ClientPut

This command will initiate an OBEX Put operation in the remote device for the object defined in the FileName parameter.

Command Parameters	Examples	Comments
Filename	"C:\VCard.vcf"	

Return Events
ClientPut_Complete
ClientPut_Error

ClientSetPath

This command will initiate an OBEX SetPath operation in the remote device. Flags indicate SetPath option such as Backup.

Command Parameters	Examples	Comments
Path	"C:\OBEX"	
Flags	0x00	Bit 0: Back up a level before applying name (equivalent to ../ on many systems) Bit 1: Don't create an directory if it not exist. Returns an error instead.

Return Events
OBEX_Command_Complete
ClientSetPath_Error

ServerDeinit

This command will deinitialize the OBEX server.

Command Parameters	Examples	Comments
N/A		

Return Events
ServerDeinit_Complete
ServerDeinit_Error

ServerInit

This command will initialize the OBEX server.

Command Parameters	Examples	Comments
N/A		

Return Events
ServerInit_Complete
ServerInit_Error

ServerSetPath

Sets the path where received OBEX files are stored.

Command Parameters	Examples	Comments
Path	"C:\OBEX"	Use the "..." button to select a path, or type one in.

Return Events
ServerSetPath_Complete
ServerSetPath_Event
ServerSetPath_Error

Appendix B: Command Generator Examples

This chapter provides fourteen Command Generator examples. These examples consist of command sequences that are presented in order to illustrate useful scenarios. Please note that these examples do not cover all possible alternatives.

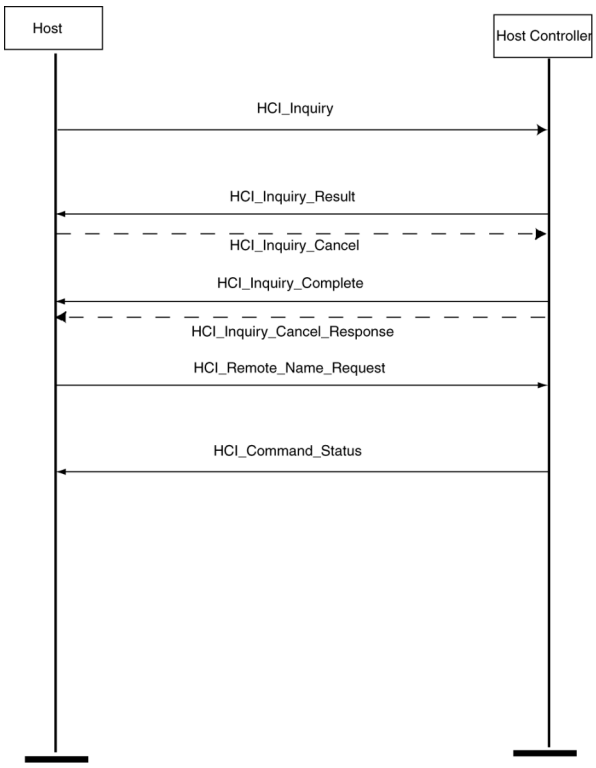
- Device Discovery and Remote Name Request
- Establish Baseband Connection
- Baseband Disconnection
- Create Audio Connection
- Establish L2CAP Connection
- L2CAP Channel Disconnect
- SDP Profile Service Search
- SDP Reset Database and Add Profile Service Record
- RFCOMM Client Channel Establishment
- RFCOMM Client Channel Disconnection
- RFCOMM Register Server Channel and Accept Incoming Connection
- Establish TCS Connection
- OBEX Server Init and Accept Incoming Connection
- OBEX Client Connection and Client Get & Put

Each example is illustrated with a diagram that shows communications between a host and host controller.

Notation used in this chapter:

- **Hexagon** = Condition needed to start the transaction.
- **Solid Arrow** represents a command.
- **Dashed Arrow** represents optional command.
- **Host** = Merlin's Wand application
- **Host Controller** = Merlin's Wand device

B.1 Device Discovery and Remote Name Request



Procedure

In this scenario, Merlin's Wand performs a General Inquiry and a Remote Name Request.

Step 1 Select HCI tab.

Step 2 Select **Inquiry** from the menu. You can use the default settings for the Inquiry_Length (8 seconds) and Num_Responses (10).

Step 3 Click **Execute.**

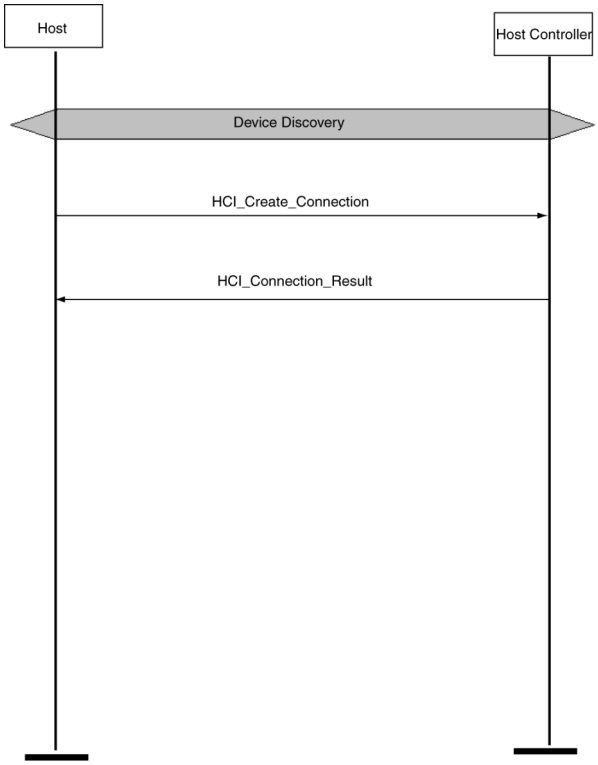
The Event Log should display an Inquiry_Result for each found device followed by an Inquiry_Complete event.

Step 4 Select **Remote_Name_Request** from the menu. Select the target device from the BD ADDR drop-down menu. You can use the default settings for the other drop-down menus.

Step 5 Click **Execute**.

The target device should then respond to the command with its name.

B.2 Establish Baseband Connection



Procedure

In this scenario, Merlin's Wand creates a Baseband (ACL) Connection.

This procedure assumes that Device Discovery has already been performed. See "Device Discovery and Remote Name Request" on page 138.

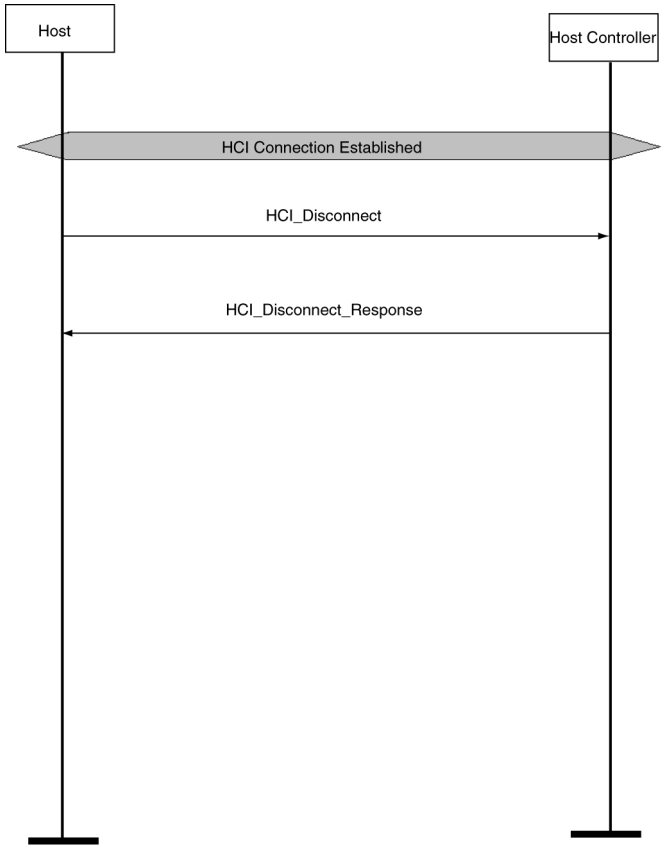
Step 1 From the HCI menu select **Create_Connection**.

Step 2 Select the target device from the BD_ADDR drop-down menu or enter a new BD_ADDR.

Step 3 Click **Execute**.

The Event Log should display a Connection_Complete or Connection_Failed response.

B.3 Baseband Disconnection



Procedure

In this scenario, Merlin's Wand terminates a Baseband connection. These steps continue the connection you established in the preceding scenario.

This procedure assumes that an ACL connection exists. See "Establish Baseband Connection" on page 139.

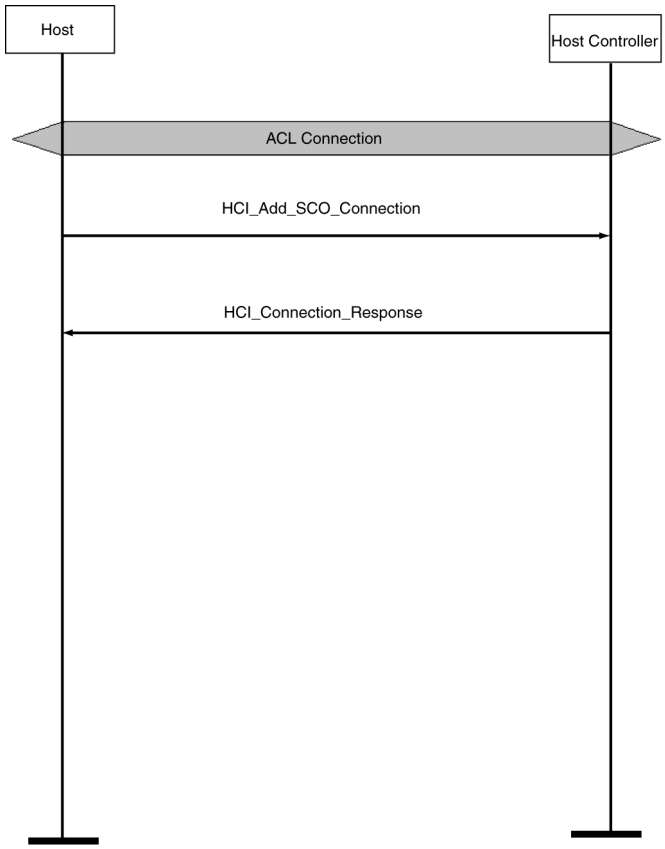
Step 1 From the HCI menu, select **Disconnect**.

Step 2 From the HCI_Handle drop-down menu, select a handle.

Step 3 Click **Execute**.

For each Disconnect, you should see a return event in the Event Log that indicates the outcome of the command.

B.4 Create Audio Connection



Procedure

In this scenario, Merlin's Wand creates an SCO connection.

This procedure assumes that you have established an ACL connection between Merlin's Wand and the target device. See "Establish Baseband Connection" on page 139.

Step 1 From the HCI menu, select Add_SCO_Connection from the menu.

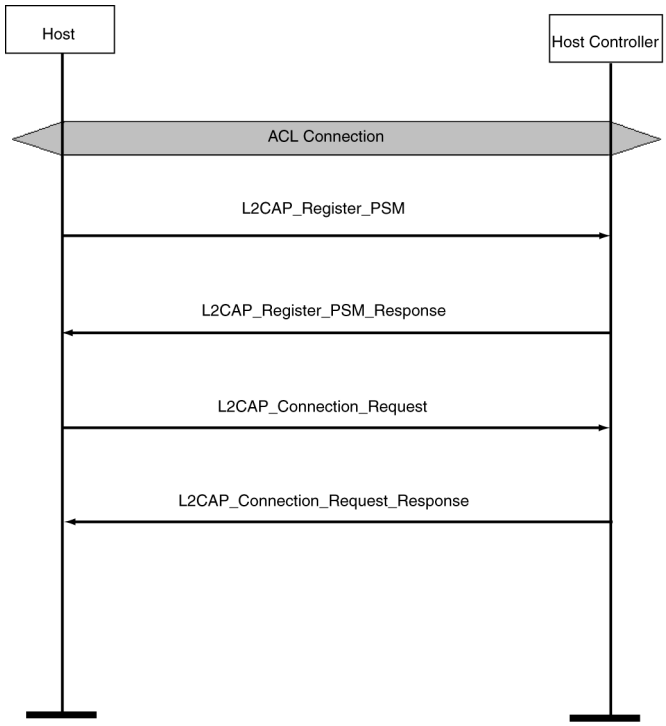
Step 2 Select the HCI_Handle for the previously established Baseband connection (for example, 0x0001) from the HCI_Handle parameter drop-down menu.

Step 3 Select a packet type from the Packet Type parameter drop-down menu.

Step 4 Click **Execute**.

The Event Log should indicate that the command was executed and that the target device responded with "Add_SCO_Connection_Complete" or "Add_SCO_Connection_Error."

B.5 L2CAP Connection



Procedure

In this scenario, Merlin's Wand establishes an L2CAP connection.

This procedure assumes that an ACL connection has been established. See "Establish Baseband Connection" on page 139.

Step 1 Click the L2CAP tab to display the L2CAP drop-down menu.

Step 2 Select **Register_PSM** from the menu.

Merlin's Wand must register its PSM channel before it can form an L2CAP connection.

Step 3 Select or type a PSM from the PSM menu.

Step 4 Select or type the Receive MTU from the Receive MTU menu (default value can be used).

Step 5 Click **Execute**.

Step 6 Repeat this procedure for the target device. The target device must also select a PSM.

Event Log should register "RegisterPsm_Complete."

Step 7 Select ConnectRequest from the L2CAP menu.

Step 8 Select an HCI Handle from the HCI_Handle drop-down menu.

To determine which HCI_Handle value is correct, open the **Piconet** window on the far left side of the Merlin's Wand application.

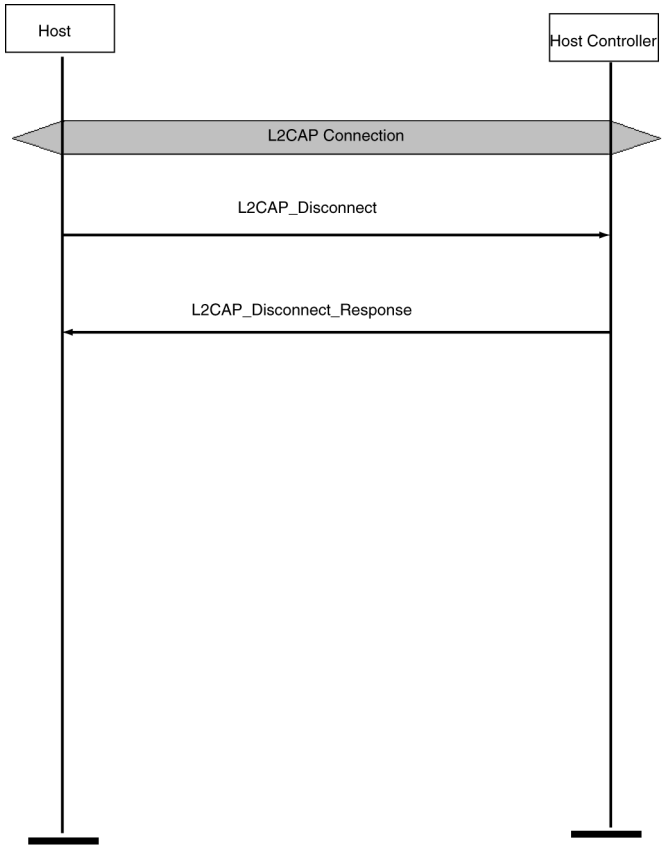
Step 9 Select or type a PSM from the PSM menu.

Step 10 Select or type the Receive MTU from the Receive MTU menu (default value can be used).

Step 11 Click **Execute**.

The Event Log should indicate that the command was executed and that a connection has been established.

B.6 L2CAP Channel Disconnect



Procedure

In this scenario, Merlin's Wand terminates an L2CAP connection.

This procedure assumes that an L2CAP connection has been established. See "L2CAP Connection" on page 142.

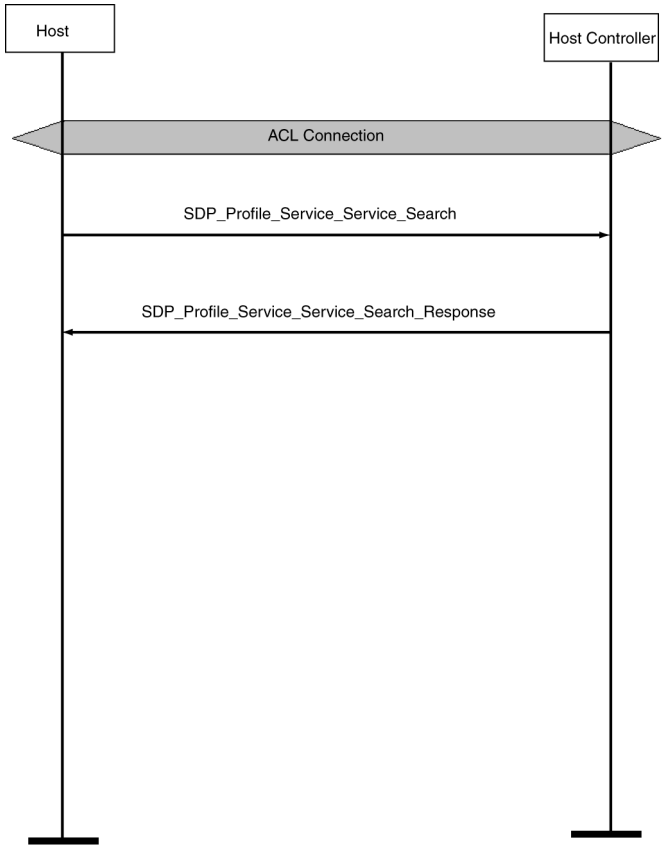
Step 1 From the L2CAP menu, select DisconnectRequest.

Step 2 Select the appropriate CID from the CID menu.

Step 3 Click **Execute**.

The Event Log should indicate that the command was successfully completed, with "Disconnection_Complete."

B.7 SDP Profile Service Search



Procedure

In this scenario, Merlin's Wand conducts a Profile Service Search.

This procedure assumes that an ACL connection has been established. "Establish Baseband Connection" on page 139.

Step 1 Click the SDP tab to display the SDP menu.

Step 2 Select ProfileServiceSearch from the menu.

Step 3 Select an HCI Handle from the HCI_Handle drop-down list.

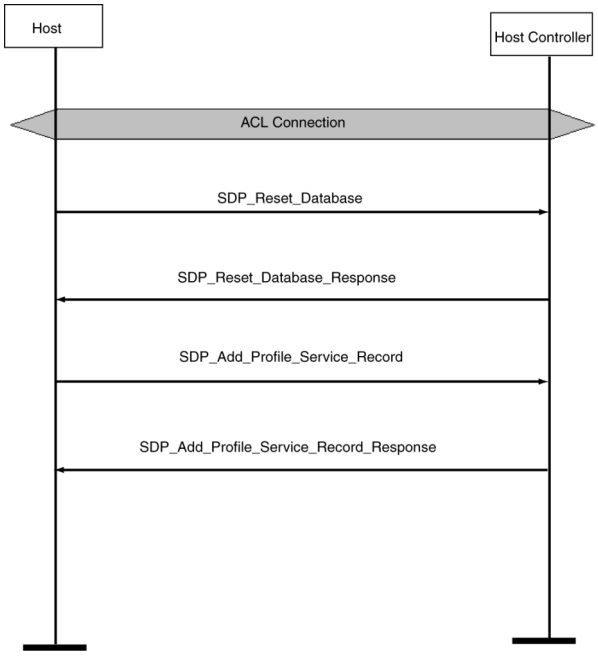
You can determine the HCI Handle from the **Piconet** window.

Step 4 Select a profile from the Profile menu.

Step 5 Click **Execute**.

The Event Log should return "ProfileServiceSearch_Complete," as well as the results of the search.

B.8 SDP Reset Database and Add Profile Service Record



Procedure

In this scenario, Merlin's Wand resets the SDP database and then adds an SDP Profile Service Record.

This procedure assumes that an ACL connection has been established between Merlin's Wand and the target device. "Establish Baseband Connection" on page 139.

Note A connection is not necessary to perform a Reset_Database or AddProfileServiceRecord.

Step 1 From the SDP tab select **ResetDatabase**.

Step 2 Click **Execute**.

The Event Log should indicate that the database was reset.

Step 3 Select **AddProfileServiceRecord** from the menu.

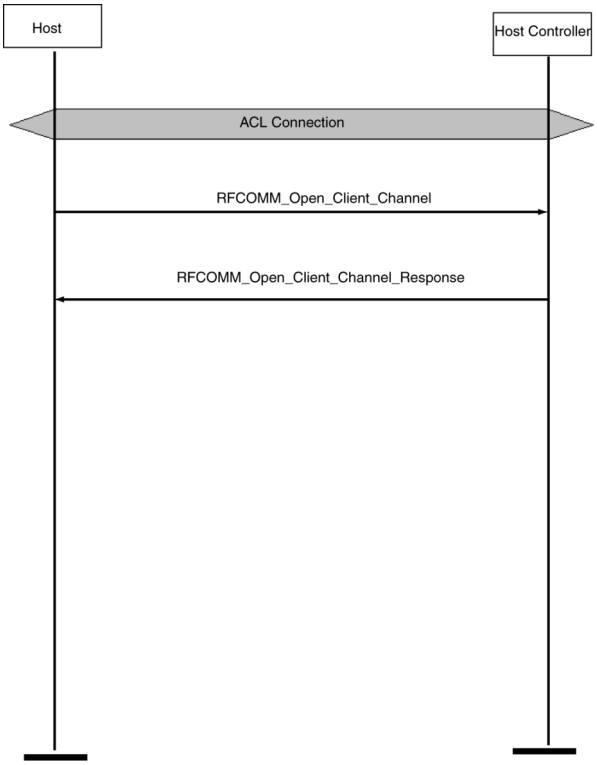
Step 4 Select a profile from the Profile menu.

Step 5 Select a server channel from the Channel menu.

Step 6 Click **Execute**.

Success will be indicated in the Event Log with "AddProfileServiceRecord_Complete" and the type of profile added.

B.9 RFCOMM Client Channel Establishment



Procedure

In this scenario, Merlin's Wand opens an RFCOMM client channel.

This procedure assumes that an ACL connection has been established and that the target device has assumed the role of an RFCOMM server. See "Establish Baseband Connection" on page 139.

Step 1 Click the RFCOMM tab to open the RFCOMM drop-down menu.

Step 2 Select **OpenClientChannel** from the menu.

Step 3 Select an HCI Handle from the HCI_Handle drop-down list.

Step 4 Select a Server Channel from the ServerChannel menu.

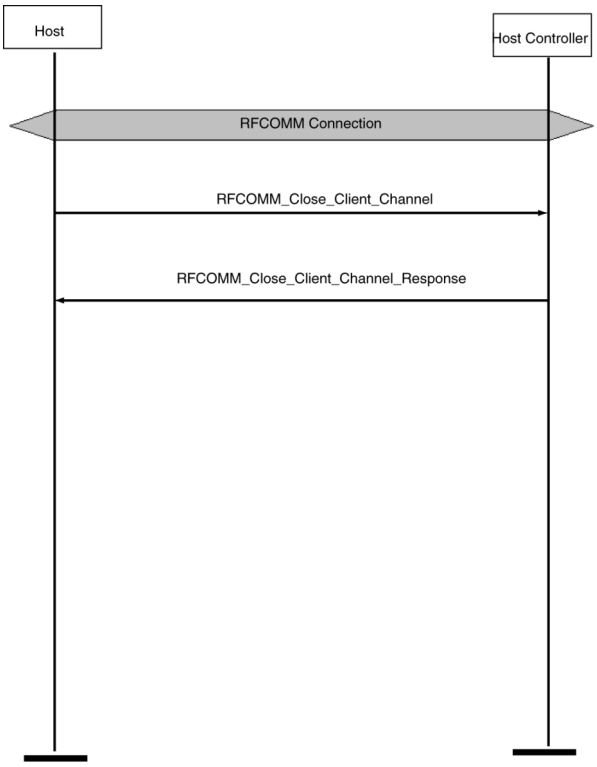
Step 5 Select a Max Frame Size from the MaxFrameSize menu.

Step 6 Select the number of credits from the Credit menu.

Step 7 Click **Execute**.

"OpenClientChannel_Complete" in the Event Log indicates that a client channel was successfully opened.

B.10 RFCOMM Client Channel Disconnection



Procedure

In this scenario, Merlin's Wand closes an RFCOMM client channel.

This procedure assumes that an RFCOMM channel has been established. See "RFCOMM Client Channel Establishment" on page 147.

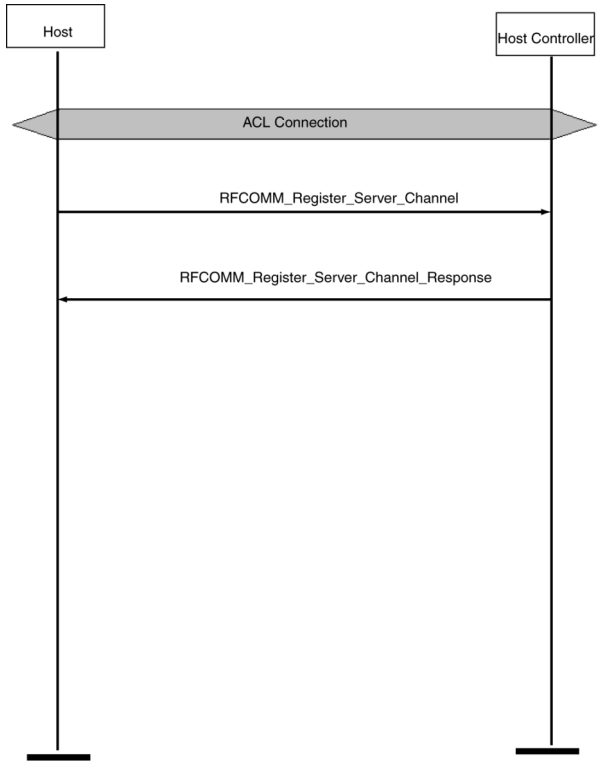
Step 1 From the RFCOMM menu select **CloseClientChannel**.

Step 2 Select the HCI/DLCI combination from the (HCI/DLCI) menu.

Step 3 Click **Execute**.

The Event Log should indicate a response to the command such as "CloseClientChannel_Complete."

B.11 RFCOMM Register Server Channel



Procedure

In this scenario, Merlin's Wand registers a Server Channel.

This procedure assumes that an ACL connection has been established. See "Establish Baseband Connection" on page 139.

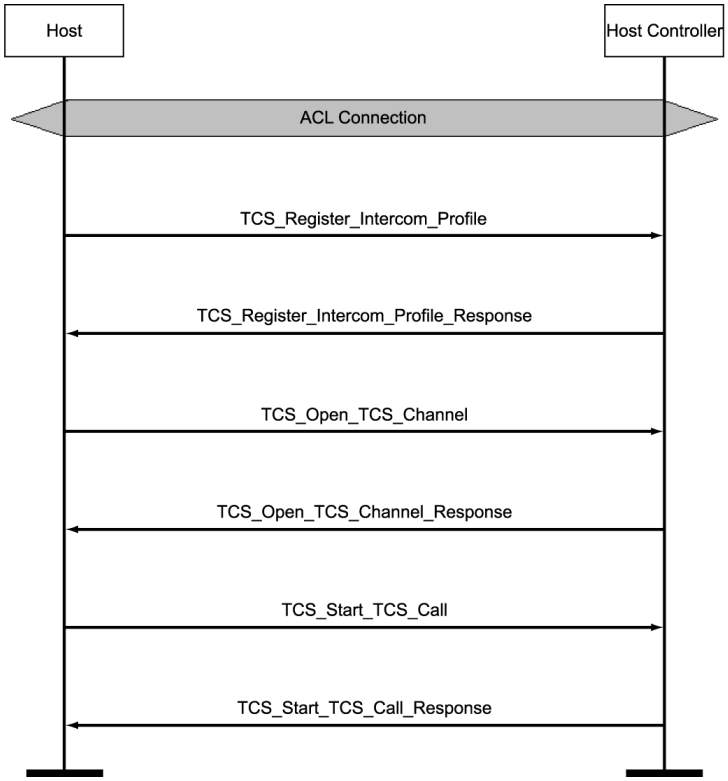
Note A connection is not necessary to call a RegisterServerChannel command.

Step 1 From the RFCOMM menu select **RegisterServerChannel**.

Step 2 Click **Execute**.

The Event Log should indicate successful completion of the command with the response "RegisterServerChannel_Complete." On completion of these steps, the application is ready to accept incoming RFCOMM connections.

B.12 Establish TCS Connection



Procedure

In this scenario, Merlin's Wand establishes a TCS connection.

This procedure assumes that an ACL connection has been established. See "Establish Baseband Connection" on page 139.

Step 1 Click the **TCS tab** to display the TCS drop-down menu.

Step 2 Select **Register_Intercom_Profile** from the menu.

Merlin 's Wand must register its Intercom profile before it can form a TCS connection.

Step 3 Click **Execute**.

The Event Log should display "Register_Intercom_Profile_Complete."

Step 4 Repeat Steps 1-3 for the target device.

Step 5 Select **Open_TCS_Channel** from the menu and select an HCI handle.

Merlin 's Wand must create an ACL connection before it can form a TCS connection.

Step 6 Click **Execute**.

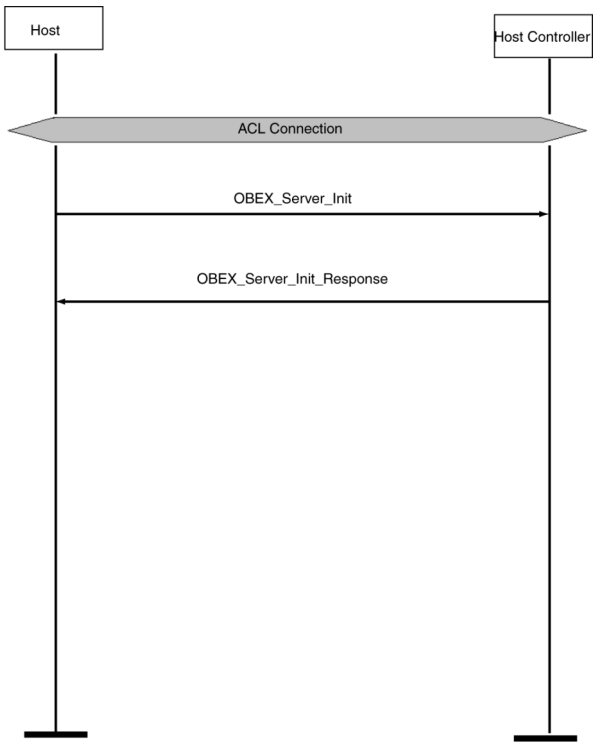
Event Log should display "Open_TCS_Channel_Complete."

Step 7 Select **Start_TCS_Call** from the menu.

Step 8 Click **Execute**.

Event Log should display "Start_TCS_Call_Complete."

B.13 OBEX Server Init



Procedure

In this scenario, Merlin's Wand initializes itself as an OBEX server. This scenario assumes that an ACL connection exists. See "Establish Baseband Connection" on page 139.

Note A connection is not necessary to call an OBEX ServerInit function.

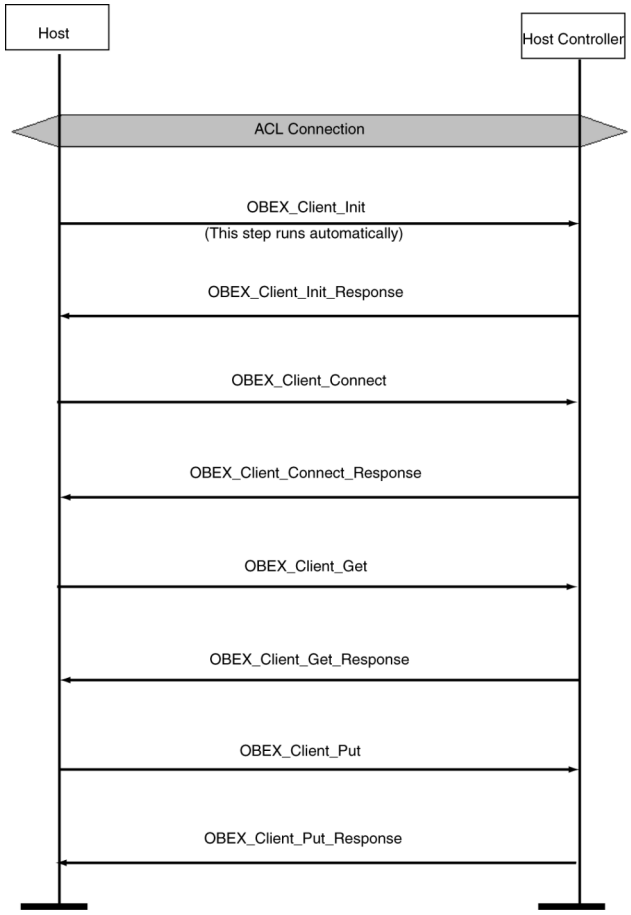
Step 1 Click the OBEX tab to display the OBEX menu.

Step 2 Select **ServerInit** from the menu.

Step 3 Click **Execute**.

On completion of these steps, the application is ready to accept an incoming OBEX connection.

B.14 OBEX Client Connection and Client Get & Put



Procedure

In this scenario, Merlin's Wand forms a client connection with the target device and then retrieves a text file from the target and sends one to it.

This procedure assumes that an ACL connection has been established (see "Establish Baseband Connection" on page 139). It also assumes that the target device has been configured as an OBEX server.

Note: When the OBEX window is first opened, Merlin's Wand will automatically run an OBEX_ClientInit command and initiate itself as an OBEX client. This means that you do not have to manually execute a ClientInit command at the start of this procedure.

Step 1 Click the OBEX tab to display the OBEX menu.

Step 2 Select **ClientConnect** from the menu.

Step 3 Select the target device from the **BD_ADDR** parameter menu.

Step 4 Click **Execute**.

The Event Log should indicate that a connection was established.

Step 5 Select **ClientGet** from the command menu.

Step 6 Type in the name of a file that is to be transferred from the Server into the **Object** parameter box.

Step 7 Click **Execute**.

The Event Log should indicate that the file was transferred. A Save As dialog box should open.

Step 8 Enter a name for the file you are retrieving. Select a directory, then click **Save**.

Step 9 Select **ClientPut** from the command menu.

Step 10 In the box marked **Filename,** type the name of a file that is to be transferred to the server, or use the browse ... button to locate the file you want to put. The Open dialog will come up, allowing you to navigate to the desired file.

Step 11 Click **Execute**.

The Event Log should indicate that the file was transferred.

Appendix C: Merlin's Wand Scripting Commands

Merlin's Wand supports scripting commands to help automate testing processes and commonly used sequences of Bluetooth commands. Custom scripts can be written, saved, and run in Script Manager.

C.1 Bluetooth Addresses

Bluetooth addresses are represented in scripts as binary strings in big-endian byte order. For example, the Bluetooth address "0x010203040506" would be represented in the script as:

```
DeviceAddress = '010203040506';
```

Comparisons can be performed using binary strings. For example:

```
if ( DeviceAddress == '010203040506' )
{
   #do something based on comparison here
}
```

C.2 Basic Commands

`Main()`

`Main()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

None.

Comments

This is the entry point into a script. When a script is run, the script's Main() function will be called. Include this command at the beginning of every script.

Example

```
Main()
{
```

```
    #include body of script here
}
```

Clock()

`Clock()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

The number of milliseconds that have elapsed since the system was started.

Comments

This function returns the amount of time that the system has been running. It can be used to find out how long it takes to run a script or a series of commands within a script.

Example

```
time1 = Clock();
# Put script commands here
time2 = Clock();
Trace("Elapsed time is ", time2-time1, "\n");
```

Connect()

`Connect(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with		

Return value

- "Success"
- "Already connected"
- "Timed out"
- "Failed: Disconnection in progress"
- "Failure"

Comments

Establishes an ACL connection with the specified device

Example

```
result = Connect(Devices[0]);
if(result != "Success")
{
   MessageBox("Failed to connect!");
}
```

Disconnect()

`Disconnect(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with		

Return value

- "Success"
- "Failure"
- "Timed out"

Comments

Closes the ACL connection with the specified device

Example

```
result = Disconnect(Devices[0]);
if(result != "Success")
{
   MessageBox("Failed to disconnect!");
}
```

DoInquiry()

`DoInquiry(IAC, Timeout)`

Parameter	Meaning	Default Value	Comments
IAC	Inquiry Access Code	GIAC	"GIAC", or a 32-bit integer value
Timeout	Timeout in units of 1.2 seconds	5	

Return value

Array of Bluetooth addresses that were found during the inquiry.

Comments

Calling DoInquiry() will block for the duration specified by *Timeout*. The function returns an array of devices that were found during the inquiry. These can be addressed individually.

The current version of Merlin's Wand hardware only supports GIAC inquiries.

Example

```
# Use default parameters
Devices = DoInquiry();
Trace("First device was: ", Devices[0]);
```

GetDeviceClass()

`GetDeviceClass()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- Class of device
- "Failure"

Comments

Returns the current device class of Merlin's Wand

Example

```
Trace("Merlin's Wand device class: ", GetDeviceClass());
```

GetRemoteDeviceName()

`GetRemoteDeviceName(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		

Return value

- Device name
- "Not connected"
- "Failure"

Comments

Queries the specified device for its name.

An ACL connection must be established before calling GetRemoteDeviceName().

Example

```
Trace("Device ", Devices[0], "is named ",
GetRemoteDeviceName(Devices[0]));
```

MessageBox()

`MessageBox(Message, Caption)`

Parameter	Meaning	Default Value	Comments
Message	Text to display in the message box		
Caption	Caption of the message box	"Script Message"	

Return value

None.

Comments

Bring up a simple message box function with one "OK" button. This is a good way to pause execution of the script or indicate errors.

Example

```
MessageBox("Failed to connect", "Connection Failure");
```

SetDeviceClass()

`SetDeviceClass(Class)`

Parameter	Meaning	Default Value	Comments
Class	Device class for Merlin's Wand		Device class is a 3-byte value

Return value

- "Success"
- "Failure"

Comments

Sets the device class of Merlin's Wand

Example

```
SetDeviceClass(0x010203);
```

Sleep()

`Sleep(Time)`

Parameter	Meaning	Default Value	Comments
Time	Time in ms		

Return value

None.

Comments

Delays program execution for Time in milliseconds.

Example

```
Sleep(1000); # Sleep for one second
```

C.3 Pipe Commands

ClosePipe()

`ClosePipe(PipeName, PipeType)`

Parameter	Meaning	Default Value	Comments
PipeName	Name of the data pipe to open		
PipeType	"Transmit" or "Receive" pipe	"Receive"	

Return value

- "Success"

- "Failure"
- "Pipe Not Found"
- "Invalid parameter"

Comments

Closes the specified data pipe.

Example

```
ClosePipe("Data1", "Receive");
```

DeletePipe()

`DeletePipe(PipeName, PipeType)`

Parameter	Meaning	Default Value	Comments
PipeName	Name of the data pipe to open		
PipeType	"Transmit" or "Receive" pipe	"Receive"	

Return value

- "Success"
- "Invalid parameter"
- "Pipe not found"

Comments

Removes a pipe from the Data Transfer Manager pipe list. In the case of "Receive" pipes, all data associated with the pipe is lost. "Transmit" pipes will only be removed from the Data Transfer Manager list.

Example

```
DeletePipe("Data1", "Receive");
```

OpenPipe()

`OpenPipe(PipeName, PipeType)`

Parameter	Meaning	Default Value	Comments
PipeName	Name of the data pipe to open		

Parameter	Meaning	Default Value	Comments
PipeType	"Transmit" or "Receive" pipe	"Receive"	

Return value

- "Success"
- "Failure"
- "Pipe Not Found"

Comments

Opens a data pipe for reading or writing. If the data pipe type is "Receive" and the pipe does not exist, a new pipe will be created. All open pipes will be automatically closed upon script termination.

Example

```
OpenPipe("Data1", "Receive");
```

ReadPipe()

`ReadPipe(PipeName, PipeType, ByteCount)`

Parameter	Meaning	Default Value	Comments
PipeName	Name of the data pipe to open		
PipeType	"Transmit" or "Receive" pipe		
ByteCount	Number of bytes to read	1-65535	

Return values

Returns a list with three values: *result, bytes read,* and *data*.

Result (element 0) is one of the following:

- "Success"
- "Failure"
- "Invalid parameter"
- "Pipe not found"
- "Pipe not open"

Bytes read (element 1) is the number of bytes read in this transaction. Valid only if result is "Success".

Data (element 2) is the raw data received in the transaction. Valid only if result is "Success".

Comments

Reads the specified amount of data from an open pipe.

Example

```
result = ReadPipe("Data1", "Receive", 1024);
if(result[0] == "Success")
{
   Trace("Read ", result[1], "bytes:\n");
   Trace(result[2]);
}
```

WritePipe()

`WritePipe(PipeName, Data)`

Parameter	Meaning	Default Value	Comments
PipeName	Name of the data pipe to open		
Data	Data to write to the pipe		This can be a string, integer, list or raw data.

Return value

- "Success"
- "Failure"
- "Pipe not found"
- "Pipe not open"
- "Not supported"

Comments

Writes data to the specified pipe. Note that only "Receive" pipes can be written to.

Example

```
result = WritePipe("Data1", "This is a string written to a
pipe");
result = WritePipe("Data1", '3C7EFFFF7E3C');
result = WritePipe("Data1", 0x01020304);
```

C.4 HCI Commands

HCIAcceptConnectionRequest()

`HCIAcceptConnectionRequest()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"

Comments

Sets the accept connection request variable to True.

Example

```
status = HCIAcceptConnectionRequest();
```

HCIAddSCOConnection()

`HCIAddSCOConnection(Address, Type)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with		
Type	Type of SCO connection to establish	["HV3"]	A list of one or more of the following strings: "DM1", "HV1", "HV3" or "DV"

Return value

- "Success"
- "Not connected"
- "Not supported"
- "Failure"

Comments

Attempts to establish an SCO connection with the specified device.

An ACL connection must already be established with the device before calling HCIAddSCOConnection.

Example

```
result = HCIAddSCOConnection(Devices[0], ["DM1", "HV1"]);
if(result != "Success")
{
   MessageBox(result, "Failed to add SCO connection!");
}
```

HCIAuthenticationRequested()

`HCIAuthenticationRequested(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to authenticate with		A connection must exist with this device for an authentication request to work.

Return value

- "Success"
- "Failure"
- "Failed: Device not found"
- "Timed Out"
- "Not connected"

Comments

Attempts to authenticate an existing link with the specified device.

Example

```
result = HCIAuthenticationRequested (Devices[0]);
if(result != "Success")
{
   MessageBox(result, "Failed to authenticate!");
}
```

HCICatcBerTest()

`HCICatcBerTest(Address, NumberOfPackets, BERPacketType, TestDataType, TestData, BERInterval)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of the remote device		

Parameter	Meaning	Default Value	Comments
NumberOf Packets	The number of baseband packets to be transmitted		0x0000 - an unlimited number of packets 0x0001 - 0xFFFF - number of packets
BERPacket Type			0x00 - DH1 0x01 - DH3 0x02 - DH5 0x03 - DM1 0x04 - DM3 0x05 - DM5
TestDataType			0x00 - send Bluetooth test mode PRBS 0x01 - every octet equals TestData
TestData			Data to send
BERInterval	A packet is sent every BERInterval frame		

Return value

- "Success"
- "Failure"
- "Not connected"

Comments

This command is used to measure BER when fully loaded baseband packets are sent from master to slave on the link.

Example

```
Trace("Test Control : ",
HCICatcTestControl(Address,1,1,2,2,4), "\n");
Trace("Enter Test Mode : ", HCICatcEnterTestMode(Address),
"\n");
Trace("BER Test : ", HCICatcBerTest(Address,1,3,0,0xFF,10),
"\n");
```

HCICatcChangeHeadsetGain()

`HCICatcChangeHeadsetGain(Device, Gain)`

Parameter	Meaning	Default Value	Comments
Device	Speaker or microphone		Values: "Speaker", "Microphone"

Parameter	Meaning	Default Value	Comments
Gain	New gain of the device	Values: 0 – 0xF	

Return value

- "Success"
- "Failure"
- "Not connected"
- "No SCO connection"

Comments

This command is used to change gain of connected speaker or microphone. In order to use this command, an SCO connection must exist.

Example

```
Main()
{
   result = Connect('00803713BDF0');
   Trace("Connection result : ", result, "\n");
   if( result == "Success")
   {
      result = HCIAddSCOConnection( '00803713BDF0',
["HV1"]);
      Trace("SCO Connection result : ", result, "\n");
      if( result == "Success")
      {
         index = 0;
         while(index < 16)
         {
            result = HCICatcChangeHeadsetGain("Speaker",
index);
            Trace("Change speaker gain: ", result, "\n");
            result = HCICatcReadHeadsetGain("Speaker",
index);
            Trace("Read speaker gain: ", result, "\n");
            index = index + 1;
            Sleep(2000);
         }
         index = 0;
         while(index < 16)
         {
            result =
HCICatcChangeHeadsetGain("Microphone", index);
            Trace("Change microphone gain: ", result,
"\n");
            result = HCICatcReadHeadsetGain("Microphone");
            Trace("Read microphone gain : ", result, "\n");
            index = index + 1;
```

```
            Sleep(2000);
         }
         status = HCIRemoveSCOConnection('00803713BDF0');
         Trace("SCO disconnect result: ", status, "\n");
      }
      status = Disconnect('00803713BDF0');
      Trace("Disconnect result: ", status, "\n");
   }
}
```

HCICatcEnterTestMode()

`HCICatcEnterTestMode(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of remote device		

Return value

- "Success"
- "Failure"
- "Not found"
- "Not connected"

Comments

This command is used to put the remote device identified by its Bluetooth address into test mode.

Example

See the example for HCICatcBerTest command.

HCICatcReadHeadsetGain()

`HCICatcReadHeadsetGain(Device)`

Parameter	Meaning	Default Value	Comments
Device	Speaker or microphone		Values: "Speaker", "Microphone"

Return values

Returns a list with two values: *status* and *gain*.

Status (element 0) is one of the following:

- "Success"
- "Failure"
- "Not found"
- "No SCO connection"

Gain (element 1) is the one-byte value of the headset gain. Range is 0 to 15.

Comments

This command is used to read current gain of connected speaker or microphone. In order to use this command, an SCO connection must exist.

Example

See the example for the HCICatcChangeHeadsetGain command.

HCICatcReadRevisionInformation()

`HCICatcReadRevisionInformation()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *revision*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Revision (element 1) is the Merlin's Wand revision information.

Comments

This command returns the information about the current firmware.

Example

```
Revision = HCICatcReadRevisionInformation();
if( Revision[0] == "Success")
Trace("Merlin's Wand Revision Info : ", Revision[1], "\n");
```

HCICatcSelfTest()

`HCICatcReadRevisionInformation()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

This command is used to perform a self test on a local device.

Example

```
Trace("Merlin's Wand Self Test : ", HCICatcSelfTest(),
"\n");
```

HCICatcTestControl()

`HCICatcTestControl(Address, TestScenario, HoppingMode, TxFrequency, RxFrequency, TestPacketType)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of the remote device		
TestScenario			
HoppingMode			
TxFrequency			
RxFrequency			
TestPacket Type			0x00 - DH1 0x01 - DH3 0x02 - DH5 0x03 - DM1 0x04 - DM3 0x05 - DM5

Return value

- "Success"
- "Failure"
- "Not found"

- "Not connected"

Comments

This command is used to start a specific test for the slave device identified by Bluetooth address. See Bluetooth LMP specification, page 246 for description of the parameters.

Example

See the example for HCICatcBerTest command.

HCICatcWriteCountryCode()

`HCICatcWriteCountryCode(CountryCode)`

Parameter	Meaning	Default Value	Comments
CountryCode			0x00 North America and Europe 0x01 France

Return value

- "Success"
- "Failure"

Comments

This command is used to define which range of the ISM 2.4 GHz frequency band will be used by the radio.

The device has to be reset after using this command.

Example

```
result = HCICatcWriteCountryCode(0);
Trace("Change CountryCode: ", result, "\n");
Trace("Don't forget to reset the device afterward!\n");
```

HCIChangeConnectionLinkKey()

`HCIChangeConnectionLinkKey(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of the remote		

Return value

- "Success"

- "Failure"
- "Not found"
- "Not connected"

Comments

This command is used to force both devices associated with a connection to generate a new link key.

Example

```
result = HCIChangeConnectionLinkKey('00803713BDF0');
Trace("Change Connection Link Key: ", result, "\n");
```

HCIChangeConnectionPacketType()

`HCIChangeConnectionPacketType(Address, PacketType)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with		
PacketType			"DH1" "DH3" "DH5" "DM1" "DM3" "DM5" "AUX1" "HV1" "HV2" "HV3" "DV"

Return value

- "Success"
- "Failure"
- "Not found"
- "Not connected"
- "Not supported"

Comments

This command is used to change which baseband packet type can be used for a connection

Example

```
result = HCIChangeConnectionPacketType('00803713BDF0',
["DM3","DM5"]);
Trace("Change Connection Packet Type:\n");
Trace(" Status            ",   result[0], "\n");
```

HCIChangeLocalName()

HCIChangeLocalName(Name)

Parameter	Meaning	Default Value	Comments
Name	String that contains the new name for the local device		

Return value

- "Success"
- "Failure"

Comments

Attempts to change the name of the local device.

Example

```
result = HCIChangeLocalName("Joe's Device");
if(result != "Success")
{
   MessageBox(result, "Failed to change name!");
}
```

HCIDeleteStoredLinkKey()

HCIDeleteStoredLinkKey(Address, DeleteAll)

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device that will have its link key deleted		

Parameter	Meaning	Default Value		Comments
DeleteAll	Boolean value that indicates whether to delete only the specified address's link key, or all link keys	0	0 or 1	

Return value

- "Success"
- "Failure"

Comments

Attempts to delete the stored link key for the specified address or for all addresses, depending on the value of DeleteAll.

Example

```
result = HCIDeleteStoredLinkKey('6E8110AC0008', 1);
if(result != "Success")
{
   MessageBox(result, "No link keys were deleted.");
}
```

HCIEnableDeviceUnderTestMode()

`HCIEnableDeviceUnderTestMode()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

This command will allow the local Bluetooth device to enter a test mode via LMP test commands

Example

```
result = HCIEnableDeviceUnderTestMode();
Trace("Enabled DUT : ", result, "\n");
```

HCIExitParkMode()

`HCIExitParkMode(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		

Return value

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Comments

Switches the current role of the device in the piconet.

Example

```
Device = '010203040506';
result = HCIExitParkMode(Device);
Trace("HCIExitParkMode result is: ", result, "\n");
```

HCIExitSniffMode()

`HCIExitSniffMode(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		

Return value

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Comments

Exits Sniff mode.

Example

```
Device = '010203040506';
```

```
result = HCIExitSniffMode(Device);
Trace("HCIExitSniffMode result is: ", result, "\n");
```

HCIHoldMode()

`HCIHoldMode(Address, MaxInterval, MinInterval)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		
MaxInterval	Maximum number of 0.625-msec intervals to wait in Hold mode.		Range is 0x0001 to 0xFFFF (0.625 msec to 40.9 sec).
MinInterval	Minimum number of 0.625-msec intervals to wait in Hold mode.		Range is 0x0001 to 0xFFFF (0.625 msec to 40.9 sec).

Return value

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Comments

Enters Hold mode with parameters as specified.

Example

```
Device = '010203040506';
result = HCIHoldMode(Device, 0xFFFF, 0x50);
Trace("HCIHoldMode result is: ", result, "\n");
```

HCIMasterLinkKey()

`HCIMasterLinkKey(KeyFlag)`

Parameter	Meaning	Default Value	Comments
KeyFlag			0x0 use semi-permanent link keys 0x1 use temporary link keys

Return values

Returns a list with three values: *status, HCI handle,* and *key flag.*

Status (element 0) is one of the following:

- "Success"
- "Failure"

HCI handle (element 1) is the handle for the ACL connection.

Key flag (element 2) is the key flag (either 0 or 1).

Comments

This command is used to force the master of the piconet to use temporary or semi-permanent link keys.

Example

```
result = HCIMasterLinkKey(0);
Trace("Merlin's Wand MasterLinkKey returned:", result[0],
"\n");
if(result[0] == "Success")
{
   Trace(" Connection Handle : 0x", result[1], "\n");
   Trace(" Key Flag          : 0x", result[2], "\n");
}
```

HCIParkMode()

`HCIParkMode(Address, BeaconMaxInterval, BeaconMinInterval)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		

Parameter	Meaning	Default Value	Comments
Beacon MaxInterval	Maximum number of 0.625-msec intervals between beacons.		Range is 0x0001 to 0xFFFF (0.625 msec to 40.9 sec).
Beacon MinInterval	Minimum number of 0.625-msec intervals between beacons.		Range is 0x0001 to 0xFFFF (0.625 msec to 40.9 sec).

Return value

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Comments

Enters Park mode with parameters as specified.

Example

```
Device = '010203040506';
result = HCIParkMode(Device, 0xFFFF, 0x100);
Trace("HCIParkMode result is: ", result, "\n");
```

HCIPINCodeRequestNegativeReply()

`HCIPINCodeRequestNegativeReply(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device for which no PIN code will be supplied.		

Return value

- "Success"
- "Failure"

Comments

Specifies a device for which no PIN code will be supplied, thus causing a pair request to fail.

Example

```
result = HCIPINCodeRequestNegativeReply('6C421742129F9');
Trace("HCIPINCodeRequestNegativeReply returned: ", result,
"\n");
```

HCIPINCodeRequestReply()

`HCIPINCodeRequestReply(Address, PINCode)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device for which PIN code will be used.		
PINCode	PIN code to use when connecting to the device.		Must be 1 to 16 characters in length.

Return value

- "Success"
- "Failure"

Comments

Specifies the PIN code to use for a connection.

Example

```
result = HCIPINCodeRequestReply('6C421742129F9', "New PIN
Code");
Trace("HCIPINCodeRequestReply returned: ", result, "\n");
```

HCIQoSSetup()

```
HCIQoSSetup(Address, ServiceType, TokenRate,
PeakBandwidth, Latency, DelayVariation)
```

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of the remote		
ServiceType	The one-byte service type: 0=No traffic; 1=Best effort; 2=Guaranteed		
TokenRate	The four-byte token rate value in bytes per second		
Peak Bandwidth	The four-byte peak bandwidth value in bytes per second		
Latency	The four-byte latency value in microseconds		
Delay Variation	The four-byte delay variation value in microseconds		

Return values

Returns a list with eight values: *status, HCI handle, flags, service type, token rate, peak bandwidth, latency,* and *delay variation.*

Status (element 0) is one of the following:

- "Success"
- "Failure"

HCI handle (element 1) is the handle for the ACL connection.

Flags (element 2) is a one-byte value reserved for future use.

Service type (element 3) is the one-byte service type. (0=No traffic; 1=Best effort; 2=Guaranteed.)

Token rate (element 4) is the four-byte token rate value in bytes per second.

Peak bandwidth (element 5) is the four-byte peak bandwidth value in bytes per second.

Latency (element 6) is the four-byte latency value in microseconds.

Delay variation (element 7) is the four-byte delay variation value in microseconds.

Comments

This command is used to specify Quality of Service parameters for the connection.

Example

```
QoS = HCIQoSSetup('00803713BDF0', 2, 0, 0, 0x12345678,
0x23456789);
Trace("Merlin's Wand Link QoS Setup returned: ", QoS[0],
"\n");
if (QoS[0] == "Success")
{
   Trace(" Connection Handle : 0x", QoS[1], "\n");
   Trace(" Flags             : 0x", QoS[2], "\n");
   Trace(" Service Type      : 0x", QoS[3], "\n");
   Trace(" Token Rate        : 0x", QoS[4], "\n");
   Trace(" Peak Bandwidth    : 0x", QoS[5], "\n");
   Trace(" Latency           : 0x", QoS[6], "\n");
   Trace(" Delay Variation   : 0x", QoS[7], "\n\n");
}
```

HCIReadAuthenticationEnable()

`HCIReadAuthenticationEnable()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *authentication enable*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Authentication enable (element 1) is the one-byte authentication enable value. (0=Authentication disabled; 1=Authentication enabled.)

Comments

This command will read the value for AuthenticationEnable parameter.

Example

```
result = HCIReadAuthenticationEnable();
if(result[0] == "Success")
Trace("Merlin's Wand Authentication Enabled : ", result[1],
"\n");
```

HCIReadBDADDR()

HCIReadBDADDR()

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *address*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Address (element 1) is the address of the local device.

Comments

This command is used to read the value for the BD_ADDR parameter. The BD_ADDR is a 48-bit unique identifier for a Bluetooth device.

Example

```
LocalAddress = HCIReadBDADDR();
if(LocalAddress [0] == "Success")
Trace("Local BDADDR:", LocalAddress[1], "\n");
```

HCIReadBufferSize()

HCIReadBufferSize()

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with five values: *status, ACL packet length, SCO packet length, ACL number of packets,* and *SCO number of packets.*

Status (element 0) is one of the following:

- "Success"
- "Failure"

ACL packet length (element 1) is the two-byte value of the maximum length (in bytes) of the data portion of each HCI ACL data packet that the Host Controller is able to accept.

SCO packet length (element 2) is the one-byte value of the maximum length (in bytes) of the data portion of each HCI SCO data packet that the Host Controller is able to accept.

ACL number of packets (element 3) is the total number of HCI ACL data packets that can be stored in the data buffers of the Host Controller.

SCO number of packets (element 4) is the total number of HCI SCO data packets that can be stored in the data buffers of the Host Controller.

Comments

This command is used to read the maximum size of the data portion of SCO and ACL data packets sent from the Host to Host Controller.

Example

```
Trace("Local Buffer parameters\n");
result = HCIReadBufferSize();
Trace(" HCIReadBufferSize() returned: ",   result[0],
"\n");
if (result[0] == "Success")
{
   Trace(" ACL Data Packet Length     : 0x", result[1],
"\n");
   Trace(" SCO Data Packet Length     : 0x", result[2],
"\n");
   Trace(" Total Num ACL Data Packets : 0x", result[3],
"\n");
   Trace(" Total Num SCO Data Packets : 0x", result[4],
"\n");
}
```

HCIReadClockOffset()

`HCIReadClockOffset(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with.		

Return values

Returns a list with two values: *status* and *offset*.

Status (element 0) is one of the following:

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Offset (element 1) is the two-byte value of the clock offset.

Comments

Reads the clock offset to remote devices.

Example

```
result = HCIReadClockOffset();
Trace("HCIReadClockOffset returned: ", result[0], "\n");
if (result[0] == "Success")
{
   Trace("Clock offset is: 0x", result[1], "\n");
}
```

HCIReadConnectionAcceptTimeout()

`HCIReadConnectionAcceptTimeout()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *timeout*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Timeout (element 1) is the two-byte value of the timeout, interpreted as multiples of 0.625-msec intervals.

Comments

Reads the current timeout interval for connection. The timeout value defines the time duration from when the Host Controller sends a Connection Request event until the Host Controller automatically rejects an incoming connection.

Example

```
result = HCIReadConnectionAcceptTimeout();
Trace("ReadConnectionAcceptTimeout returned: ", result[0],
"\n");
if (result[0] == "Success")
{
   Trace("Timeout value is: 0x", result[1], "\n");
}
```

HCIReadCountryCode()

`HCIReadCountryCode()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *country code*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Country code (element 1) is the one-byte country code value. (0=North America and Europe; 1=France.)

Comments

Reads the country code value. This value defines which range of frequency band of the ISM 2.4-GHz band is used by the device.

Example

```
result = HCIReadCountryCode();
Trace("HCIReadCountryCode returned: ", result[0], "\n");
if (result[0] == "Success")
{
```

```
   Trace("Country code is: 0x", result[1], "\n");
}
```

HCIReadCurrentIACLAP()

HCIReadCurrentIACLAP()

Parameter	Meaning	Default Value	Comments
N/A			

Return value

Returns a list with two values: *status* and *Current IAC LAP.*

Status (element 0) is one of the following:

- "Success"
- "Failure"

Current IAC LAP (element 1) is the 3-byte value of the LAPs (Lower Address Part) that make up the current IAC (Inquiry Access Code).

Comments

Reads the number and values of the currently used IAC LAPs.

Example

```
result = HCIReadCurrentIACLAP();
if(result[0] == "Success")
{
   Trace("Current number of used IAC LAPs is: ", result[1],
"\n");
   if(result[1] > 0)
   {
      Trace("Currently used IAC LAPs are:");
      for(i = 0; i < result[1]; i = i + 1)
      {
         Trace(" 0x", result[2 + i]);
      }
      Trace("\n\n");
   }
}
```

HCIReadEncryptionMode()

`HCIReadEncryptionMode()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *encryption mode*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Encryption mode (element 1) is the one-byte encryption mode value. (0=Encryption disabled; 1=Encryption enabled for point-to-point packets only; 2=Encryption enabled for both point-to-point and broadcast packets.)

Comments

Reads the encryption mode value. This value controls whether the local device requires encryption to the remote device at connection setup.

Example

```
result = HCIReadEncryptionMode();
Trace("HCIReadEncryptionMode returned: ", result[0], "\n");
if (result[0] == "Success")
{
   Trace("Encryption mode is: 0x", result[1], "\n");
}
```

HCIReadLinkPolicySettings()

`HCIReadLinkPolicySettings(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		

Return value

Returns the following list of values: *status* and *link policy settings.*

Status (element 0) is one of the following:

- "Success"

- "Failure"
- "Failed: Device not found"
- "Not connected"

Link policy settings (element 1) is the two-byte value of the link policy settings.

Comments

Reads the value of the Link_Policy_Settings parameter for the device.

Example

```
Device = '010203040506';
result = HCIReadLinkPolicySettings(Device);
Trace("HCIReadLinkPolicySettings returned: ", result[0],
"\n");
if (result[0] == "Success")
{
   Trace("Link Policy Settings : ", result[1] ,"\n");
}
```

HCIReadLinkSupervisionTimeout()

`HCIReadLinkSupervisionTimeout(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		

Return value

Returns the following list of values: *status* and *link supervision timeout.*

Status (element 0) is one of the following:

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Link supervision timeout (element 1) is the timeout, interpreted as multiples of 0.625-msec intervals.

Comments

Reads the value for the Link_Supervision_Timeout parameter for the device.

Example

```
Device = '010203040506';
result = HCIReadLinkSupervisionTimeout(Device);
Trace("HCIReadLinkSupervisionTimeout returned: ",
result[0], "\n");
if (result[0] == "Success")
{
   Trace("Link Supervision Timeout is: ", result[1] ,"\n");
}
```

HCIReadLocalName()

`HCIReadLocalName()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *name*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Name (element 1) is a string representing the device name.

Comments

Reads the "user-friendly" name of the local Bluetooth device.

Example

```
result = HCIReadLocalName();
Trace("HCIReadLocalName returned: ", result[0], "\n");
if (result[0] == "Success")
{
   Trace("Local device name is: ", result[1], "\n");
}
```

HCIReadLocalSupportedFeatures()

`HCIReadLocalSupportedFeatures()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *features*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Features (element 1) is the eight-byte bit mask list of Link Manager Protocol features.

Comments

Reads the LMP supported features for the local device.

Example

```
result = HCIReadLocalSupportedFeatures();
Trace("HCIReadLocalSupportedFeatures returned: ",
result[0], "\n");
if (result[0] == "Success")
{
   Trace("Local supported features data is: ", result[1],
"\n");
}
```

HCIReadLocalVersionInformation()

`HCIReadLocalVersionInformation()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with six values: *status, HCI version, HCI revision, LMP version, manufacturer name,* and *LMP subversion.*

Status (element 0) is one of the following:

- "Success"
- "Failure"

HCI version (element 1) is the one-byte HCI version value. (0=1.0B, 1=1.1.)

HCI revision (element 2) is the two-byte HCI revision value.

LMP version (element 3) is the one-byte Link Manager Protocol version value.

Manufacturer name (element 4) is the two-byte manufacturer name of the Bluetooth hardware.

LMP subversion (element 5) is the two-byte Link Manager Protocol subversion value.

Comments

Reads the version information for the local device.

Example

```
result = HCIReadLocalVersionInformation();
Trace("HCIReadLocalVersionInformation returned: ",
result[0], "\n");
if (result[0] == "Success")
{
   Trace("HCI version is: 0x",       result[1], "\n");
   Trace("HCI revision is: 0x",      result[2], "\n");
   Trace("LMP version is: 0x",       result[3], "\n");
   Trace("Manufacturer name is: 0x", result[4], "\n");
   Trace("LMP subversion is: 0x",    result[5], "\n");
}
```

HCIReadLoopbackMode()

`HCIReadLoopbackMode()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *loopback mode*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Loopback mode (element 1) is the one-byte loopback mode value. (0=No loopback mode; 1=Local loopback mode; 2=Remote loopback mode.)

Comments

Reads the loopback mode value. This value determines the path by which the Host Controller returns information to the Host.

Example

```
result = HCIReadLoopbackMode();
Trace("HCIReadLoopbackMode returned: ", result[0], "\n");
```

```
if (result[0] == "Success")
{
   Trace("Loopback mode is: 0x", result[1], "\n");
}
```

HCIReadNumberOfSupportedIAC()

HCIReadNumberOfSupportedIAC()

Parameter	Meaning	Default Value	Comments
N/A			

Return value

Returns a list with two values: *status* and *number of supported IAC*.

Status is one of the following:

- "Success"
- "Failure"

Number of supported IAC is a 1-byte value that specifies the number of Inquiry Access Codes that the local Bluetooth device can listen for at one time.

Comments

Reads the number of supported IACs.

Example

```
result = HCIReadNumberOfSupportedIAC ();
Trace("HCIReadNumberOfSupportedIAC returned: ", result[0],
"\n");
if (result[0] == "Success")
{
   Trace("The number of supported IAC is: ", result[1],
"\n\n");
}
```

HCIReadPageScanMode()

HCIReadPageScanMode()

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *page scan mode*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Page scan mode (element 1) is the one-byte page scan mode value. (0=Mandatory page scan mode; 1=Optional page scan mode I; 2=Optional page scan mode II; 3=Optional page scan mode III.)

Comments

Reads the page scan mode value. This value indicates the mode used for default page scan.

Example

```
result = HCIReadPageScanMode();
Trace("HCIReadPageScanMode returned: ", result[0], "\n");
if (result[0] == "Success")
{
   Trace("Page scan mode is: 0x", result[1], "\n");
}
```

HCIReadPageScanPeriodMode()

`HCIReadPageScanPeriodMode()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *page scan period mode*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Page scan period mode (element 1) is the one-byte page scan period mode value. (0=P0; 1=P1; 2=P2.)

Comments

Reads the page scan period mode value. Each time an inquiry response message is sent, the Bluetooth device will start a timer, the value of which depends on the page scan period mode.

Example

```
result = HCIReadPageScanPeriodMode();
Trace("HCIReadPageScanPeriodMode returned: ", result[0],
"\n");
if (result[0] == "Success")
{
   Trace("Page scan period mode is: 0x", result[1], "\n");
}
```

HCIReadPageTimeout()

`HCIReadPageTimeout()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *page timeout*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Page timeout (element 1) is the two-byte page timeout value, in increments of 0.625-msec intervals.

Comments

Reads the page timeout value. This value defines the maximum time the local Link Manager will wait for a baseband page response from the remote device at a locally initiated connection attempt.

Example

```
result = HCIReadPageTimeout();
Trace("HCIReadPageTimeout returned: ", result[0], "\n");
if (result[0] == "Success")
{
   Trace("Page timeout is: 0x", result[1], "\n");
}
```

HCIReadPINType()

`HCIReadPINType()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *PIN type*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

PIN type (element 1) is the one-byte PIN type. (0=Variable PIN; 1=Fixed PIN.)

Comments

Reads the PIN type, which determines whether the Host supports variable PIN codes or only a fixed PIN code.

Example

```
result = HCIReadPINType();
Trace("HCIReadPINType returned: ", result[0], "\n");
if (result[0] == "Success")
{
   Trace("PIN type is: 0x", result[1], "\n");
}
```

HCIReadRemoteSupportedFeatures()

`HCIReadRemoteSupportedFeatures(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with.		

Return values

Returns a list with two values: *status* and *features*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

- "Failed: Device not found"
- "Not connected"

Features (element 1) is the eight-byte bit mask list of Link Manager Protocol features.

Comments

Reads the LMP supported features for the specified device. An ACL connection with the device is required.

Example

```
Device = '010203040506';
result = HCIReadRemoteSupportedFeatures(Device);
Trace("HCIReadRemoteSupportedFeatures returned: ",
result[0], "\n");
if (result[0] == "Success")
{
   Trace("Remote supported features data is: 0x",
result[1], "\n");
}
```

HCIReadRemoteVersionInformation()

`HCIReadRemoteVersionInformation(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with.		

Return values

Returns a list with four values: *status, LMP version, manufacturer name,* and *LMP subversion.*

Status (element 0) is one of the following:

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

LMP version (element 1) is the one-byte Link Manager Protocol version value.

Manufacturer name (element 2) is the two-byte manufacturer name of the Bluetooth hardware.

LMP subversion (element 3) is the two-byte Link Manager Protocol subversion value.

Comments

Reads the version information for the specified device. An ACL connection with the device is required.

Example

```
Address = '010203040506';
result = HCIReadRemoteVersionInformation(Address);
Trace("HCIReadRemoteVersionInformation returned: ",
result[0], "\n");
if (result[0] == "Success")
{
   Trace("LMP version is: 0x", result[1], "\n");
   Trace("Manufacturer name is: 0x", result[2], "\n");
   Trace("LMP subversion is: 0x", result[3], "\n");
}
```

HCIReadScanEnable()

HCIReadScanEnable()

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "GENERAL_ACCESSIBLE"
- "LIMITED_ACCESSIBLE"
- "NOT_ACCESSIBLE"
- "CONNECTABLE_ONLY"
- "Failure"

Comments

Retrieves the current accessible mode of Merlin's Wand.

Example

```
Trace("Merlin's Wand accessible mode: ",
HCIReadScanEnable());
```

HCIReadSCOFlowControlEnable()

`HCIReadSCOFlowControlEnable()`

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *SCO flow control enable.*

Status (element 0) is one of the following:

- "Success"
- "Failure"

SCO flow control enable (element 1) is the one-byte SCO flow control value. (0=SCO flow control disabled; 1=SCO flow control enabled.)

Comments

Reads the SCO flow control enable value. This value determines whether the Host Controller will send Number Of Completed Packets events for SCO Connection Handles.

Example

```
result = HCIReadSCOFlowControlEnable();
Trace("HCIReadSCOFlowControlEnable returned: ", result[0],
"\n");
if (result[0] == "Success")
{
   Trace("SCO flow control enable is: 0x", result[1],
"\n");
}
```

HCIReadStoredLinkKey()

`HCIReadStoredLinkKey(Address, ReadAll)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device that will have its link key read		

Parameter	Meaning	Default Value		Comments
ReadAll	Boolean value that indicates whether to read only the specified address's link key, or all link keys	0	0 or 1	

Return values

Returns a list with two values: *status* and *data*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Data (element 1) is a list containing zero or more pairs of the following two values:

- BDADDR: the Bluetooth Address that the link key corresponds to
- LinkKey: the link key for the specified address

Comments

Attempts to read the stored link key for the specified address or for all addresses, depending on the value of ReadAll.

Example

```
result = HCIReadStoredLinkKey('6E8110AC0008', 1);
Trace("HCIReadStoredLinkKey() returned: ", result[0],
"\n\n");

if (result[0] == "Success")
{
   list = result[1];
   i = 0;
   while (list[(i*2)] != null)
   {
      Trace("**********************\n");
      Trace("BDADDR: ", list[(i*2)], "\n");
      Trace("Link Key: ", list[(i*2)+1], "\n");
      Trace("**********************\n");
      i = i + 1;
   }
}
```

HCIReadVoiceSetting()

HCIReadVoiceSetting()

Parameter	Meaning	Default Value	Comments
N/A			

Return values

Returns a list with two values: *status* and *voice setting*.

Status (element 0) is one of the following:

- "Success"
- "Failure"

Voice setting (element 1) is the 10-bit voice setting value.

Comments

Reads the voice setting value. This value controls all settings for voice connections.

Example

```
result = HCIReadVoiceSetting();
Trace("HCIReadVoiceSetting returned: ", result[0], "\n");
if (result[0] == "Success")
{
   Trace("Voice setting is: 0x", result[1], "\n");
}
```

HCIRejectConnectionRequest()

HCIRejectConnectionRequest()

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"

Comments

Sets the accept connection request variable to False.

Example

```
status = HCIRejectConnectionRequest();
Trace("HCIRejectConnectionRequest returned: ", status,
"\n\n");
```

HCIRemoveSCOConnection()

`HCIRemoveSCOConnection(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with		

Return value

- "Success"
- "Not connected"
- "Failure"

Comments

Removes an existing SCO connection associated with the specified device.

Example

```
result = HCIRemoveSCOConnection(Devices[0]);
```

HCIReset()

`HCIReset()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"
- "Invalid parameter"
- "Failed: Invalid Type"
- "Failed: HCI initialization error"

Comments

Resets the Host Controller and Link Manager.

Example

```
result = HCIReset();
```

HCIRoleDiscovery()

`HCIRoleDiscovery(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device relative to which we want to know our role		A connection must exist with this address for Role Discovery to work.

Return value

- "Master"
- "Slave"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Comments

Attempts to discover the role of our device relative to the specified device.

Example

```
result = HCIRoleDiscovery('6E8110AC0108');
if(result != "Success")
{
   MessageBox(result, "Failed to get role!");
}
else
{
   Trace("Our role is: ", result, "\n\n");
}
```

HCISetConnectionEncryption()

`HCISetConnectionEncryption(Address, SetEncryption)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device whose encryption to enable or disable		

Parameter	Meaning	Default Value	Comments
SetEncryption	Boolean value that indicates whether to enable or disable encryption	0	0 or 1 A connection must be established and authenticated before you can enable encryption successfully

Return value

- "Success"
- "Failure"
- "Timed Out"
- "Failed: Device not found"
- "Not connected"

Comments

Enables and disables the link-level encryption for the address specified

Example

```
result = HCISetConnectionEncryption('6E8110AC0108', 0);
if(result != "Success")
{
   MessageBox(result, "Failed to disable encryption!");
}
```

HCISetEventFilter()

`HCISetEventFilter(FilterType, FilterConditionType, Condition)`

Parameter	Meaning	Default Value	Comments
FilterType	Filter type: 0 = Clear all filters; 1 = Inquiry result; 2 = Connection setup; 3-255 = Reserved		If value 0 is used, no other parameters should be supplied.
Filter Condition Type	Type of filter condition.		This parameter has different meanings depending on the filter type.

Parameter	Meaning	Default Value	Comments
Condition	Details of the filter to be set.		Must be entered as a series of bytes within brackets, e.g., [0x1, 0x12, 0x0F]. Byte values must be entered in hex notation separated by commas.

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

Instructs the Host Controller to send only certain types of events to the Host.

Examples

```
# Clear All Filters
result = HCISetEventFilter(0);
Trace("Result of clearing all filters: ", result, "\n");

# Inquiry Result
result = HCISetEventFilter(1, 2,
[0xA,0x1,0x24,0x12,0xFB,0xAA]);
Trace("Result of Inquiry Result filter: ", result, "\n");

# Connection Setup
result = HCISetEventFilter(2, 0, [0x1]);
Trace("Result of Connection Setup filter: ", result, "\n");
```

HCISniffMode()

`HCISniffMode(Address, MaxInterval, MinInterval, Attempt, Timeout)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		
MaxInterval	Maximum number of 0.625-msec intervals between sniff periods.		Range is 0x0001 to 0xFFFF (0.625 msec to 40.9 sec).

Parameter	Meaning	Default Value	Comments
MinInterval	Minimum number of 0.625-msec intervals between sniff periods.		Range is 0x0001 to 0xFFFF (0.625 msec to 40.9 sec).
Attempt	Number of receive slots for sniff attempt.		Range is 0x0001 to 0x7FFF (0.625 msec to 20.5 sec).
Timeout	Number of receive slots for sniff time-out.		Range is 0x0001 to 0x7FFF (0.625 msec to 20.5 sec).

Return value

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Comments

Enters Sniff mode with parameters as specified.

Example

```
Device = '010203040506';
result = HCISniffMode(Device, 0xFFFF, 100, 0x3FF6, 0x7FFF);
Trace("HCISniffMode result is: ", result, "\n");
```

HCISwitchRole()

`HCISwitchRole(Address, Role)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		
Role			Values: "Master", "Slave"

Return value

- "Success"
- "Failure"

- "Failed: Device not found"
- "Not connected"
- "Invalid parameter"

Comments

Switches the current role of the device in the piconet.

Example

```
Device = '010203040506';
result = HCISwitchRole(Device, "Slave");
Trace("HCISwitchRole result is: ", result, "\n\n");
```

HCIWriteAuthenticationEnable()

`HCIWriteAuthenticationEnable(AuthenticationEnable)`

Parameter	Meaning	Default Value	Comments
AuthenticationEnable	Authentication value: 0 = Authentication disabled; 1 = Authentication enabled for all connections; 2-255 = Reserved	0	

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

Controls whether the local device is required to authenticate the remote device at connection setup.

Example

```
result = HCIWriteAuthenticationEnable(0);
```

HCIWriteConnectionAcceptTimeout()

`HCIWriteConnectionAcceptTimeout(Interval)`

Parameter	Meaning	Default Value	Comments
Interval	Number of 0.625-msec intervals before the connection request times out.	0x1FA0 (= 5 sec)	Range is 0x0001 to 0xB540 (0.625 msec to 29 sec).

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

Sets a timeout interval for connection. The parameter defines the time duration from when the Host Controller sends a Connection Request event until the Host Controller automatically rejects an incoming connection.

Example

```
result = HCIWriteConnectionAcceptTimeout(0x1234);
```

HCIWriteCurrentIACLAP()

`HCIWriteCurrentIACLAP(NumCurrentIACs, IACLAPs)`

Parameter	Meaning	Default Value	Comments
NumCurrent IACs	Number of current IACs.		Must be 1 or 2.
IACLAPs	List of IAC_LAPs, each with a value in the range 0x9E8B00-0x 9E8B3F.		The number of values in this list must match the NumCurrentIACs parameter.

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

Writes the number and values of the IAC LAPs to be used. One of the values has to be the General Inquiry Access Code, 0x9E8B33.

Example

```
result = HCIWriteCurrentIACLAP(2, 0x9E8B33, 0x9E8B34);
Trace("Result of HCIWriteCurrentIACLAP: ", result, "\n\n");
```

HCIWriteEncryptionMode()

`HCIWriteEncryptionMode(EncryptionMode)`

Parameter	Meaning	Default Value	Comments
Encryption Mode	Encryption mode: 0 = Encryption disabled; 1 = Encryption enabled for point-to-point packets only; 2 = Encryption enabled for both point-to-point and broadcast packets; 3-255=Reserved	0	

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

Controls whether the local device requires encryption to the remote device at connection setup.

Example

```
result = HCIWriteEncryptionMode(0);
```

HCIWriteLinkPolicySettings()

`HCIWriteLinkPolicySettings(Address, LinkPolicySettings)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		
LinkPolicy Settings			Range is 0x0000-0x8000.

Return value

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Comments

Writes the value for the Link_Policy_Settings parameter for the device.

Example

```
Device = '010203040506';
result = HCIWriteLinkPolicySettings(Device, 0xF);
Trace("HCIWriteLinkPolicySettings result is: ", result,
"\n\n");
```

HCIWriteLinkSupervisionTimeout()

`HCIWriteLinkSupervisionTimeout(Address, Timeout)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device in question		
Timeout	Number of 0.625-msec intervals before connection request times out.		Range is 0x0001 to 0xFFFF (0.625 msec to 40.9 sec).

Return value

- "Success"
- "Failure"
- "Failed: Device not found"
- "Not connected"

Comments

Writes the value for the Link_Supervision_Timeout parameter for the device.

Example

```
Device = '010203040506';
result = HCIWriteLinkSupervisionTimeout(Device, 0x7D00);
Trace("HCIWriteLinkSupervisionTimeout result is: ",
result[0], "\n\n");
```

HCIWriteLoopbackMode()

`HCIWriteLoopbackMode(LoopbackMode)`

Parameter	Meaning	Default Value	Comments
Loopback Mode	Loopback mode: 0 = No loop-back mode; 1 = Local loopback mode; 2 = Remote loopback mode; 3-255 = Reserved	0	

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

Determines the path by which the Host Controller returns information to the Host.

Example

```
result = HCIWriteLoopbackMode(2);
```

HCIWritePageTimeout()

`HCIWritePageTimeout(Interval)`

Parameter	Meaning	Default Value	Comments
Interval	Number of 0.625-msec intervals before the connection attempt times out.	0x2000 (= 5.12 sec)	Range is 0x0001 to 0xFFFF (0.625 msec to 40.9 sec).

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

Sets the maximum time the local Link Manager will wait for a baseband page response from the remote device at a locally initiated connection attempt.

Example

```
result = HCIWritePageTimeout(0x4000);
```

HCIWritePINType()

`HCIWritePINType(PINType)`

Parameter	Meaning	Default Value	Comments
PINType	PIN type: 0=Variable PIN; 1=Fixed PIN		Range is 0 to 1.

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

Determines whether the Host supports variable PIN codes or only a fixed PIN code.

Example

```
result = HCIWritePINType(0);
```

HCIWriteScanEnable()

`HCIWriteScanEnable(AccessibleMode)`

Parameter	Meaning	Default Value	Comments
Accessible Mode	Access mode to set Merlin's Wand	"GENERAL_ ACCESSI- BLE"	Mode can be one of: "GENERAL_ACCESSIBLE", "NOT_ACCESSIBLE", "CONNECTABLE_ONLY"

Return value

- "Success"
- "Timed out"
- "Failure"

Comments

Sets the accessible mode of Merlin's Wand.

Example

```
HCIWriteScanEnable("CONNECTABLE_ONLY");
```

HCIWriteStoredLinkKey()

`HCIWriteStoredLinkKey(Address, LinkKey)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device that will have its link key stored		
LinkKey	String containing the link key to be stored		Up to 32 Hex digits

Return value

- "Success"
- "Failure"

Comments

Attempts to store the link key for the specified address. If a link key already exists for the specified address, it will be overwritten.

Example

```
result = HCIWriteStoredLinkKey('6E8110AC0108', "ABC123");
Trace("HCIWriteStoredLinkKey() returned: ", result,
"\n\n");
```

HCIWriteVoiceSettings()

`HCIWriteVoiceSettings(Address, VoiceSetting)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device whose voice settings to write		
VoiceSetting	Three-digit hex value containing the voice settings		Possible values: 0x0060=CVSD coding 0x0061=u-Law coding 0x0062=A-law coding

Return value

- "Success"
- "Failure"
- "Timed Out"
- "Failed: Device not found"
- "Not connected"

Comments

Attempts to write the voice settings for the specified address. A connection must be established before voice settings can be written.

Example

```
result = HCIWriteVoiceSettings('6E8110AC0108', 0x0060);
Trace("HCIWriteVoiceSettings() returned: ", result,
"\n\n");
```

C.5 OBEX Commands

OBEXClientConnect()

`OBEXClientConnect(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with		

Return value

- "Success"
- "Failure"
- "Failed: Busy"
- "Failed: Not connected"
- "Failed: Packet too small"

Comments

Establishes an OBEX client connection with the specified device.

Example

```
result = OBEXClientConnect(Devices[0]);
if(result != "Success")
{
   MessageBox("Failed to establish OBEX connection.");
}
```

OBEXClientDeinit()

`OBEXClientDeinit()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Failure"

Comments

This command is obsolete. It is provided for backward compatibility only. (The application is initialized as an OBEX client at startup and cannot be deinitialized.)

Example

```
result = OBEXClientDeinit();
```

OBEXClientDisconnect()

`OBEXClientDisconnect()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"
- "Failed: Busy"
- "Failed: Not connected"
- "Failed: Packet too small"

Comments

Breaks the current OBEX client connection.

Example

```
result = OBEXClientDisconnect();
```

OBEXClientGet()

`OBEXClientGet(RemotePath, LocalPath)`

Parameter	Meaning	Default Value	Comments
RemotePath	Path and name of object to be retrieved from server.		Path is relative to server's OBEX directory. Example: If the OBEX directory is C:\temp, a RemotePath of "file.txt" would cause the client to retrieve "C:\temp\file.txt"
LocalPath	Path and name of object to be created on client.	RemotePath argument	If omitted, object will be stored to the local OBEX directory with the name it has on the server. If specified as a relative path (i.e., without a drive letter), the path will be considered relative to the OBEX directory. If specified as a full path (i.e., with a drive letter), the object will be stored to the exact name and path specified.

Return value

- "Success"

- "Failure"
- "Failed: Busy"
- "Failed: Not connected"
- "Failed: Packet too small"
- "Failed: Invalid handle"

Comments

Retrieves object from a server and saves it to the client.

If directory names are included in either path argument, **be sure to use double-slashes to separate components** (e.g., "temp1\\temp2\\file-name.txt"). Using single slashes will cause errors.

Note that the second argument may be omitted, in which case the object will be stored to the client's OBEX directory with the same name it has on the server.

Examples

In these examples, the local OBEX directory is assumed to be c:\obexdir.

```
#store file to "file.txt" in local OBEX directory
#  (i.e., c:\obexdir\file.txt)
OBEXClientGet("file.txt");

#store file to "newfile.txt" in temp dir under OBEX dir
#  (i.e., c:\obexdir\temp\newfile.txt)
OBEXClientGet("file.txt", "temp\\newfile.txt");

#store file to "file.txt" in C:\temp
OBEXClientGet("file.txt", "C:\\temp\\file.txt");

#get file from a directory below the server's OBEX dir,
#  and save it with the same name to the same directory
#  below the local OBEX dir (i.e.,
"c:\obexdir\temp\file.txt")
OBEXClientGet("temp\\file.txt");
```

OBEXClientInit()

`OBEXClientInit()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"

Comments

This command is obsolete. It is provided for backward compatibility only. (The application is initialized as an OBEX client at startup and cannot be deinitialized.)

Example

```
result = OBEXClientInit();
```

OBEXClientPut()

`OBEXClientPut(LocalPath, RemotePath)`

Parameter	Meaning	Default Value	Comments
LocalPath	Full (*not* relative) path and name of file to be sent from client.		
RemotePath	Path and name of object to be stored on server.	Name-only portion of LocalPath argument	Path is relative to server's OBEX directory. Example: If the server's OBEX directory is C:\Temp, a RemotePath of "file.txt" would cause the server to save the file to "C:\Temp\file.txt". Note that you cannot save a file to an absolute path on the server.

Return value

- "Success"
- "Failure"
- "Failed: Busy"
- "Failed: Invalid handle"
- "Failed: Invalid parameter"
- "Failed: Media busy"
- "Failed: Not connected"
- "Failed: Packet too small"

Comments

Sends a file to the OBEX directory of the server.

If directory names are included in either path argument, **be sure to use double-slashes to separate components** (e.g., "temp1\\temp2\\filename.txt"). Using single slashes will cause errors.

Note that the second argument may be omitted, in which case the object will be stored to the server's OBEX directory with the same name it has on the client.

Examples

In these examples, the server's OBEX directory is assumed to be c:\obexdir.

```
#store file to "file.txt" in server's OBEX directory
#  (i.e., c:\obexdir\file.txt)
OBEXClientPut("c:\\temp\\file.txt");

#store file to "newfile.txt" in server's OBEX dir
#  (i.e., c:\obexdir\newfile.txt)
OBEXClientPut("c:\\temp\\file.txt", "newfile.txt");

#store file to "newfile.txt" in temp dir under OBEX dir
#  (i.e., c:\obexdir\temp\newfile.txt)
OBEXClientPut("c:\\temp\\file.txt", "temp\\newfile.txt");
```

OBEXClientSetPath()

`OBEXClientSetPath(Path, Flags)`

Parameter	Meaning	Default Value	Comments
Path	New path to set		Path is relative to server's current working directory. Cannot begin "C:" or "\\\\".
Flags	SetPath flags: 0=No flags 1=Back up one level 2=Don't create specified folder if it doesn't exist 3=Back up one level and don't create specified folder		When backup is set (flag = 1 or 3), the working directory is backed up one level before the specified directory is appended (e.g., if the server's current working directory is C:\Temp, a SetPath of "Temp2" with a flag of 1 would change the directory to C:\Temp2). To set path to the OBEX root directory, use an empty path and a flag of 0 or 2.

Return value

- "Success"
- "Failure"
- "Failed: Busy"
- "Failed: Not connected"
- "Failed: Packet too small"
- "Failed: Invalid parameter"

Comments

Temporarily changes a server's current working directory, accessed by clients during `ClientGet` and `ClientPut` operations. The device must be connected to an OBEX server before the command can be successfully executed. The change is lost when the connection is broken. Note that the server's OBEX root directory cannot be changed with this command.

If the path includes multiple levels, **be sure to use double-slashes to separate components** (e.g., "temp1\\temp2"). Using single slashes will cause errors.

Example

```
#set path to <root>
status = OBEXClientSetPath("", 0);
Sleep(1000);

#set path to <root>\temp2
status = OBEXClientSetPath("temp2", 0);
Sleep(1000);

#set path to <root>\temp2\temp3
status = OBEXClientSetPath("temp3", 0);
Sleep(1000);

#set path to <root>\temp2
status = OBEXClientSetPath("", 1);
Sleep(1000);

#keep path at <root>\temp2 (assuming <root>\temp2\temp4
doesn't exist)
status = OBEXClientSetPath("temp4", 2);
Sleep(1000);

#set path to <root>\temp3\temp4
status = OBEXClientSetPath("temp3\\temp4", 1);
```

OBEXServerDeinit()

`OBEXServerDeinit()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

- "Failed: Busy"

Comments

Deinitializes an OBEX server.

Example

```
result = OBEXServerDeinit();
```

OBEXServerSetPath(Path)

`OBEXServerSetPath(Path)`

Parameter	Meaning	Default Value	Comments
Path	Path to be used as the OBEX root directory on the server		Path must be fully specified (e.g., "C:\\temp" rather than "temp")

Return value

- "Success"
- "Failure"
- "Failed: Device must be initialized as a server"

Comments

Sets the OBEX root directory on a server. This path is accessed by clients during remote `ClientGet` and `ClientPut` operations. The device must be initialized as a server before the command can be successfully executed.

In the path, **be sure to use double-slashes to separate components** (e.g., "C:\\temp\\temp2"). Using single slashes will cause errors.

Example

```
status = OBEXServerInit();
if ( status == "Success" )
{
   status = OBEXServerSetPath("c:\\temp");
}
Trace("OBEXServerSetPath returned: ", status, "\n\n");
```

C.6 RFCOMM Commands

RFCloseClientChannel()

`RFCloseClientChannel(Address, DLCI)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()

Return value

- "Success"
- "Not connected"
- "Failure"
- "Timed out"

Comments

Closes an RFCOMM channel

Example

```
RFCloseClientChannel(Devices[0], DLCI);
```

RFOpenClientChannel()

`RFOpenClientChannel(Address, ServerID)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		
ServerID	Service ID for RFCOMM channel		

Return value

The return value from RFOpenClientChannel is a list containing up to two elements. The first element is the status of the command and is one of the following strings:

- "Success"
- "Failure"
- "Timed out"
- "Not connected"
- "Restricted"

If the return value is "Success", the second element in the list is the DLCI of the established connection.

Comments

An ACL connection must already be established with the device.

Example

```
result = RFOpenClientChannel(Devices[0], 1);
if(result[0] == "Success")
{
   Trace("Successfully connected with DLCI ", result[1],
"\n");
   # Send some data over RFCOMM
}
```

RFRegisterServerChannel()

`RFRegisterServerChannel()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- Server channel ID
- "Failure"

Comments

Example

```
channel = RFRegisterServerChannel();
if(channel != "Failure")
{
   Trace("Channel ID is ", channel);
}
```

RFSendData()

`RFSendData(Address, DLCI, Data)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to send data to a master RFCOMM connection. Note that this will work only if exactly one device is connected via RFCOMM.
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()
Data	Data to send		Data can be a string, 32-bit integer value or a list containing either or both types

Return value

- "Success"
- "Timed out"
- "Not supported" (invalid data type)
- "Not connected"

Comments

An RFCOMM connection must already be established with the device.

Example

```
RFSendData(Devices[0], DLCI, "ATDT 555-1212");
RFSendData("CONNECTED_DEVICE", dlci, "AT+CKPD=200\r\n");
```

RFSendDataFromPipe()

`RFSendDataFromPipe(Address, DLCI, PipeName)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to send data to a master RFCOMM connection
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()
PipeName	Name of the transmit data pipe to get data to send		This pipe must exist.

Return value

- "Success"
- "Timed out"
- "Not supported" (invalid data type)
- "Not connected"
- "Pipe not found"
- "Internal Error"

Comments

An RFCOMM connection must already be established with the device. The pipe specified must already be set up in the Data Transfer Manager. The pipe should not be open when RFSendDataFromPipe is called.

Example

```
RFSendDataFromPipe(Devices[0], dlci, "MyPipe");
RFSendDataFromPipe("CONNECTED_DEVICE", dlci, "Pipe2");
```

RFReceiveData()

`RFReceiveData(Address, DLCI, Timeout)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to receive data from a master RFCOMM connection
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()
Timeout	Time in ms to wait for an RFCOMM connection	0 (Infinite wait)	Use 0 as the timeout value to wait infinitely

Return value

Returns a list with three values: *status*, *number of bytes*, and *data array.*

Status (element 0) is one of the following:

- "Success"
- "Not connected"
- "Timed out"

Number of bytes (element 1) is the number of bytes received.

Data array (element 2) is the sequence of data bytes received.

Comments

Receives data from a device connected via RFCOMM. Waits Timeout milliseconds (or infinitely if 0 is specified) for the device to begin sending data to Merlin's Wand.

Example

```
#Get the data; stop when no data is received for 5 secs
result = RFReceiveData(Device, DLCI, 5000);
while(result[0] == "Success")
{
   Trace("Number of data bytes received: ", result[1],
"\n");
   result = RFReceiveData(Device, DLCI, 5000);
}
```

RFWaitForConnection()

`RFWaitForConnection(ServerID, Timeout)`

Parameter	Meaning	Default Value	Comments
ServerID	Service ID for RFCOMM channel		
Timeout	Time in ms to wait for an RFCOMM connection	0 (Infinite wait)	Use 0 as the timeout value to wait infinitely.

Return value

Returns a list with three values: *status*, *DLCI*, and *BluetoothDevice*.

Status (element 0) is one of the following:

- "Success"
- "Timed out"
- "Failure"

DLCI (element 1) is the data link connection identifier.

BluetoothDevice (element 2) is the address of the connecting device.

Comments

Waits Timeout milliseconds for a device to establish an RFCOMM connection with Merlin's Wand. This function will block the specified amount of time (or infinitely if 0 is specified) unless a connection is established. If an

RFCOMM connection is already present when this function is called, it will immediately return "Success".

Example

```
# Wait 3 seconds for RFCOMM connection
Trace("RFWaitForConnection\n");
result = RFWaitForConnection(1, 3000);
if( result[0] == "Success" )
{
   Trace("Incoming RFCOMM connection DLCI: ", result[1],
"\n");
   Trace("Connecting device address: ", result[2], "\n");
}
```

RFAcceptChannel()

`RFAcceptChannel(bAccept)`

Parameter	Meaning	Default Value	Comments
bAccept	Boolean value indicating whether to accept the channel or not	0	0 or 1

Return value

- "Success"

Comments

Example

```
status = RFAcceptChannel(1);
Trace("RFAcceptChannel returned: ", status, "\n\n");
```

RFAcceptPortSettings()

`RFAcceptPortSettings(bAccept)`

Parameter	Meaning	Default Value	Comments
bAccept	Boolean value indicating whether to accept the channel or not	0	0 or 1

Return value

- "Success"

Comments

Example

```
status = RFAcceptPortSettings(0);
Trace("RFAcceptPortSettings returned: ", status, "\n\n");
```

RFCreditFlowEnabled()

`RFCreditFlowEnabled(Address, DLCI)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to check if credit flow is enabled on a master RFCOMM connection. Note that this will work only if exactly one device is connected via RFCOMM.
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()

Return value

- "Enabled"
- "Disabled"
- "Not Connected"

Comments

Checks to see if credit flow is enabled on a particular RFCOMM connection.

Example

```
result = RFOpenClientChannel(Device, 1);
DLCI = result[1];
if(result[0] == "Success")
{
   status = RFCreditFlowEnabled("CONNECTED_DEVICE", DLCI);
   Trace("RFCreditFlowEnabled returned: ", status, "\n\n");
}
```

RFRequestPortSettings()

```
RFRequestPortSettings(Address, DLCI, BaudRate,
DataFormat, FlowControl, Xon, Xoff)
```

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to request port settings on a master RFCOMM connection. Note that this will work only if exactly one device is connected via RFCOMM.
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()
BaudRate	String containing the baud rate		Can be "2400", "4800", "7200", "9600", "19200", "38400", "57600", "115200", "230400"
DataFormat	List of strings containing data bits, stop bits, and parity settings		Can be "RF_DATA_BITS_5", "RF_DATA_BITS_6", "RF_DATA_BITS_7", "RF_DATA_BITS_8", "RF_STOP_BITS_1", "RF_STOP_BITS_1_5", "RF_PARITY_NONE", "RF_PARITY_ON", "RF_PARITY_TYPE_ODD", "RF_PARITY_TYPE_EVEN", "RF_PARITY_TYPE_MARK", "RF_PARITY_TYPE_SPACE", "RF_DATA_BITS_MASK", "RF_STOP_BITS_MASK", "RF_PARITY_MASK", "RF_PARITY_TYPE_MASK"
FlowControl	List of strings indicating port flow control options		Can be "RF_FLOW_CTRL_NONE", "RF_XON_ON_INPUT", "RF_XON_ON_OUTPUT", "RF_RTR_ON_INPUT", "RF_RTR_ON_OUTPUT", "RF_RTC_ON_INPUT", "RF_RTC_ON_OUTPUT", "RF_FLOW_RTS_CTS", "RF_FLOW_DTR_DSR", "RF_FLOW_XON_XOFF"
Xon	Number indicating the XON character		
Xoff	Number indicating the XOFF character		

Return value

- "Success"
- "Failure"
- "Not connected"
- "Timed Out"

Comments

Submits a request to change the port settings on a particular RFCOMM connection.

Example

```
result = RFOpenClientChannel(Device, 1);
DLCI = result[1];
if(result[0] == "Success")
{
   status = RFRequestPortSettings("CONNECTED_DEVICE", DLCI,
"57600", ["RF_DATA_BITS_8"], ["RF_FLOW_CTRL_NONE"], 11,
13);
   Trace("RFRequestPortSettings returned: ", status,
"\n\n");
}
```

RFRequestPortStatus()

`RFRequestPortStatus(Address, DLCI)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to request the port status on a master RFCOMM connection. Note that this will work only if exactly one device is connected via RFCOMM.
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()

Return value

Returns a list with two values: *status* and *portSettings*.

Status (element 0) is one of the following:

- "Success"
- "Failure"
- "Not Connected"
- "Timed Out"

portSettings (element 1) is a list containing the following five values:

- BaudRate (element 0) is a string containing the baud rate
- DataFormat (element 1) is a string containing data bits, stop bits, and parity settings
- FlowControl (element 2) is a string indicating port flow control options
- Xon (element 3) is a string containing the XON character
- Xoff (element 4) is a string containing the XOFF character

Comments

Requests the port settings on a particular RFCOMM connection.

Example

```
result = RFOpenClientChannel(Device, 1);
DLCI = result[1];
if(result[0] == "Success")
{
   res = RFRequestPortStatus(Device, DLCI);
   Trace("RFRequestPortStatus returned: ", res[0], "\n\n");
   if (res[0] == "Success")
   {
      settingsList = res[1];
      Trace("BaudRate:  ", settingsList[0], "\n");
      Trace("DataFormat:  ", settingsList[1], "\n");
      Trace("Xon:  ", settingsList[3], "\n");
      Trace("Xoff:  ", settingsList[4], "\n");
   }
}
```

RFSetLineStatus()

`RFSetLineStatus(Address, DLCI, LineStatus)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to set line status on a master RFCOMM connection. Note that this will work only if exactly one device is connected via RFCOMM.
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()
LineStatus	List of strings representing the Line Status		Can be "RF_LINE_ERROR", "RF_OVERRUN", "RF_PARITY", "RF_FRAMING"

Return value

- "Success"

- "Failure"
- "Not Connected"
- "Timed Out"

Comments

Sets the line status on a particular RFCOMM connection.

Example

```
result = RFOpenClientChannel(Device, 1);
DLCI = result[1];
if(result[0] == "Success")
{
   status = RFSetLineStatus("CONNECTED_DEVICE", DLCI,
["RF_LINE_ERROR", "RF_FRAMING"]);
   Trace("RFSetLineStatus returned: ", status, "\n\n");
}
```

RFSetModemStatus()

`RFSetModemStatus(Address, DLCI, ModemSignals, BreakLength)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to set modem status on a master RFCOMM connection. Note that this will work only if exactly one device is connected via RFCOMM.
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()
ModemSignals	List of strings specifying signal types		Can be "RF_FLOW", "RF_RTC", "RF_RTR", "RF_IC", "RF_DV", "RF_DSR", "RF_CTS", "RF_RI", "RF_CD", "RF_DTR", "RF_RTS"
BreakLength	Indicates the length of the break signal in 200 ms units		Must be between 0 and 15 (inclusive). If 0, no break signal was sent.

Return value

- "Success"
- "Failure"
- "Not Connected"
- "Timed Out"

Comments

Sets the modem status on a particular RFCOMM connection.

Example

```
result = RFOpenClientChannel(Device, 1);
DLCI = result[1];
if(result[0] == "Success")
{
   status = RFSetModemStatus("CONNECTED_DEVICE", DLCI,
["RF_FLOW"], 3);
   Trace("RFSetModemStatus returned: ", status, "\n\n");
}
```

RFSendTest()

`RFSendTest(Address, DLCI)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to send a test frame on a master RFCOMM connection. Note that this will work only if exactly one device is connected via RFCOMM.
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()

Return value

- "Success"
- "Failure"
- "Not Connected"
- "Failure"

Comments

Sends a test frame on a particular RFCOMM connection.

Example

```
result = RFOpenClientChannel(Device, 1);
DLCI = result[1];
if(result[0] == "Success")
{
   status = RFSendTest("CONNECTED_DEVICE", DLCI);
   Trace("RFSendTest returned: ", status, "\n\n");
}
```

RFAdvanceCredit()

`RFAdvanceCredit(Address, DLCI, credit)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device		Can use "CONNECTED_DEVICE" to advance a credit to a master RFCOMM connection. Note that this will work only if exactly one device is connected via RFCOMM.
DLCI	Data link connection identifier		The DLCI is returned by RFOpenClientChannel()
credit	Number of credits to advance		

Return value

- "Success"
- "Failure"
- "Not Connected"

Comments

Advances a specified number of credits to a particular RFCOMM connection.

Example

```
result = RFOpenClientChannel(Device, 1);
DLCI = result[1];
if(result[0] == "Success")
{
   status = RFAdvanceCredit(Device, DLCI, 2);
   Trace("RFAdvanceCredit returned: ", status, "\n\n");
}
```

C.7 TCS Commands

TCSRegisterProfile()

`TCSRegisterProfile()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

Register Intercom profile with the application.

Example

```
result = TCSRegisterProfile();
Trace("TCSRegisterProfile returned: ", result, "\n");
```

TCSOpenChannel()

`TCSOpenChannel(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with		

Return value

- "Success"
- "Failure"
- "Not Found"
- "Timed Out"

Comments

This command opens an L2CAP channel with TCS PSM and initializes a TCS state machine into NULL state

Example

```
result = TCSOpenChannel('010203040506');
Trace("TCSOpenChannel result : ", result, "\n");
if( result != "Success")
 return result;
```

TCSStartCall()

`TCSStartCall()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

This command must be called right after `TCSOpenChannel`. It automatically sends a sequence of TCS messages according to the Intercom profile specification of the TCS state machine. After successful execution of this command, TCS state machine is in ACTIVE state and SCO connection is opened.

Example

```
result = TCSStartCall();
Trace("TCSStartCall result : ", result, "\n");
if( result != "Success")
   return result;
```

TCSDisconnectCall()

`TCSDisconnectCall()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

This command is called to close an existing TCS connection according to the Intercom profile specification of the TCS state machine, close the L2CAP connection, and close the SCO connection.

Example

```
result = TCSDisconnectCall();
   Trace("TCSDisconnectCall result : ", result, "\n");
   if( result != "Success")
      return result;
```

TCSSendInfoMessage()

TCSSendInfoMessage(Phone_Number)

Parameter	Meaning	Default Value	Comments
Phone_Number	Up to 10-digit Phone Number		

Return value

- "Success"
- "Failure"
- "Invalid Parameter"

Comments

This command can be called after a TCS channel is opened. It sends an INFORMATION TCS message with a called party number.

Example

```
result = TCSSendInfoMessage("4088447081");
   Trace("TCSSendInfoMessage result : ", result, "\n");
   if( result != "Success")
      return result;

#############################################################
#  Tested TCS Call initiation, information sending,     #
#  and Call clearing                                    #
#############################################################

Main()
{
   #Device = '838010AC0008';

   Device = DoInquiry();
   Trace(Device, "\n");

   result = Connect(Device[0]);
   Trace("Connection result : ", result, "\n");
   if( result != "Success")
      return result;

   Sleep(1000);

   result = TCSRegisterProfile();
   Trace("TCSRegisterProfile result : ", result, "\n");
   if( result != "Success")
      return result;
```

```
   Sleep(1000);

   result = TCSOpenChannel(Device[0]);
   Trace("TCSOpenChannel result : ", result, "\n");
   if( result != "Success")
      return result;

   Sleep(1000);

   result = TCSStartCall();
   Trace("TCSStartCall result : ", result, "\n");
   if( result != "Success")
      return result;

   Sleep(1000);

   result = TCSSendInfoMessage("4088447081");
   Trace("TCSSendInfoMessage result : ", result, "\n");
   if( result != "Success")
      return result;

   Sleep(1000);

   result = TCSDisconnectCall();
   Trace("TCSDisconnectCall result : ", result, "\n");
   if( result != "Success")
      return result;

   Sleep(1000);

   Trace("HCI Disconnect result: ",
   Disconnect(Device[0]), "\n");
}
```

C.8 L2CAP Commands

L2CAPConfigurationSetup()

`L2CAPConfigurationSetup(FlushTimeout, ServiceType, TokenRate, TokenBucketSize, PeakBandwidth, Latency, DelayVariation)`

Parameter	Meaning	Default Value	Comments
FlushTimeout	Amount of time that the sender will attempt transmission before flushing the packet	0xFFFF	Time is in milliseconds.
ServiceType	The required level of service	0x01	Possible values: 0x00 (no traffic), 0x01 (best effort),0x02 (guaranteed), Other (reserved)
TokenRate	The rate at which traffic credits are granted	0x00000000	0x00000000: no token rate is specified. 0xFFFFFFFF: a wild card value that matches the maximum token rate. Rate is in bytes per second.
TokenBucket Size	The size of the token bucket	0x00000000	0x00000000: no token bucket is needed. 0xFFFFFFFF: a wild card value that matches the maximum token bucket. Size is in bytes.
PeakBandwidth	A value that limits the speed at which packets may be sent consecutively	0x00000000	The default value indicates that the maximum bandwidth is unknown. The speed is in bytes per second.
Latency	The maximum delay that is acceptable between transmission of a bit and its initial transmission over the air	0xFFFFFFFF	The default value represents a Do Not Care. The delay is in milliseconds.

Parameter	Meaning	Default Value	Comments
DelayVariation	This value represents the difference between the maximum and minimum delay possible that a packet will experience	0xFFFFFFFF	The default value represents a Do Not Care. The difference is in microseconds.

Return value

- "Success"
- "Failure"

Comments

This command is used to request specified configuration for L2CAP channel. It should be executed before L2CAPConnectRequest().

For a detailed description of parameters, see the L2CAP section of the Bluetooth Specification.

Example

```
L2CAPConfigurationSetup(0xFFFF, 1, 0, 0, 0, 0xFFFFFFFF,
0xFFFFFFFF);
```

L2CAPConfigurationResponse()

`L2CAPConfigurationResponse(Reason)`

Parameter	Meaning	Default Value	Comments
Reason	Configuration response	"Accept"	Possible values: "Accept" "Reject-unknown options" "Reject-unacceptable params" "Reject"

Return value

- "Success"
- "Failure"

Comments

This command is used to automatically send the response to an incoming configuration request. It should be executed before an incoming configuration request.

Example

```
L2CAPConfigurationResponse("Reject-unknown options");
```

L2CAPConnectRequest()

`L2CAPConnectRequest(Address, PSM, ReceiveMTU)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of the remote device		
PSM			
ReceiveMTU		0x01C2	

Return value

Returns a list with three values: *result*, *ACL Handle*, and a *list of all L2CAP CIDs*.

Result (element 0) is one of the following:

- "Success"
- "Failure"
- "Not found"
- "Not connected"

ACL Handle (element 1) is a unique identifier for an ACL connection.

List of all L2CAP CIDs (element 2) is a list of all channel identifiers for an L2CAP connection with a particular device.

Comments

This command is used to establish an L2CAP channel to the specified remote device.

Example

```
result = L2CAPConnectRequest('0080370DBD02', 0x1001,
0x1C2);
Trace("L2CAPConnectRequest returned: ", result[0], "\n");
if (result[0] == "Success")
{
   Handle = result[1];
   CID = result[2];
}
```

L2CAPConnectResponse()

`L2CAPConnectResponse(Response)`

Parameter	Meaning	Default Value	Comments
Response		"Accept"	Possible values: "Accept" "Reject_Pending" "Reject_PSM_Not_Supported" "Reject_Security_Block" "Reject_No_Resources"

Return value

- "Success"
- "Failure"

Comments

This command is used to send an automatic response to an incoming L2CAP connection request. Execute this command before an incoming connection request.

Example

```
L2CAPConnectResponse("Reject_No_Resources");
```

L2CAPDeregisterAllPsm()

`L2CAPDeregisterAllPsm()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

This command is used to deregister all registered PSMs identifiers with L2CAP

Example

```
result = L2CAPDeregisterAllPsm();
Trace("DeregisterAllPsm : ", result, "\n");
```

L2CAPDisconnectRequest()

`L2CAPDisconnectRequest(CID)`

Parameter	Meaning	Default Value	Comments
CID	L2CAP channel identifier		

Return value

- "Success"
- "Failure"
- "Not connected"

Comments

This command is used to disconnect specified L2CAP channel

Example

```
L2CAPDisconnectRequest(0x0040);
```

L2CAPEchoRequest()

`L2CAPEchoRequest(Address, Data)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of the remote device		
Data			

Return value

Returns a list with two values: *status* and *data*.

Status (element 0) is one of the following:

- "Success"
- "Failure"
- "Not found"
- "Not connected"

Data (element 1) is the data returned by the remote device.

Comments

This command sends an Echo Request to the L2CAP protocol on the specified remote device. The data length should not exceed the default L2CAP signaling MTU (44 bytes).

Example

```
result = L2CAPEchoRequest('838010AC0008', "Test");
Trace("L2CAPEchoRequest result : ", result[0], "\n");

if(result[0] == "Success")
{
   Trace("Data : ", result[1], "\n");
}
```

L2CAPInfoRequest()

`L2CAPInfoRequest(Address)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of the remote device		

Return values

Returns a list with three values: *status, number of bytes,* and *data*.

Status (element 0) is one of the following:

- "Success"
- "Failure"
- "Not found"
- "Not connected"

Number of bytes (element 1) is the number of bytes of data that follow.

Data (element 2) is the raw data.

Comments

Sends an Info Request to the L2CAP protocol on the specified remote device. Info requests are used to exchange implementation-specific information regarding L2CAP's capabilities.

Example

```
result = L2CAPInfoRequest('838010AC0008');
Trace("L2CAPInfoRequest result : ", result[0], "\n");
if(result[0] == "Success")
{
   Trace("Data length : ", result[1], "\n");
   Trace("Data        : ", result[2], "\n");
}
```

L2CAPRegisterPsm()

`L2CAPRegisterPsm(PSM, ReceiveMTU)`

Parameter	Meaning	Default Value	Comments
PSM			
ReceiveMTU		0x1C2	Incoming MTU size for L2CAP connection with that PSM

Return value

- "Success"
- "Failure"
- "In use"

Comments

This command is used to register a PSM identifier with L2CAP.

Example

```
Trace("Register PSM\n");
result = L2CAPRegisterPsm(0x1001, 0x1C2);
Trace(" Result : ", result, "\n");
```

L2CAPSendData()

`L2CAPSendData(ChannelID, Data)`

Parameter	Meaning	Default Value	Comments
ChannelID	L2CAP ChannelID to send data to		
Data	Data to send		Data can be a string, 32-bit integer value or a list containing either or both types

Return value

- "Success"
- "Timed out"
- "Not supported" (invalid data type)
- "Not connected"

Comments

An L2CAP connection must already be established with the device.

Example

```
result = L2CAPSendData(0x40, "test data");
Trace("Result : ", result, "\n");
```

L2CAPSendDataFromPipe()

`L2CAPSendDataFromPipe(ChannelID, PipeName)`

Parameter	Meaning	Default Value	Comments
ChannelID	L2CAP ChannelID to send data to		
PipeName	Name of the transmit data pipe to get data to send		This pipe must exist.

Return value

- "Success"
- "Timed out"
- "Not supported" (invalid data type)
- "Not connected"
- "Pipe not found"

Comments

An L2CAP connection must already be established with the device. The pipe specified must already be set up in the Data Transfer Manager. The pipe should not be open when L2CAPSendDataFromPipe is called.

Example

```
L2CAPSendDataFromPipe(0x0040, "Pipe1");
```

L2CAPWaitForConnection()

`L2CAPWaitForConnection(Timeout)`

Parameter	Meaning	Default Value	Comments
Timeout	Time in ms to wait for an incoming L2CAP connection	0 (Infinite wait)	Use 0 as the timeout value to wait infinitely.

Return value

- "Success"
- "Timed out"

Comments

Waits Timeout milliseconds for a device to establish an L2CAP connection with Merlin's Wand. This function will block the specified amount of time (or infinitely if 0 is specified) unless a connection is established.

Example

```
Trace("L2CAPWaitForConnection\n");
result = L2CAPWaitForConnection();
if( result == "Success")
{
   #Do something else
}
```

C.9 SDP Commands

SDPAddProfileServiceRecord()

`SDPAddProfileServiceRecord(ServerChannel, Profile)`

Parameter	Meaning	Default Value	Comments
Server Channel	RFCOMM server channel to accept incoming connections to this profile		Use the result from RFRegisterServerChannel() here
Profile	Name of SDP profile		Profile can be one of: "Headset", "HeadsetAudioGateway", "SerialPort", "DialUp", "FileTransfer", "Fax", "LAN", "ObjectPush", "Intercom", "Cordless", "Sync", "SyncCommand"

Return value

- "Success"
- "Failure"

Comments

Adds a profile to Merlin's Wand SDP database

Example

```
SDPAddProfileServiceRecord(rfChannel, "ObjectPush");
```

SDPQueryProfile()

`SDPQueryProfile(Address, Profile)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to query		
Profile	Name of SDP profile		Profile can be one of: "Headset", "HeadsetAudioGateway", "SerialPort", "DialUp", "FileTransfer", "ObjectPush", "Intercom", "Cordless", "Fax", "LAN", "Sync" or "SyncCommand"

Return value

- RFCOMM channel of the requested profile (profile is supported)
- "Failure"

Comments

Queries the specified device to see if a profile is supported.

An ACL connection must already be established with the device.

Example

```
if((RFCommId = SDPQueryProfile(Devices[0], "SerialPort"))
!= "Failure")
{
   RFOpenClientChannel(Devices[0], RFCommId);
}
```

SDPResetDatabase()

`SDPResetDatabase()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

Clears all records out of the Merlin's Wand SDP profile database.

Example

```
SDPResetDatabase();
```

SDPAddServiceRecord()

`SDPAddServiceRecord(FileName, RecordName, ServerChannel)`

Parameter	Meaning	Default Value	Comments
FileName	String containing the full path of the file that contains the record		
RecordName	String containing the name of the service record to be added		
ServerChannel	RFCOMM server channel	0	If you don't want to change the RFCOMM Server ID, set this value to 0 (or leave it blank)

Return value

- "Success"
- "Failure"
- "Failure: Could not load file"
- "Failure: Record not found"
- "Failure: Could not set RFCOMM server channel X"

Comments

If a server channel is specified, tries to set the RFCOMM server channel. If it succeeds, then it parses the file specified by FileName and tries to add the record specified by RecordName.

Example

```
status = SDPAddServiceRecord("C:\Records.sdp", "FTP Test
Record", 1);
Trace("SDPAddServiceRecord returned: ", status, "\n\n");
```

C.10 Merlin Commands

MerlinResetAllEncryptionOptions()

`MerlinResetAllEncryptionOptions()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

This command is used to remove all previously associated link keys and PIN numbers.

Example

```
MerlinResetAllEncryptionOptions();
```

MerlinSetDisplayOptions()

`MerlinSetDisplayOptions(DispOptionsFile)`

Parameter	Meaning	Default Value	Comments
DispOptionsFile	Name and full path to the display options file		

Return value

- "Success"

Comments

This command is used to set the display options file.

Example

```
MerlinSetDisplayOptions("C:\\Program
Files\\CATC\\Merlin\\default.opt");
```

MerlinSetEncryptionLinkKey()

`MerlinSetEncryptionLinkKey(Address, LinkKey)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth Address		
LinkKey	Corresponding PIN for the address		LinkKey length must be no longer than 16 bytes

Return value

- "Success"
- "Failure"
- "Invalid parameter"

Comments

This command is used to associate the LinkKey with the device address used to decrypt and display encrypted traffic.

Example

```
MerlinSetEncryptionLinkKey('008037BB2100',
"003344121222ACBBEE6172000FF0");
```

MerlinSetEncryptionPIN()

`MerlinSetEncryptionPIN(Address, PIN)`

Parameter	Meaning	Default Value	Comments
Address	Bluetooth Address		
PIN	Corresponding PIN for the address		

Return value

- "Success"
- "Failure"

Comments

This command is used to associate the PIN number with the device address used to decrypt and display encrypted traffic.

Example

```
MerlinSetEncryptionPIN('008037BB2100', "1234");
```

MerlinSetRecordingOptions()

`MerlinSetRecordingOptions(RecOptionsFile)`

Parameter	Meaning	Default Value	Comments
RecOptionsFile	Name and full path to the recording options file		

Return value

- "Success"

Comments

This command is used to set the recording options file to be used for the next recording.

Example

```
MerlinSetRecordingOptions("C:\\CATC\\Merlin
\\default.rec");
```

MerlinStart()

`MerlinStart(RemoteMachine)`

Parameter	Meaning	Default Value	Comments
Remote Machine	Specify a remote machine to start Merlin		RemoteMachine can be a name or an IP address

Return value

- "Success"
- "Failure"
- "Already connected"

Comments

This command is used to start Merlin on a specified remote machine or on a local machine by default.

Example

```
result = MerlinStart("192.168.2.1");
Trace("Result : ", result, "\n");
```

MerlinStartRecording()

`MerlinStartRecording()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"
- "Failure"

Comments

This command is used to start a Merlin recording.

Example

```
MerlinStartRecording();
```

MerlinStop()

`MerlinStop()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"

Comments

This command is used to close the Merlin application.

Example

```
MerlinStop();
```

MerlinStopRecording()

`MerlinStopRecording()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

- "Success"

Comments

This command is used to stop a Merlin recording.

Example

```
MerlinStopRecording();
```

Appendix D: CATC Scripting Language

CATC Scripting Language (CSL) was developed to allow users to automate test processes and provide textual output to suit specific needs. CSL is used in Merlin's Wand to write traffic-generating scripts, making it possible to automate some Bluetooth command sequences. Scripts are written, saved, and run using the Script Manager utility. Scripts' output can be viewed in the Script Log.

CSL is based on C language syntax, so anyone with a C programming background will have no trouble learning CSL. The simple, yet powerful, structure of CSL also enables less experienced users to easily acquire the basic knowledge needed to start writing custom scripts.

Features of CATC Scripting Language

- Powerful -- provides a high-level API to the Bluetooth stack while simultaneously allowing implementation of complex algorithms.
- Easy to learn and use -- has a simple but effective syntax.
- Self-contained -- needs no external tools to run scripts.
- Wide range of value types -- provides efficient and easy processing of data.
- Integrated with over 100 commands -- includes commands for HCI, L2CAP, SDP, RFCOMM, TCS, OBEX, data pipes, and the CATC Merlin Analyzer.
- General purpose -- is integrated in a number of CATC products.

D.1 Values

There are five value types that may be manipulated by a script: **integers**, **strings**, **lists**, **raw bytes**, and `null`. CSL is not a strongly typed language. Value types need not be pre-declared. Literals, variables and constants can take on any of the five value types, and the types can be reassigned dynamically.

D.2 Literals

Literals are data that remain unchanged when the program is compiled. Literals are a way of expressing hard-coded data in a script.

Integers

Integer literals represent numeric values with no fractions or decimal points. Hexadecimal, octal, decimal, and binary notation are supported:

Hexadecimal numbers must be preceded by `0x`: `0x2A, 0x54, 0xFFFFFF01`

Octal numbers must begin with `0`: `0775, 017, 0400`

Decimal numbers are written as usual: `24, 1256, 2`

Binary numbers are denoted with `0b`: `0b01101100, 0b01, 0b100000`

Strings

String literals are used to represent text. A string consists of zero or more characters and can include numbers, letters, spaces, and punctuation. An *empty string* (`""`) contains no characters and evaluates to false in an expression, whereas a non-empty string evaluates to true. Double quotes surround a string, and some standard backslash (\) escape sequences are supported.

String	Represented text
`"Quote: \"This is a string literal.\""`	`Quote: "This is a string literal."`
`"256"`	`256` **Note that this does not represent the integer 256, but only the characters that make up the number.
`"abcd!$%&*"`	`abcd!$%&*`
`"June 26, 2001"`	`June 26, 2001`
`"[ 1, 2, 3 ]"`	`[ 1, 2, 3 ]`

Escape Sequences

These are the available escape sequences in CSL:

Character	Escape Sequence	Example	Output
backslash	`\\`	`"This is a backslash: \\"`	This is a backslash: \
double quote	`\"`	`"\"Quotes!\""`	"Quotes!"
horizontal tab	`\t`	`"Before tab\tAfter tab"`	Before tab After tab
newline	`\n`	`"This is how\nto get a newline."`	This is how to get a newline.
single quote	`\'`	`"\'Single quote\'"`	'Single quote'

Lists

A list can hold zero or more pieces of data. A list that contains zero pieces of data is called an *empty list*. An empty list evaluates to false when used in an expression, whereas a non-empty list evaluates to true. List literals are expressed using the square bracket (`[]`) delimiters. List elements can be of any type, including lists.

```
[1, 2, 3, 4]
[]
["one", 2, "three", [4, [5, [6]]]]
```

Raw Bytes

Raw binary values are used primarily for efficient access to packet payloads. A literal notation is supported using single quotes:

```
'00112233445566778899AABBCCDDEEFF'
```

This represents an array of 16 bytes with values starting at `00` and ranging up to `0xFF`. The values can only be hexadecimal digits. Each digit represents a nybble (four bits), and if there are not an even number of nybbles specified, an implicit zero is added to the first byte. For example:

```
'FFF'
```

is interpreted as

```
'0FFF'
```

Null

`Null` indicates an absence of valid data. The keyword `null` represents a literal `null` value and evaluates to false when used in expressions.

```
result = null;
```

D.3 Variables

Variables are used to store information, or data, that can be modified. A variable can be thought of as a container that holds a value.

All variables have names. Variable names must contain only alphanumeric characters and the underscore (_) character, and they cannot begin with a number. Some possible variable names are

```
x
_NewValue
name_2
```

A variable is created when it is assigned a value. Variables can be of any value type, and can change type with re-assignment. Values are assigned using the assignment operator (=). The name of the variable goes on the left side of the operator, and the value goes on the right:

```
x = [ 1, 2, 3 ]
New_value = x
name2 = "Smith"
```

If a variable is referenced before it is assigned a value, it evaluates to null.

There are two types of variables: global and local.

Global Variables

Global variables are defined outside of the scope of functions. Defining global variables requires the use of the keyword `set`. Global variables are visible throughout a file (and all files that it includes).

```
set Global = 10;
```

If an assignment in a function has a global as a left-hand value, a variable will not be created, but the global variable will be changed. For example

```
set Global = 10;

Function()
{
   Global = "cat";
   Local = 20;
}
```

will create a local variable called `Local`, which will only be visible within the function `Function`. Additionally, it will change the value of `Global` to `"cat",` which will be visible to all functions. This will also change its value type from an integer to a string.

Local Variables

Local variables are not declared. Instead, they are created as needed. Local variables are created either by being in a function's parameter list, or simply by being assigned a value in a function body.

```
Function(Parameter)
{
   Local = 20;
}
```

This function will create a local variable `Parameter` and a local variable `Local`, which has an assigned value of `20`.

D.4 Constants

A constant is similar to a variable, except that its value cannot be changed. Like variables, constant names must contain only alphanumeric characters and the underscore (_) character, and they cannot begin with a number.

Constants are declared similarly to global variables using the keyword `const`:

```
const CONSTANT = 20;
```

They can be assigned to any value type, but will generate an error if used in the left-hand side of an assignment statement later on. For instance,

```
const constant_2 = 3;

Function()
{
   constant_2 = 5;
}
```

will generate an error.

Declaring a constant with the same name as a global, or a global with the same name as a constant, will also generate an error. Like globals, constants can only be declared in the file scope.

D.5 Expressions

An expression is a statement that calculates a value. The simplest type of expression is assignment:

```
x = 2
```

The expression `x = 2` calculates 2 as the value of x.

All expressions contain operators, which are described in Section D.6, "Operators," on page 261. The operators indicate how an expression should be evaluated in order to arrive at its value. For example

```
x + 2
```

says to add 2 to x to find the value of the expression. Another example is

```
x > 2
```

which indicates that x is greater than 2. This is a Boolean expression, so it will evaluate to either true or false. Therefore, if `x = 3`, then `x > 2` will evaluate to true; if `x = 1`, it will return false.

True is denoted by a non-zero integer (any integer except 0), and false is a zero integer (0). True and false are also supported for lists (an empty list is false, while all others are true), and strings (an empty string is false, while all others are true), and null is considered false. However, all Boolean operators will result in integer values.

`select` expression

The `select` expression selects the value to which it evaluates based on Boolean expressions. This is the format for a `select` expression:

```
select {
   <expression1> : <statement1>
   <expression2> : <statement2>
   ...
};
```

The expressions are evaluated in order, and the statement that is associated with the first true expression is executed. That value is what the entire expression evaluates to.

```
x = 10
Value_of_x = select {
   x < 5 : "Less than 5";
   x >= 5 : "Greater than or equal to 5";
};
```

The above expression will evaluate to "Greater than or equal to 5" because the first true expression is `x >= 5`. Note that a semicolon is required at the end of a `select` expression because it is not a compound statement and can be used in an expression context.

There is also a keyword `default,` which in effect always evaluates to true. An example of its use is

```
Astring = select {
   A == 1 : "one";
   A == 2 : "two";
   A == 3: "three";
   A > 3 : "overflow";
   default : null;
```

```
};
```

If none of the first four expressions evaluates to true, then `default` will be evaluated, returning a value of `null` for the entire expression.

`select` expressions can also be used to conditionally execute statements, similar to C `switch` statements:

```
select {
   A == 1 : DoSomething();
   A == 2 : DoSomethingElse();
   default: DoNothing();
};
```

In this case the appropriate function is called depending on the value of A, but the evaluated result of the `select` expression is ignored.

D.6 Operators

An operator is a symbol that represents an action, such as addition or subtraction, that can be performed on data. Operators are used to manipulate data. The data being manipulated are called *operands*. Literals, function calls, constants, and variables can all serve as operands. For example, in the operation

```
x + 2
```

the variable `x` and the integer `2` are both operands, and `+` is the operator.

Operations can be performed on any combination of value types, but will result in a null value if the operation is not defined. Defined operations are listed in the Operand Types column of the table on page 264. Any binary operation on a null and a non-null value will result in the non-null value. For example, if

```
x = null;
```

then

```
3 * x
```

will return a value of 3.

A binary operation is an operation that contains an operand on each side of the operator, as in the preceding examples. An operation with only one operand is called a unary operation, and requires the use of a unary operator. An example of a unary operation is

```
!1
```

which uses the logical negation operator. It returns a value of 0.

The unary operators are `sizeof()`, `head()`, `tail()`, `~` and `!`.

Operator Precedence and Associativity

Operator rules of precedence and associativity determine in what order operands are evaluated in expressions. Expressions with operators of higher precedence are evaluated first. In the expression

```
4 + 9 * 5
```

the * operator has the highest precedence, so the multiplication is performed before the addition. Therefore, the expression evaluates to 49.

The associative operator `( )` is used to group parts of the expression, forcing those parts to be evaluated first. In this way, the rules of precedence can be overridden. For example,

```
( 4 + 9 ) * 5
```

causes the addition to be performed before the multiplication, resulting in a value of 65.

When operators of equal precedence occur in an expression, the operands are evaluated according to the associativity of the operators. This means that if an operator's associativity is left to right, then the operations will be done starting from the left side of the expression. So, the expression

```
4 + 9 - 6 + 5
```

would evaluate to 12. However, if the associative operator is used to group a part or parts of the expression, those parts are evaluated first. Therefore,

```
( 4 + 9 ) - ( 6 + 5 )
```

has a value of 2.

In the following table, the operators are listed in order of precedence, from highest to lowest. Operators on the same line have equal precedence, and their associativity is shown in the second column.

Operator Symbol	Associativity
`[]  ()`	Left to right
`~  !  sizeof  head  tail`	Right to left
`*  /  %`	Left to right

Operator Symbol	Associativity
++ --	Right to left
[] ()	Left to right
~ ! sizeof head tail	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< > <= >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
= += -= *= /= %= >>= <<= &= ^= |=	Right to left

Operator Symbol	Description	Operand Types	Result Types	Examples
Index Operator				
[ ]	Index or subscript	Raw Bytes	Integer	`Raw = '001122'` `Raw[1] = 0x11`

		List	Any	`List = [0, 1, 2, 3, [4, 5]]`
				`List[2] = 2`
				`List[4] = [4, 5]`
				`List[4][1] = 5`
				*Note: if an indexed Raw value is assigned to any value that is not a byte (> 255 or not an integer), the variable will be promoted to a list before the assignment is performed.

Associative Operator				
( )	Associative	Any	Any	`( 2 + 4 ) * 3 = 18` `2 + ( 4 * 3 ) = 14`
Arithmetic Operators				
*****	Multiplication	Integer-integer	Integer	`3 * 1 = 3`
/	Division	Integer-integer	Integer	`3 / 1 = 3`
%	Modulus	Integer-integer	Integer	`3 % 1 = 0`
+	Addition	Integer-integer	Integer	`2 + 2 = 4`
		String-string	String	`"one " + "two" = "one two"`
		Raw byte-raw byte	Raw	`'001122' + '334455' = '001122334455'`
		List-list	List	`[1, 2] + [3, 4] = [1, 2, 3, 4]`
		Integer-list	List	`1 + [2, 3] = [1, 2, 3]`
		Integer-string	String	`"number = " + 2 = "number = 2"` *Note: integer-string concatenation uses decimal conversion.
		String-list	List	`"one" + ["two"] = ["one", "two"]`
-	Subtraction	Integer-integer	Integer	`3 – 1 = 2`
Increment and Decrement Operators				
++	Increment	Integer	Integer	*a* = 1 ++*a* = 2 *b* = 1 *b*++ = 1 *Note that the value of *b* after execution is 2.
--	Decrement	Integer	Integer	*a* = 2 --*a* = 1 *b* = 2 *b*-- = 2 *Note that the value of *b* after execution is 1.

Operators

Operator Symbol	Description	Operand Types	Result Types	Examples
Equality Operators				
==	Equal	Integer-integer	Integer	`2 == 2`
		String-string	Integer	`"three" == "three"`
		Raw byte-raw byte	Integer	`'001122' == '001122'`
		List-list	Integer	`[1, [2, 3]] == [1, [2, 3]]` *Note: equality operations on values of different types will evaluate to false.
!=	Not equal	Integer-integer	Integer	`2 != 3`
		String-string	Integer	`"three" != "four"`
		Raw byte-raw byte	Integer	`'001122' != '334455'`
		List-list	Integer	`[1, [2, 3]] != [1, [2, 4]]` *Note: equality operations on values of different types will evaluate to false.
Relational Operators				
<	Less than	Integer-integer	Integer	`1 < 2`
		String-string	Integer	`"abc" < "def"`
>	Greater than	Integer-integer	Integer	`2 > 1`
		String-string	Integer	`"xyz" > "abc"`
<=	Less than or equal	Integer-integer	Integer	`23 <= 27`
		String-string	Integer	`"cat" <= "dog"`
>=	Greater than or equal	Integer-integer	Integer	`2 >= 1`
		String-string	Integer	`"sun" >= "moon"` *Note: relational operations on string values are evaluated according to character order in the ASCII table.
Logical Operators				
!	Negation	All combinations of types	Integer	`!0 = 1  !"cat" = 0` `!9 = 0  !"" = 1`
&&	Logical AND	All combinations of types	Integer	`1 && 1 = 1  1 && !"" = 1` `1 && 0 = 0  1 && "cat" = 1`
||	Logical OR	All combinations of types	Integer	`1 || 1 = 1  0 || 0 = 0` `1 || 0 = 1  "" || !"cat" = 0`

Operators (Continued)

Operator Symbol	Description	Operand Types	Result Types	Examples
Bitwise Logical Operators				
~	Bitwise complement	Integer-integer	Integer	`~0b11111110 = 0b00000001`
&	Bitwise AND	Integer-integer	Integer	`0b11111110 & 0b01010101 = 0b01010100`
^	Bitwise exclusive OR	Integer-integer	Integer	`0b11111110 ^ 0b01010101 = 0b10101011`
|	Bitwise inclusive OR	Integer-integer	Integer	`0b11111110 | 0b01010101 = 0b11111111`
Shift Operators				
<<	Left shift	Integer-integer	Integer	`0b11111110 << 3 = 0b11110000`
>>	Right shift	Integer-integer	Integer	`0b11111110 >> 1 = 0b01111111`
Assignment Operators				
=	Assignment	Any	Any	`A = 1` `B = C = A`
+=	Addition assignment	Integer-integer	Integer	*x* `= 1` *x* `+= 1 = 2`
		String-string	String	*a* `= "one "` *a* `+= "two" = "one two"`
		Raw byte-raw byte	Raw	*z* `= '001122'` *z* `+= '334455' = '001122334455'`
		List-list	List	*x* `= [1, 2]` *x* `+= [3, 4] = [1, 2, 3, 4]`
		Integer-list	List	*y* `= 1` *y* `+= [2, 3] = [1, 2, 3]`
		Integer-string	String	*a* `= "number = "` *a* `+= 2 = "number = 2"` *Note: integer-string concatenation uses decimal conversion.
		String-list	List	*s* `= "one"` *s* `+ ["two"] = ["one", "two"]`
-=	Subtraction assignment	Integer-integer	Integer	*y* `= 3` *y* `–= 1 = 2`
***=**	Multiplication assignment	Integer-integer	Integer	*x* `= 3` *x* `*= 1 = 3`
/=	Division assignment	Integer-integer	Integer	*s* `= 3` *s* `/= 1 = 3`
%=	Modulus assignment	Integer-integer	Integer	*y* `= 3` *y* `%= 1 = 0`
>>=	Right shift assignment	Integer-integer	Integer	*b* `= 0b11111110` *b* `>>= 1 = 0b01111111`
<<=	Left shift assignment	Integer-integer	Integer	*a* `= 0b11111110` *a* `<<= 3 = 0b11111110000`

Operators (Continued)

Operator Symbol	Description	Operand Types	Result Types	Examples
Assignment Operators (continued)				
&=	Bitwise AND assignment	Integer-integer	Integer	*a* = 0b11111110 *a* &= 0b01010101 = 0b01010100
^=	Bitwise exclusive OR assignment	Integer-integer	Integer	*e* = 0b11111110 *e* ^= 0b01010101 = 0b10101011
|=	Bitwise inclusive OR assignment	Integer-integer	Integer	*i* = 0b11111110 *i* |= 0b01010101 = 0b11111111
List Operators				
sizeof()	Number of elements	Any	Integer	sizeof([1, 2, 3]) = 3 sizeof('0011223344') = 5 sizeof("string") = 6 sizeof(12) = 1 sizeof([1, [2, 3]]) = 2 *Note: the last example demonstrates that the sizeof() operator returns the shallow count of a complex list.
head()	Head	List	Any	head([1, 2, 3]) = 1 *Note: the Head of a list is the first item in the list.
tail()	Tail	List	List	tail([1, 2, 3]) = [2, 3] *Note: the Tail of a list includes everything except the Head.

Operators (Continued)

D.7 Comments

Comments may be inserted into scripts as a way of documenting what the script does and how it does it. Comments are useful as a way to help others understand how a particular script works. Additionally, comments can be used as an aid in structuring the program.

Comments in CSL begin with a hash mark (#) and finish at the end of the line. The end of the line is indicated by pressing the Return or Enter key. Anything contained inside the comment delimiters is ignored by the compiler. Thus,

```
# x = 2;
```

is not considered part of the program. CSL supports only end-of-line comments, which means that comments can be used only at the end of a line or on their own line. It's not possible to place a comment in the middle of a line.

Writing a multi-line comment requires surrounding each line with the comment delimiters

```
# otherwise the compiler would try to interpret
# anything outside of the delimiters
# as part of the code.
```

The most common use of comments is to explain the purpose of the code immediately following the comment. For example:

```
# Add a profile if we got a server channel
if(rfChannel != "Failure")
{
   result =
SDPAddProfileServiceRecord(rfChannel,
"ObjectPush");
   Trace("SDPAddProfileServiceRecord returned ",
result, "\n");
}
```

D.8 Keywords

Keywords are reserved words that have special meanings within the language. They cannot be used as names for variables, constants or functions.

In addition to the operators, the following are keywords in CSL:

Keyword	Usage
`select`	`select` expression
`set`	define a global variable
`const`	define a constant
`return`	`return` statement
`while`	`while` statement
`for`	`for` statement
`if`	`if` statement
`else`	`if-else` statement
`default`	`select` expression
`null`	null value
`in`	input context
`out`	output context

D.9 Statements

Statements are the building blocks of a program. A program is made up of list of statements.

Seven kinds of statements are used in CSL: expression statements, if statements, if-else statements, while statements, for statements, return statements, and compound statements.

Expression Statements

An expression statement describes a value, variable, or function.

```
<expression>
```

Here are some examples of the different kinds of expression statements:

```
Value: x + 3;
Variable: x = 3;
Function: Trace ( x + 3 );
```

The variable expression statement is also called an *assignment statement*, because it assigns a value to a variable.

`if` Statements

An `if` statement follows the form

`if <expression> <statement>`

For example,

```
if (3 && 3) Trace("True!");
```

will cause the program to evaluate whether the expression `3 && 3` is nonzero, or True. It is, so the expression evaluates to True and the `Trace` statement will be executed. On the other hand, the expression `3 && 0` is not nonzero, so it would evaluate to False, and the statement wouldn't be executed.

`if-else` Statements

The form for an `if-else` statement is

```
if <expression> <statement1>
else <statement2>
```

The following code

```
if ( 3 - 3 || 2 - 2 ) Trace ( "Yes" );
else Trace ( "No" );
```

will cause "No" to be printed, because `3 - 3 || 2 - 2` will evaluate to False (neither `3 - 3` nor `2 - 2` is nonzero).

`while` Statements

A `while` statement is written as

```
while <expression> <statement>
```

An example of this is

```
x = 2;
while ( x < 5 )
{
   Trace ( x, ", " );
   x = x + 1;
}
```

The result of this would be

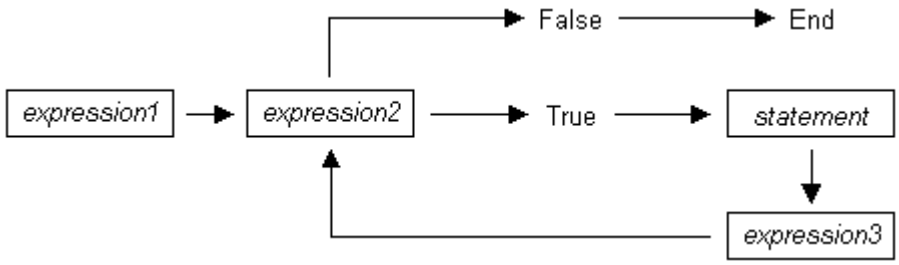
```
2, 3, 4,
```

`for` Statements

A `for` statement takes the form

```
for (<expression1>; <expression2>;
<expression3>) <statement>
```

The first expression initializes, or sets, the starting value for *x*. It is executed one time, before the loop begins. The second expression is a conditional expression. It determines whether the loop will continue -- if it evaluates true, the function keeps executing and proceeds to the statement; if it evaluates false, the loop ends. The third expression is executed after every iteration of the statement.



The example

```
for ( x = 2; x < 5; x = x + 1 ) Trace ( x, "\n" );
```

would output

```
2
3
4
```

The example above works out like this: the expression `x = 2` is executed. The value of *x* is passed to `x < 5,` resulting in `2 < 5`. This evaluates to true, so the statement `Trace (x, "\n" )` is performed, causing `2` and a new line to print. Next, the third expression is executed, and the value of *x* is increased to 3. Now, `x < 5` is executed again, and is again true, so the `Trace` statement is executed, causing 3 and a new line to print. The third expression increases the value of *x* to 4; `4 < 5` is true, so 4 and a new line are printed by the `Trace` statement. Next, the value of *x* increases to `5`. `5 < 5` is *not* true, so the loop ends.

`return` Statements

Every function returns a value, which is usually designated in a `return` statement. A `return` statement returns the value of an expression to the calling environment. It uses the following form:

```
return <expression>
```

An example of a `return` statement and its calling environment is

```
Trace ( HiThere() );
...
HiThere()
{
   return "Hi there";
}
```

The call to the primitive function `Trace` causes the function `HiThere()` to be executed. `HiThere()` returns the string "Hi there" as its value. This value is passed to the calling environment (`Trace`), resulting in this output:

```
Hi there
```

A `return` statement also causes a function to stop executing. Any statements that come after the `return` statement are ignored, because `return` transfers control of the program back to the calling environment. As a result,

```
Trace ( HiThere() );
...
HiThere()
{
   a = "Hi there";
   return a;
   b = "Goodbye";
   return b;
}
```

will output only

```
Hi there
```

because when `return a;` is encountered, execution of the function terminates, and the second return statement (`return b;`) is never processed. However,

```
Trace ( HiThere() );
```

```
...
HiThere()
{
   a = "Hi there";
   b = "Goodbye";
   if ( 3 != 3 ) return a;
   else return b;
}
```

will output

```
Goodbye
```

because the `if` statement evaluates to false. This causes the first `return` statement to be skipped. The function continues executing with the `else` statement, thereby returning the value of `b` to be used as an argument to `Trace`.

Compound Statements

A compound statement, or *statement block*, is a group of one or more statements that is treated as a single statement. A compound statement is always enclosed in curly braces (`{}`). Each statement within the curly braces is followed by a semicolon; however, a semicolon is not used following the closing curly brace.

The syntax for a compound statement is

```
{
   <first_statement>;
   <second_statement>;
   ...
   <last_statement>;
}
```

An example of a compound statement is

```
{
   x = 2;
   x + 3;
}
```

It's also possible to nest compound statements, like so:

```
{
   x = 2;
   {
```

```
        y = 3;
    }
    x + 3;
}
```

Compound statements can be used anywhere that any other kind of statement can be used.

```
if (3 && 3)
{
    result = "True!";
    Trace(result);
}
```

Compound statements are required for function declarations and are commonly used in `if`, `if-else`, `while`, and `for` statements.

D.10 Preprocessing

The preprocessing command `%include` can be used to insert the contents of a file into a script. It has the effect of copying and pasting the file into the code. Using `%include` allows the user to create modular script files that can then be incorporated into a script. This way, commands can easily be located and reused.

The syntax for `%include` is this:

```
%include "includefile.inc"
```

The quotation marks around the filename are required, and by convention, the included file has a .inc extension.

The filenames given in the include directive are always treated as being relative to the current file being parsed. So, if a file is referenced via the preprocessing command in a .script file, and no path information is provided (`%include "file.inc"`), the application will try to load the file from the current directory. Files that are in a directory one level up from the current file can be referenced using `"..\file.inc"`, and likewise, files one level down can be referenced using the relative pathname (`"directory\file.inc"`). Last but not least, files can also be referred to using a full pathname, such as `"C:\global_scripts\include\file.inc"`.

D.11 Functions

A function is a named statement or a group of statements that are executed as one unit. All functions have names. Function names must contain only alphanumeric characters and the underscore (_) character, and they cannot begin with a number.

A function can have zero or more *parameters*, which are values that are passed to the function statement(s). Parameters are also known as *arguments*. Value types are not specified for the arguments or return values. Named arguments are local to the function body, and functions can be called recursively.

The syntax for a function declaration is

```
name(<parameter1>, <parameter2>, ...)
{
   <statements>
}
```

The syntax to call a function is

```
name(<parameter1>, <parameter2>, ...)
```

So, for example, a function named `add` can be declared like this:

```
add(x, y)
{
   return x + y;
}
```

and called this way:

```
add(5, 6);
```

This would result in a return value of 11.

Every function returns a value. The return value is usually specified using a `return` statement, but if no `return` statement is specified, the return value will be the value of the last statement executed.

Arguments are not checked for appropriate value types or number of arguments when a function is called. If a function is called with fewer arguments than were defined, the specified arguments are assigned, and the remaining arguments are assigned to null. If a function is called with more arguments than were defined, the extra arguments are ignored. For example, if the function `add` is called with just one argument

```
add(1);
```

the parameter `x` will be assigned to 1, and the parameter `y` will be assigned to null, resulting in a return value of 1. But if `add` is called with more than two arguments

```
add(1, 2, 3);
```

`x` will be assigned to 1, `y` to 2, and 3 will be ignored, resulting in a return value of 3.

All parameters are passed by value, not by reference, and can be changed in the function body without affecting the values that were passed in. For instance, the function

```
add_1(x, y)
{
   x = 2;
   y = 3;
   return x + y;
}
```

reassigns parameter values within the statements. So,

```
a = 10;
b = 20;
add_1(a, b);
```

will have a return value of 5, but the values of a and b won't be changed.

The scope of a function is the file in which it is defined (as well as included files), with the exception of primitive functions, whose scopes are global.

Calls to undefined functions are legal, but will always evaluate to null and result in a compiler warning.

D.12 Primitives

Primitive functions are called similarly to regular functions, but they are implemented outside of the language. Some primitives support multiple types for certain arguments, but in general, if an argument of the wrong type is supplied, the function will return null.

Call()

Call(<function_name *string*>, <arg_list *list*>)

Parameter	Meaning	Default Value	Comments
function_name *string*			
arg_list *list*			Used as the list of parameters in the function call.

Return value

Same as that of the function that is called.

Comments

Calls a function whose name matches the `function_name` parameter. All scope rules apply normally. Spaces in the `function_name` parameter are interpreted as the '_' (underscore) character since function names cannot contain spaces.

Example

```
Call("Format", ["the number is %d", 10]);
```

is equivalent to:

```
Format("the number is %d", 10);
```

Format()

Format (<format *string*>, <value *string* or *integer*>)

Parameter	Meaning	Default Value	Comments
format *string*			
value *string* or *integer*			

Return value

None.

Comments

`Format` is used to control the way that arguments will print out. The format string may contain conversion specifications that affect the way in which the arguments in the value string are returned. Format conversion characters, flag characters, and field width modifiers are used to define the conversion specifications.

Example

```
Format("0x%02X", 20);
```

would yield the string `0x14`.

`Format` can only handle one value at a time, so

```
Format("%d %d", 20, 30);
```

would not work properly. Furthermore, types that do not match what is specified in the format string will yield unpredictable results.

Format Conversion Characters

These are the format conversion characters used in CSL:

Code	Type	Output
`c`	Integer	Character
`d`	Integer	Signed decimal integer.
`i`	Integer	Signed decimal integer
`o`	Integer	Unsigned octal integer
`u`	Integer	Unsigned decimal integer
`x`	Integer	Unsigned hexadecimal integer, using "abcdef."
`X`	Integer	Unsigned hexadecimal integer, using "ABCDEF."
`s`	String	String

A conversion specification begins with a percent sign (%) and ends with a conversion character. The following optional items can be included, in order, between the % and the conversion character to further control argument formatting:

- Flag characters are used to further specify the formatting. There are five flag characters:
  - A minus sign (`-`) will cause an argument to be left-aligned in its field. Without the minus sign, the default position of the argument is right-aligned.
  - A plus sign will insert a plus sign (`+`) before a positive signed integer. This only works with the conversion characters `d` and `i`.
  - A space will insert a space before a positive signed integer. This only works with the conversion characters `d` and `i`. If both a space and a plus sign are used, the space flag will be ignored.
  - A hash mark (`#`) will prepend a `0` to an octal number when used with the conversion character `o`. If `#` is used with `x` or `X`, it will prepend `0x` or `0X` to a hexadecimal number.
  - A zero (`0`) will pad the field with zeros instead of with spaces.

- Field width specification is a positive integer that defines the field width, in spaces, of the converted argument. If the number of characters in the argument is smaller than the field width, then the field is padded with spaces. If the argument has more characters than the field width has spaces, then the field will expand to accommodate the argument.

GetNBits()

`GetNBits (<bit_source` *`list`* `or` *`raw`*`>, <bit_offset` *`integer`*`>, <bit_count` *`integer`*`>)`

Parameter	Meaning	Default Value	Comments
bit_source *list*, *raw*, or *integer*			Can be an integer value (4 bytes) or a list of integers that are interpreted as bytes.
bit_offset *integer*	Index of bit to start reading from		
bit_count	Number of bits to read		

Return value

None.

Comments

Reads `bit_count` bits from `bit_source` starting at `bit_offset`. Will return null if `bit_offset` + `bit_count` exceeds the number of bits in `bit_source`. If `bit_count` is 32 or less, the result will be returned as an integer. Otherwise, the result will be returned in a list format that is the same as the input format. `GetNBits` also sets up the bit data source and global bit offset used by `NextNBits`. Note that bits are indexed starting at bit 0.

Example

```
raw = 'F0F0';  # 1111000011110000 binary
result = GetNBits ( raw, 2, 4 );
Trace ( "result = ", result );
```

The output would be

```
   result = C     # The result is given in hexadecimal. The
result in binary is 1100
```

In the call to `GetNBits`: starting at bit 2, reads 4 bits (1100) and returns the value 0xC.

NextNBits()

NextNBits (<bit_count *integer*>)

Parameter	Meaning	Default Value	Comments
bit_count *integer*			

Return value

None.

Comments

Reads `bit_count` bits from the data source specified in the last call to `GetNBits`, starting after the last bit that the previous call to `GetNBits` or `NextNbits` returned. If called without a previous call to `GetNBits`, the result is undefined. Note that bits are indexed starting at bit 0.

Example

```
raw = 'F0F0';  # 1111000011110000 binary
result1 = GetNBits ( raw, 2, 4 );
result2 = NextNBits(5);
result3 = NextNBits(2);
Trace ( "result1 = ", result1 " result2 = ", result2 "
result3 = ", result3 );
```

This will generate this trace output:

```
result1 = C result2 = 7 result3 = 2
```

In the call to `GetNBits`: starting at bit 2, reads 4 bits (1100), and returns the value 0xC.

In the first call to `NextNBits`: starting at bit 6, reads 5 bits (00111), and returns the value 0x7.

In the second call to `NextNBits`: starting at bit 11 (=6+5), reads 2 bits (10), and returns the value 0x2.

Resolve()

Resolve(<symbol_name *string*>)

Parameter	Meaning	Default Value	Comments
symbol_name *string*			

Return value

The value of the symbol. Returns null if the symbol is not found.

Comments

Attempts to resolve the value of a symbol. Can resolve global, constant, and local symbols. Spaces in the `symbol_name` parameter are interpreted as the '_' (underscore) character since function names cannot contain spaces.

Example

```
a = Resolve( "symbol" );
```

is equivalent to:

```
a = symbol;
```

Trace()

`Trace( <arg1 any>, <arg2 any>, ... )`

Parameter	Meaning	Default Value	Comments
arg *any*			The number of arguments is variable.

Return value

None.

Comments

The values given to this function are given to the debug console.

Example

```
list = ["cat", "dog", "cow"];
Trace("List = ", list, "\n");
```

would result in the output

```
List = [cat, dog, cow]
```