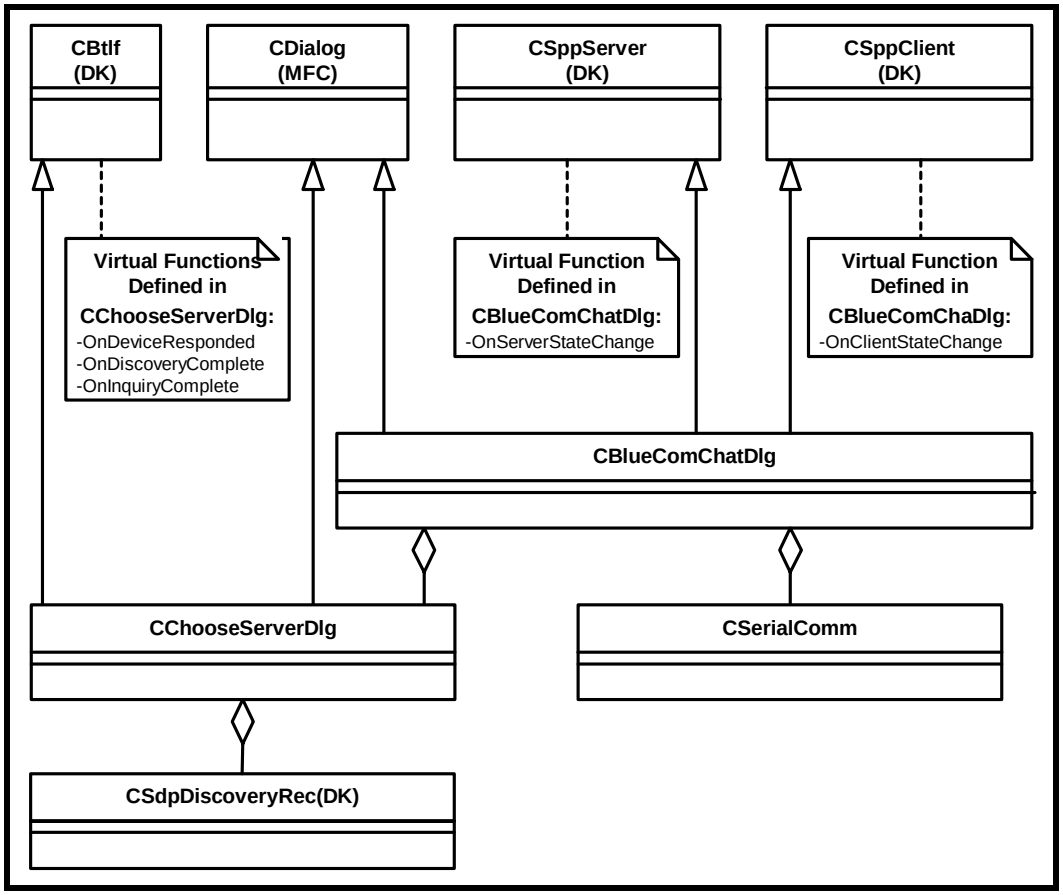
Once the connection is established the text edit line and the and the **Send** buttons are enabled. The users can send messages to each other.

Click the **Stop** button to close the connection.

Click the **Close** button to close the application.

6.6.3 Key Class Descriptions

Figure 16: BlueComChat—the relationships between the DK classes and the primary application class, CBlueComClientDlg.



CBlueCOMChatDlg—implements the dialog functionality for the main chat window. This module demonstrates the following features of the DK:

- Open a client or server Bluetooth SPP connection.
- Check the connection status.
- Open and Close the associated Windows COM serial port.
- Read messages entered by the user and write them to the COM port.
- Read messages received from the COM port and write them to the user interface.

This is a typical MFC dialog class, derived on the CDialog base class.

For this application this class also inherits from the DK classes CSppClient and CSppServer. The virtual methods from these classes may then be incorporated within CBlueComChatDlg for convenient access to common data members.

CChooseServerDlg—implements the dialog functionality to choose a chat server.

This module demonstrates the following features of the DK:

- Perform a device inquiry.
- Perform service discovery.
- Read discovery records and select only those for service class serial port for display.
- Pass the name of the selected service back to CBlueComChatDlg

CSerialComm—implements an interface to the Windows COM serial ports. Methods are provided to Open, Close, Read and Write to the selected COM port.

6.7 OBEX EXERCISER, PROJECT BLUEOBEX

This sample application provides a user interface to run a sequence of object transfers between OBEX client and OBEX server.

BlueObex is installed on two platforms. When they are executed, one plays the part of client the other the server, as selected by the user.

The application also includes a module 'checkheaders', which runs a test sequence validating the OBEX support classes CObexHeaders and CobexUserDefined.

6.7.1 Building the Application

Use the following steps to build the sample:

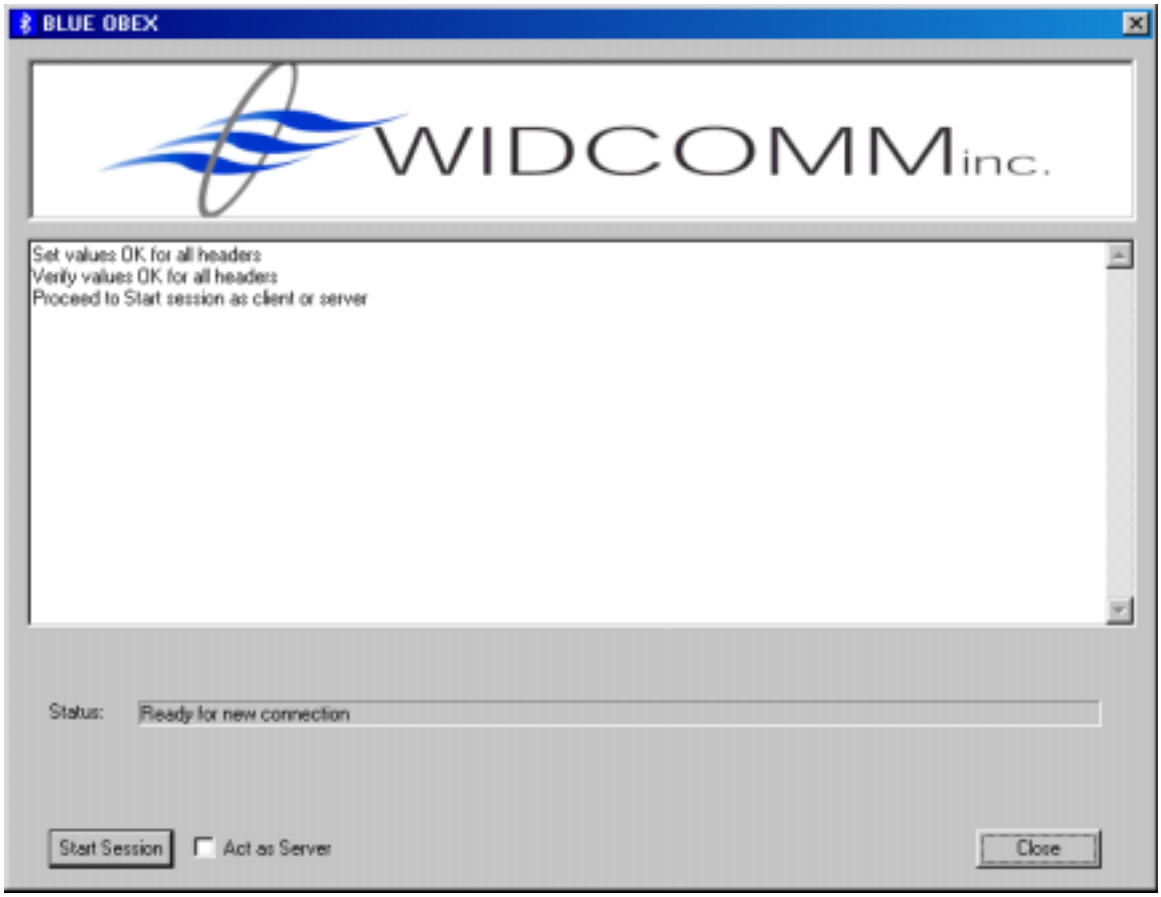
- Start Microsoft Visual C++.
- Open the workspace named 'BlueObex.dsw' in the \Sdk\Samples\BlueObex directory.
- Build the application.
- Run `BlueObex.exe`

This is a Windows32 dialog-based application

6.7.2 Application Functionality

At startup, the application automatically runs the checkheaders function. A pass/fail result is logged to the user interface.

Figure 17: BlueObex Main Window at Start



Then the user can request an OBEX connection, either as a client or as a server. If server mode is desired, the ***Act as Server*** check box must be checked first. Then clicking the ***Start Session*** Button will start an OBEX server session.

6.7.2.1 Check Headers Module

The main purpose of this module is to demonstrate the use of all methods offered by the DK classes CObexHeaders and CobexUserDefined.

This module also validates the methods are consistent. Module checkheaders defines a class ObexCheckHeaders. This class has only two methods, *Fill()* and *Verify()*. These methods are called from the ConnectionMgrDlg module as an automatic part of the startup sequence for the BlueObex application.

Method *Fill()* populates every headers structure in a CObexHeaders object. The return codes from the CobexHeaders methods are checked for error. The first time an error is detected *Fill()* returns with an error message indicating which header and which method reported a problem. If all headers are filled successfully, an 'OK' message is returned. The message returned from *Fill()* is reported by ConnectionMgrDlg on the log window.

Method *Verify()* calls the CObexHeaders methods which read out the header values. These are compared with the original input used in *Fill()*. As with *Fill()*, the first error condition causes *Verify()* to stop and return a descriptive error message. If all headers are verified, *Verify()* proceeds to delete each header and to verify that the header is no longer accessible. Any error condition is reported as a return error message. If all headers are verified successfully, an 'OK' message is returned and reported on the user's log window.

The test values are defined as 'static' variables at the beginning of the checkheaders.cpp module. The BlueObex user can easily modify them and rebuild the sample application to check for special cases or variations.

6.7.2.2 OBEX Connection Function

Server mode:

The server initiates an SPP server connection for the service name defined in m_serviceNameForServer in application source file ConnectionMgrDlg.cpp. The DK is installed with the service name 'Widcomm DK OBEX Service'. After registering the service, the server then waits for a client to connect.

Client mode:

From the client side, when the user clicks the **Start Session** button, the application performs device inquiry and service discovery. This process is the same as that used for BlueChat and BlueTime applications, when run as clients. The only difference is that the GUID and service name are different.

A GUID has been defined for this sample application. See variable 'service_guid' in source file ConnectionMgrDlg.cpp. All servers offering service for this GUID value are displayed on the client user interface. For each discovered service, the server device name and the service name are displayed. The user then selects one of the services, and an SPP connection to that server is attempted.

When the connection is established, the client sends a pre-defined sequence of OBEX objects to the server and monitors the responses.

Common:

For both the client and the server side the user interface shows a status line and a scrolling event log.

The status line reports major events such as connection , close, abort.

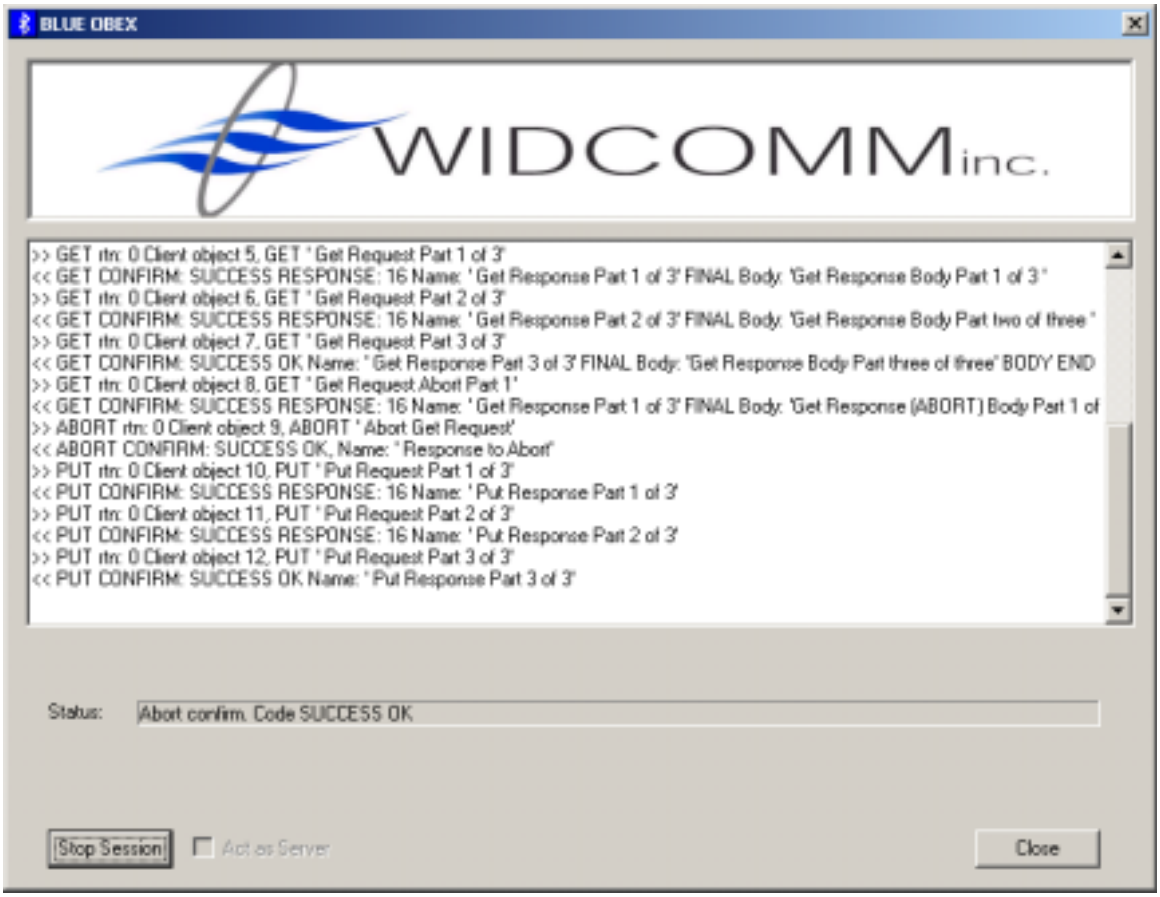
The event log reports each object sent, as '>> …', and each object received, as '<< …'.

Once a connection is established, the **Start Session** button is labeled **Stop Session**. The user can then stop the session as desired from the client or the server side.

The **Close** button exits the application.

The pre-defined set of objects included in the ConnectionMgrDlg.ccp module, under labels, m_client_list and m_server_list. The list contains a simple Get, a simple Put, a SetPath request, a Create Empty File request, a Delete File request, a multi-part Get request, a multi-part Get which is aborted, and a multi-part Put request.

Figure 18: BlueObex after Object Transfer

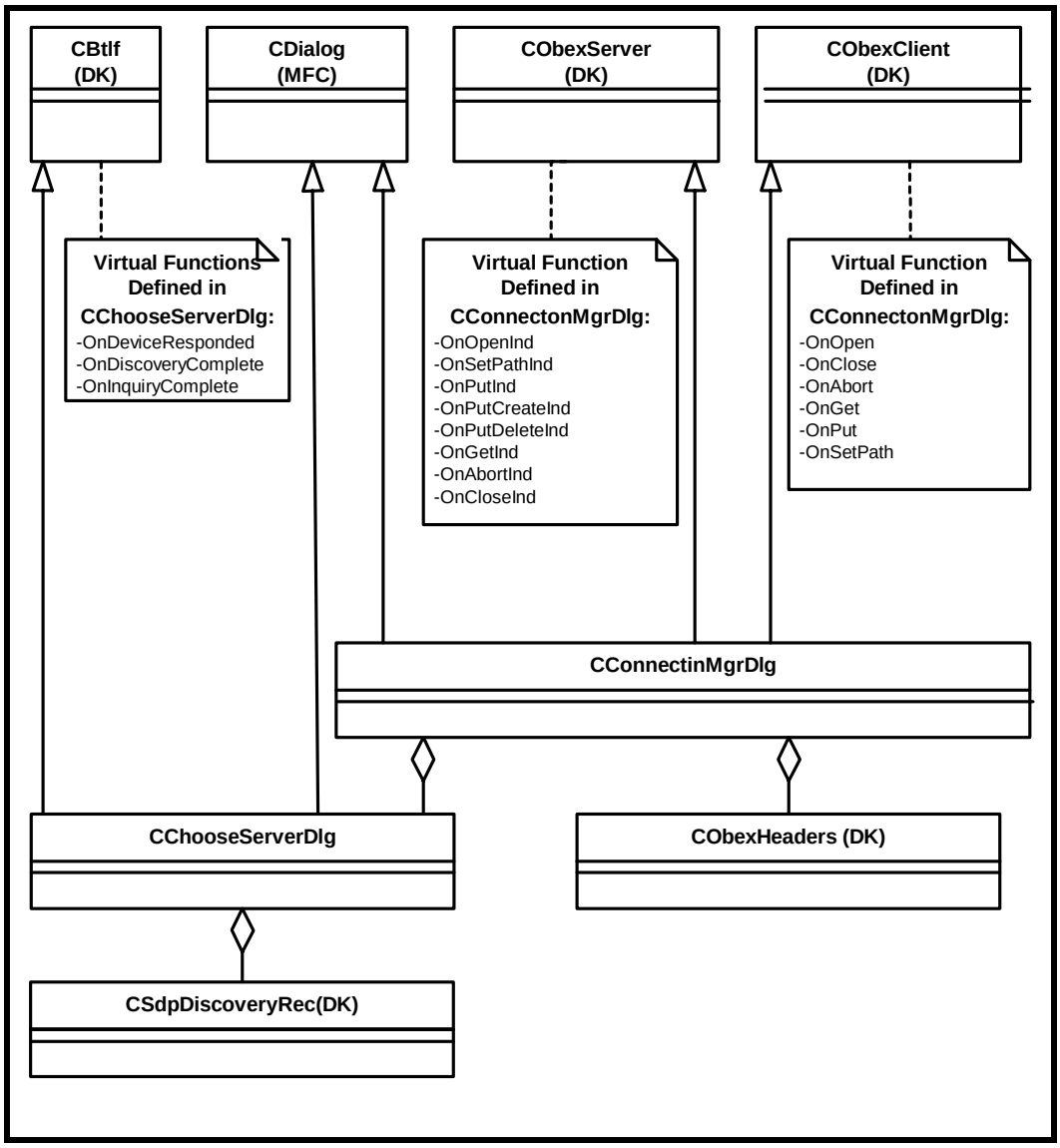


This list can be easily modified for exercise other OBEX sequences. Each object sent by the client must have a corresponding object for the server to respond with.

6.7.3 Key Class Descriptions

The following diagram shows the relationships between the DK classes and the primary application class, CConnectionMgrDlg.

Figure 19: BlueObex—the relationships between the DK classes and the primary application class, CConnectionMgrDlg.



CConnectionMgrDlg – This class implements the dialog functionality for the Connection Manager window. It is derived from the CObexClient and CObexServer classes defined in the DK.

CObexClient – provides the client-side functions for an OBEX connection.

CObexServer – provides the server-side functions for OBEX connections.

CObexHeaders – a container for the various OBEX 'header' structures that may be needed to control an OBEX transfer.