plantronics.

# Plantronics® SDK v3.6 API Specification

September 8, 2015

http://developer.plantronics.com/community/sdk

# PREFACE

## About Plantronics®

Plantronics is a global leader in audio communications for businesses and consumers. We have pioneered new trends in audio technology, creating innovative products that allow people to simply communicate. From unified communication to Bluetooth® headsets to gaming solutions, we deliver uncompromising quality, an ideal experience, and extraordinary service. Plantronics is used by every company in the Fortune 100™, as well as 911 dispatch, air traffic control and various mission-critical applications for those on the frontline. We have active and collaborative alliances with the leading UC providers for enterprise, including Avaya, Cisco, and Microsoft, as well as consumer VoIP providers to ensure seamless interoperability with our product portfolio. For more information, please visit www.plantronics.com or call (800) 544-4660.

## About this Document

This document describes Plantronics® SDK v3.6. It describes how to extend or develop client applications to interact with Plantronics devices and softphones using Plantronics SDK APIs. If you are looking at a locally downloaded copy, please refer to the Plantronics web site for the latest version: http://developer.plantronics.com/docs/DOC-1557.

## More Information/Product Support

For more information or for product support, visit the Plantronics Developer Connection at http://developer.plantronics.com/community/sdk.

## What's New in the Plantronics SDK v3.4 API

The following APIs were added since the release of the Plantronics SDK API v3.2:

- IHostCommand.h
  - *IDeviceSettingsExt* Interface

The following changes were made since the release of the Plantronics SDK API v3.3:

- The name of the Spokes runtime and client executable file changed from `PLTSpokes.exe` to `PLTHub.exe`.
- Added `SpokesSDKNativeRuntime.msi`

## What's New in the Plantronics SDK v3.5 API

The following changes were made since the release of the Plantronics SDK v3.4:

- Added VERSION_TYPE_PIC for enum *VersionType*
- Added BASE_EVENT_TYPE_MOBILE_CONNECTED and BASE_EVENT_TYPE_MOBILE_DISCONNECTED for the *BaseEventType* enum.
- Added BASE_STATE_CHANGE_MOBILE_CONNECTED and BASE_STATE_CHANGE_MOBILE_DISCONNECTED for the *BaseStateChange* enum
- Added DEVICE_RING_TONE_OFF to the *eDeviceRingTone* enum
- Added DD_SOFTPHONE_ID_SWYX to the *DDSoftphoneID* enum
- Added BASE_EVENT_TYPE enum

- Added `isLineConnected(eLineType i_lineType, bool &o_active)` to the *iHostCommandExt* interface.

## What's New in the Plantronics SDK v3.6 API

The following changes were made since the release of the Plantronics SDK v3.5:

- DeviceManager interface:
  - Add INTF IDeviceBaseExt
  - Remove from IDevice interface: virtual IDeviceExt *getDeviceExtension
  - Remove: #include "IConnectedDevice.h" (for internal use only)
- DMEnums:
  - Add typedef enum CommandOption
- IHostCommand:
  - Add IHostCommandOption interface
- Five new SDK samples added:
  - RestJsClientSample
  - Spokes3GCOMSample
  - SpokesNativeSample
  - SpokesSDKCOMNETSample
  - MobileCallNative

# *Table of Contents*

# Chapter 1: About the Plantronics® SDK

Plantronics has developed the Plantronics® Software Development Kit (SDK) for Windows and Mac to help developers take advantage of the new opportunities available with context-enabled devices. With Contextual Intelligence, applications become smarter because they benefit from understanding what the user is doing. Tapping into contextual information available in headsets allows developers to create more compelling business solutions.

The Plantronics Software Developers Kit (SDK) provides information such as mobile phone caller ID, mobile phone call state, proximity to a connected computer system, and headset wearing state. By incorporating information from the physical world, context-aware software can dramatically improve business productivity.

The SDK is the entry point for all external application interactions. This document describes all the interfaces that the Plantronics SDK exposes to applications that interact with Plantronics devices (current and future). The interfaces extend to a wide variety of applications, such as:

- softphones
- media players
- applications that want to know about device state
- send commands to devices

The interfaces also command the device and capture device event notification. In addition, through a metadata interface, Plantronics SDK exposes an application to such functionality as client sessions in progress, calls active, number of calls on hold, and so on.

## 1.1: What is Contextual Intelligence?

In today's enterprise, more and more employees are working from locations outside of the traditional office, so they need to be able to communicate and collaborate effortlessly and securely. In this new distributed and increasingly social work environment, more is needed from a communication solutions to give users a more satisfying and productive experience.

With contextual intelligence, applications become smarter because they benefit from understanding what the users are doing — where they are, who they're near, if they're busy, what device they're using, and more.

The emergence of context-enabled devices such as smart phones and sensor-enabled headsets is enabling a new class of applications that have the ability to securely and privately access information about users and what they are doing in real time. These intelligent solutions promise to deliver a more integrated and seamless communication environment that enhances productivity. Now, application developers can access the intelligence built into Plantronics smart devices to deliver innovative solutions that enhance communication-enabled business processes.

Tapping into contextual information available in headsets allows developers to create more compelling business solutions. With the Plantronics Software Development Kit (SDK) for Windows and Mac, independent software vendors (ISVs), system integrators (SIs), and enterprise developers can quickly integrate wearable technology into their applications.

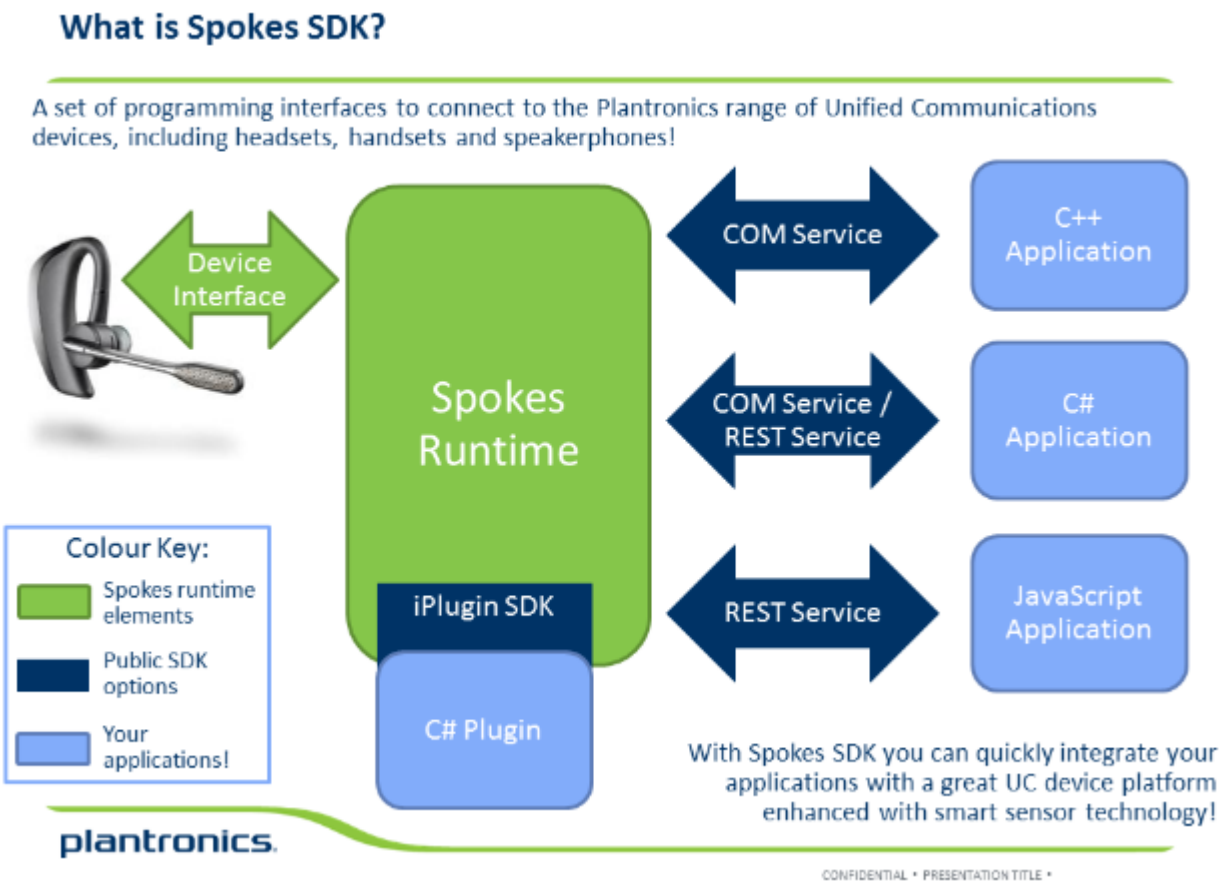
With the ability to create software that interfaces with wearable technologies, developers can explore new approaches to enhancing communications and collaboration in business applications. By utilizing the data provided by Plantronics headsets, developers can create innovative solutions that are integrated with applications and business processes to add business value. The following list provides a few examples of the types of functions that can be added to enhance enterprise applications.

- Streamline user interaction by sending audio notifications and alerts directly to the device.
- Personalize your application's user experience by listening for device events that tell you what a user is doing.

- Control Unified Communications (UC) applications such as conferencing and voicemail from the device to create a more natural user experience.
- Improve collaboration by altering presence to truly reflect what a worker is doing.
- Bring visibility to the mobile blind spot by listening for mobile caller ID on the computer system.
- Use the mobile caller ID in screen pop, presence and metrics applications or any other way you can imagine. For example, control the mobile phone from the computer system to keep users productive.
- Access event logs to better understand a user's activities across computer system and mobile as well as accessing contextual history.
- Trigger behaviors in your applications based on user actions such as when a user approaches his desk or leaves it. Locking the computer system, dimming the screen and changing presence are great ways to enhance security, reduce costs, go green and improve productivity.
- Extend call control functions (answer, end, ring, mute, volume control) to user devices, thus enhancing the overall user experience.

## 1.2: What are the Plantronics SDK APIs?

The Plantronics SDK APIs are a set of programming interfaces that enable you to connect to the Plantronics range of Unified Communications devices, including headsets, handsets, and speakerphones.



## 1.3: Understanding the Plantronics SDK Architecture

The Plantronics SDK is a software stack that gives developers access to Plantronics headsets. A variety of interfaces are exposed that provide access to different types of device, environment, and session functionality, including call interface, device/base state and control, contact, notifications/alerts, and media control. With these APIs, developers can control devices, capture event notifications, and integrate this information with applications. The Plantronics SDK is organized logically based on the type of capability offered: Plug-in, Session Manager, Call Manager, Device Listener, Contact Manager, Call Log, and Alerts.

Figure 1 shows the software and hardware stack for Plantronics SDK.

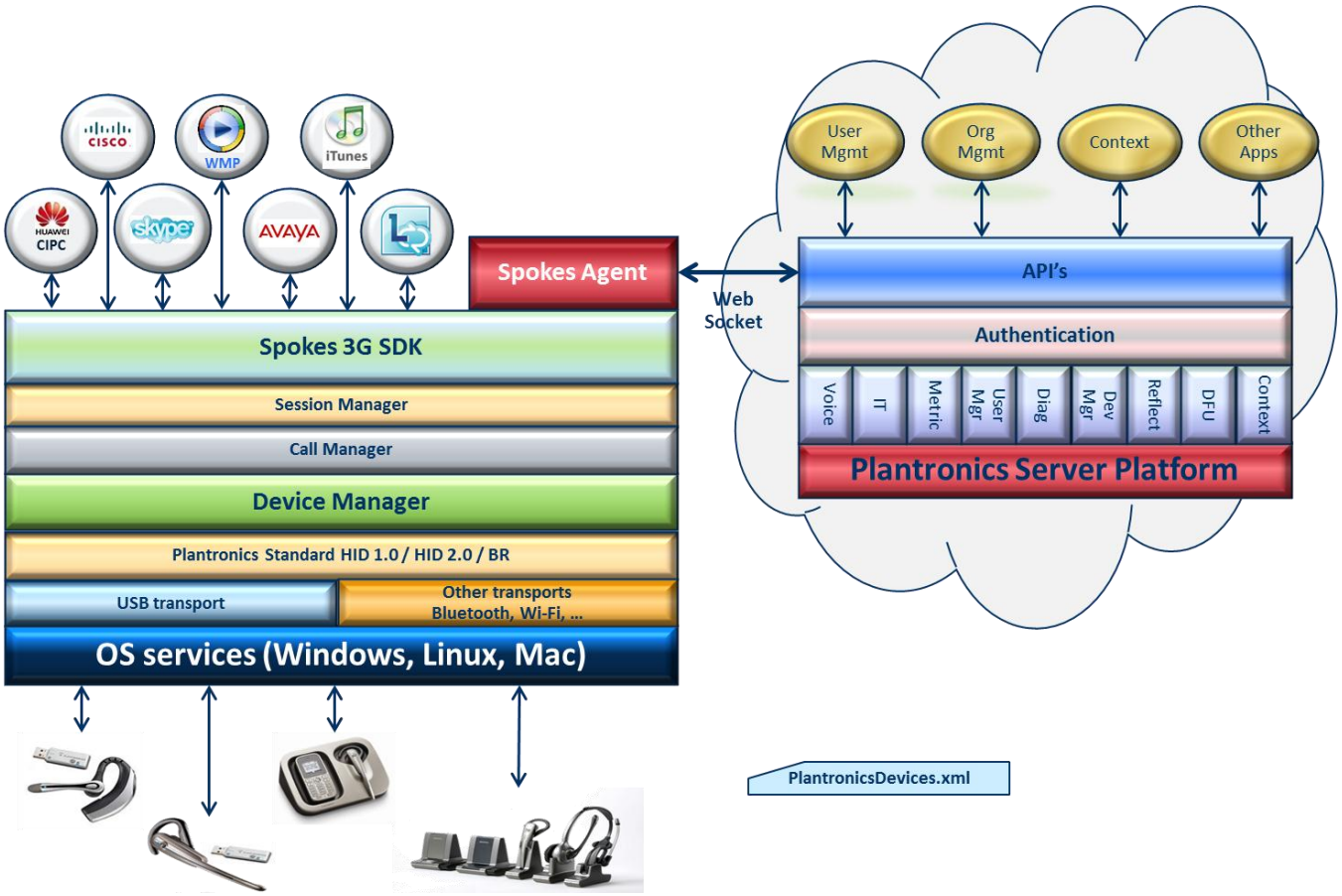



**Figure 1: Plantronics SDK v Architecture**

The Plantronics SDK software and hardware stack has the following functional modules.

| Module | Description |
|---|---|
| Plantronics SDK Client applications | Client applications that implement the Plantronics SDK API, such as softphones and media players. |

| | |
|---|---|
| Spokes DLL | Entry point for all external application interactions. The interfaces extend to a wide variety of applications, such as softphones and media players. The interfaces also command the device and capture device event notification. In addition, through a metadata interface, Spokes exposes an application to such functionality as client sessions in progress, calls active, number of calls on hold, and so on.<br><br>Plantronics SDK Interfaces are described in Chapter 2:Using Plantronics SDK Native Interfaces.<br><br>This is a change from Spokes 2.x, and is described more in Appendix D: Architectural Differences. |
| Session Manager | The Session Manager helps applications manage incoming calls by handling client requests and maintaining a list of client sessions. In addition, the Session Manager interfaces with the Call Manager and the Device Listener, and propagates events and commands to and from the Spokes clients. The Session Manager exposes interfaces for call handling, device state and events, call log, contact management, and media player control. There can be many different contact sources, such as Outlook, softphone, or a UC client. The Contact Provider module resides outside of Spokes and interacts with the contact management interface exposed by the Session Manager.<br><br>A plug-in uses the *iSessionManager* interface and the *iSessionManagerEvents* interface to send to and receive session information from the Session Manager.<br><br>The Session Manager is described in 0 See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.<br><br>ISessionManager Interface. |
| Call Manager | The Call Manager provides call control functionality. Through this interface, applications can manage multiple calls to the same device, identify active and on-hold calls, initiate a redial, or switch channels on a Bluetooth device when necessary.<br><br>The Call Manager also handles outgoing calls initiated from the device. Devices with keypad, contact lists, or call logs can make call requests to the Device Manager. Once the Device Listener receives the call request, it sends it to the Call Manager. The Call Manager can support multiple calls from the same softphone or from different softphones, and can also manage instances of active and held calls.<br><br>The *ICallCommand* Interface and the *ICallEvents* Interface are used to send to and receive call notifications from the Call Manager.<br><br>Call Manager API's are described in Chapter 2: Using Plantronics SDK Native Interfaces. |
| Device Manager | The Device Manager provides device level interfaces to events, commands, and properties, and handles the interaction between devices and applications. Through this interface, applications can detect actions taken on the device, such as button presses, and also monitor status information (for example, proximity signals and battery capacity). The |

| | |
|---|---|
| | Device Manager also enables applications to send commands to devices, such as ringing a headset.<br><br>The Device Manager interfaces with a human interface device (HID) layer to communicate with USB devices. The HID layer provides USB device communication and management by calling Win32 APIs using Platform Invoke services (Pinvoke) and is exposed only to the Device Manager.<br><br>0 See Also: Section 4.1.1.1: DMResult enum.<br><br>Using Device Manager Interfaces describes this interface. |
| Device Listener | The Device Listener is not separately called out in this diagram because it is a part of the Device Manager, but is described here to provide a better overall picture of the architecture.<br><br>The Device Listener provides a consistent interface across all Plantronics devices, so applications need not handle device-specific events and commands. Some Plantronics devices support multiple interfaces, for example, Voice over Internet Protocol (VoIP), public switched telephone network (PTSN), and mobile. These devices may require device-specific commands and events. The Device Manager exposes all of the device-specific events and commands supported by these devices.<br><br>0 See Also: Section 4.1.1.1: DMResult enum.<br><br>Using Device Manager Interfaces discusses the APIs. |
| PLT HID | Human Interface Driver (HID) 1.0, HID 2.0, and Bluetooth for USB connections. |
| Transport Layer | Bluetooth, Wi-Fi, USB, or other transport. |
| OS Services | OS Services include Win 32/Win 64 and Mac OS. |
| PlantronicsDevices.xml | Configuration file for all custom device handlers. List of Plantronics devices and their attributes (product ID and name, device and host command event handlers, device handler, and other settings). |

## 1.4: Plantronics SDK API Interfaces

On the Windows platform, Plantronics SDK v3.6 is compatible with the 32-bit Windows XP operating system and 32-bit and 64-bit Vista, Windows 7 and Windows 8 operating systems. On the Mac operating system, Plantronics SDK v3.6 is compatible with 10.7 or later. It is offered free of any licensing fee.

Developers are requested to complete a profile on the Plantronics Developer Connection (PDC) to access forums, support, and documentation on the PDC and to be contacted with pertinent information, for example, news about upgrades.

The Plantronics SDK provides access to devices through several interfaces (Native, COM, and REST), enabling developers to select the level of integration desired with the Plantronics platform. Depending on which interface is used, developers have different application deployment options.

The Plantronics SDK has the following application integration options:

NOTICE: For all deployments other than #1, the Spokes runtime engine, `PLTHub.exe`, is needed.

1. Developers can build native C++ applications, link with import library Spokes.lib (runtime `Spokes.dll` will be needed), and build an application.
2. Developers can build C++ COM applications, use Plantronics.tlb, and do out-of-proc integration with the COM Server hosted in `PLTHub.exe`.
3. Developers can build .NET managed applications, use Primary Interop assembly, and do out-of-proc integration with the COM Server hosted in `PLTHub.exe`.
4. Developers can build REST applications, and do out-of-proc integration with the REST Services plug-in hosted by `PLTHub.exe`.

Appendix A: Plantronics SDK v3.6 Feature Parity breaks down each interface by function, and then displays a checklist of supported application deployment options for that function.

### *1.4.1: Native Plantronics SDK Library API*

The native Plantronics SDK API allows any application to run with only `Spokes.dll`, independent of the Spokes runtime. Applications utilizing the native API can be deployed with only `Spokes.dll` to a computer system with or without Plantronics SDK preinstalled.

The native library utilizes C++ to develop cross-platform applications.

The native Plantronics SDK API is described in the following chapters of this document:

- Chapter 2:Using Plantronics SDK Native Interfaces

0 See Also: Section 4.1.1.1: DMResult enum.

- Using Device Manager Interfaces
- Chapter 4: Using Device Listener Interfaces
- Chapter 5: Using Host Command Interfaces
- Chapter 6: Using Device Events Interfaces

## 1.5: For More Information

The following links provide more information on the Plantronics SDK. A Plantronics Developer Connection (PDC) account is required to access these sites.

- Plantronics SDK White Paper at http://developer.plantronics.com/docs/DOC-1045
- The http://developer.plantronics.com/community/sdk provides links to sample code, getting started documentation, SDK downloads, video tutorials, and more.

### *1.5.1: COM APIs*

The Plantronics SDK provides a COM interface that represents a single, unified API and makes the same level of functionality available to both managed and unmanaged Windows-platform COM applications. To achieve this, Spokes acts as a COM Server, and a set of interfaces (such as *ICOMSessionManager*, *ICOMSession*, and *ICOMSessionManagerEvents*) that represent the Plantronics SDK are exposed to COM interop. For more information on these API's, please read

0See Also: Section 4.1.1.1: DMResult enum.

- Using Device Manager Interfaces
- Chapter 4:Using Device Listener Interfaces
- Chapter 5:Using Host Command Interfaces
- Chapter 6:Using Device Events Interfaces

### *1.5.2: RESTful APIs*

Representational State Transfer (REST) is an architecture style for designing distributed applications. The REST service plug-in hosts a REST Service to expose certain features of the Device Manager SDK and the Plantronics SDK. The Plantronics SDK REST Service can be used by business applications (Web based or thick client) to interface with the Spokes runtime engine via a RESTful interface on the client computer system.

These APIs are discussed in the following sections:

- Using the Plantronics SDK REST Service
- Appendix E: REST in Plantronics SDK v3.x: Migration Guide

# Chapter 2: Using Plantronics SDK Native Interfaces

The interfaces in this group contain all of the abstract classes for the Plantronics SDK.

## 2.1: ICall Interface

Use the *ICall* interface to provide Spokes with the Softphone-assigned *CallID* for a call.

```
INTF ICall
    {
    virtual int32_t getID()=0;
    virtual void setID(int32_t id)=0;
    virtual int32_t getConferenceID()=0;
    virtual void setConferenceID(int32_t id)=0;
    virtual bool getInConference()=0;
    virtual void setInConference(bool bConference)=0;
    };
```

### *2.1.1: int32_t getID()*

This method allows reading and setting the ID of an instance of a call.

### *2.1.2: void setID(int32_t id)*

This method allows reading and setting the ID of an instance of a call.

### *2.1.3: int32_t getConferenceID()*

This method allows reading and setting the conference mode of an instance of a call.

### *2.1.4: void setConferenceID(int32_t id)*

This method allows reading and setting the ID of the conference of an instance of a call.

### *2.1.5: bool getInConference()*

This method returns true if the call is In-Conference with other calls and false otherwise.

### *2.1.6: void setInConference(bool bConference)*

This method sets the Conference state of a call to true if the call is in-conference and to false otherwise.

## 2.2: ICallCommand Interface

*ICallCommand* is the interface used to communicate call-related information to the Session Manager and the Call Manager. The Call Manager will support devices with single (VoIP only) and multiple channels (VoIP, PSTN, Mobile). On multi-channel devices where there is pre-defined behavior implemented in the device firmware (such as Savi Office), the Call Manager will

manage only the VoIP channel. On multi-channel devices where there is no pre-defined behavior, the Call Manager will manage all the channels.

The Call Manager will also manage instances of active and held calls. There will be only one active call that can go through transitions based on the user action on the device or on the softphone. The Call Manager will be able to support multiple calls from the same softphone or from different softphones. A well-defined behavior for talk and flash button presses is implemented in the Call Manager. The Call Manager will also handle outgoing calls initiated from the device. Devices with keypad, contact list or call log can make call requests to the Device Manager. The Device Listener will get the call request and propagate it to the Call Manager. The Call Manager will be responsible for creating an outgoing call instance and manage it as per the behavior.

In the following method descriptions, *pCall* is the pointer to *ICall**. Parameter *callID* is the Call ID for the incoming call. Parameter *contact* is the contact information for the call. Parameter *tones* is the ring tone to use for the incoming call. Parameter *route* is the audio route to the headset or speaker.

In the following method descriptions, methods return CM_RESULT_SUCCESS for successfully notifying Spokes about the incoming call and other *CMResult* codes for failure.

```
INTF ICallCommand : public IDisposable
    {
    virtual CMResult incomingCall(pCall callID, pContact contact,
        eRingTone tones, eAudioRoute route)= 0;
    virtual CMResult outgoingCall(pCall callID, pContact contact,
        eAudioRoute route)=0;
    virtual CMResult terminateCall(pCall callID)=0;
    virtual CMResult answeredCall(pCall callID)=0;
    virtual CMResult holdCall(pCall callID)=0;
    virtual CMResult resumeCall(pCall callID)=0;
    virtual CMResult insertCall(pCall callID, pContact
        contact)=0;
    virtual CMResult muteCall(bool bMute)=0;
    virtual CMResult setAudioRoute(pCall callID, eAudioRoute
        route)=0;
    virtual CMResult makeCall(pContact contact)=0;
    virtual CMResult setConferenceId(pCall callID)=0;
    }
```

With *CMResult* return code:

### 2.2.1.1: CMResult enum

```
CM_RESULT_SUCCESS = 0
CM_RESULT_FAIL = 1
CM_RESULT_INVALID_ARGUMENT = 2
CM_RESULT_INVALID_STATE = 3
CM_RESULT_NO_DEVICE = 4
CM_RESULT_A2DP_DEVICE = 5
CM_RESULT_INVALID_LINETYPE = 6
CM_RESULT_NOT_SUPPORTED = 7
```

#### 2.2.1.2: eRingTone enum

```
RING_TONE_UNKNOWN = 0
```

### *2.2.2: CMResult incomingCall(pCall callID, pContact contact, eRingTone tones, eAudioRoute route)*

Spokes plug-in notifies Spokes about an incoming softphone call.

Parameter *callID* is the Call ID for the incoming call. Parameter *contact* is the contact information for the call. Parameter *tones* is the ring tone to use for the incoming call. Parameter *route* is the audio route to the headset or speaker.

Returns CM_RESULT_SUCCESS for successfully notifying Spokes about the incoming call and other *CMResult* codes for failure.

#### 2.2.2.1: eAudioRoute enum

```
AUDIO_ROUTE_TO_HEADSET = 0
AUDIO_ROUTE_TO_SPEAKER = 1
```

#### 2.2.2.2: eRingTone enum

```
RING_TONE_UNKNOWN = 0
```

### *2.2.3: CMResult outgoingCall(pCall callID, pContact contact, eAudioRoute route)*

Spokes plug-in notifies Spokes about an outgoing softphone call.

Parameter *callID* is the Call ID for the outgoing call. Parameter *contact* is the contact information for the call. Parameter *route* is the audio route to the headset or speaker.

Returns CM_RESULT_SUCCESS for successfully notifying Spokes about the incoming call and other *CMResult* codes for failure.

See Also: Section 2.2.2.1:eAudioRoute enum.

### *2.2.4: CMResult terminateCall(pCall callID)*

Spokes plug-in notifies Spokes about a call terminated in the GUI.

### *2.2.5: CMResult answeredCall(pCall callID)*

Spokes plug-in notifies Spokes about a call that is answered through the GUI, for example, in the softphone.

### *2.2.6: CMResult holdCall(pCall callID)*

Spokes plug-in notifies Spokes about a softphone call put on hold through the softphone GUI.

### *2.2.7: CMResult resumeCall(pCall callID)*

Spokes plug-in notifies Spokes about a softphone call resumed (after being put on hold) through the softphone GUI.

### *2.2.8: CMResult muteCall(bool bMute)*

Spokes plug-in notifies Spokes about a softphone call muted or unmuted through the softphone GUI.

The parameter *bMute* is true if muted, false if not muted.

### *2.2.9: CMResult insertCall(pCall callID, pContact contact)*

Spokes plug-in notifies Spokes about a softphone call that is already active by calling *insertCall()*. Some softphones provide notification about already active calls as part of their SDK so this is applicable only for softphones that support the same.

### *2.2.10: CMResult setAudioRoute(pCall callID, eAudioRoute route)*

Spokes plug-in sets the AudioRoute by calling *setAudioRoute()*. The audio is routed either to the Plantronics Headset or to the computer system Speaker phone. The Call Manager uses the audio route to decide if it needs to bring the RF link up for wireless headsets.

See Also: Section 2.2.2.1:eAudioRoute enum.

### *2.2.11: CMResult setConferenceId(pCall callID)*

Use this method to set the conference ID of the conference to which this call belongs, if any. Needed only for Softphone integration and Softphones that support conference calls.

### *2.2.12: CMResult makeCall(pContact contact)*

Plug-ins can call this API to make a call to the given contact through the Default Softphone. The Default Softphone should be one that is capable of making outbound calls given a contact phone number.

## 2.3: ICallEvents Interface

The *ICallEvents* interface provides methods to receive information when a call state changes due to a device action or when an outgoing call request is made from a Device (for example, Capri).

```
INTF ICallEvents{
    virtual bool OnCallStateChanged(CallStateEventArgs const&
        pcscArgs) = 0;
    virtual bool OnCallRequest(CallRequestEventArgs const&
        pcscArgs) = 0;
    };
```

### *2.3.1: bool OnCallStateChanged(CallStateEventArgs const& pcscArgs)*

The Call Manager triggers an *OnCallStateChanged* event when the call managed by Spokes changes state. In this event, a Spokes plug-in manages softphone calls. Alternatively, a Spokes plug-in can register for *IDeviceListenerEvents* and receive raw button presses from the device.

#### 2.3.1.1: CallStateEventArgs struct

```
struct CallStateEventArgs
    {
    enum {
        MAX_OPTIONS = 32,
        MAX_SOURCE = 128
        };
    eCallState callState;
    eDeviceEventKind deviceEventkind;
    int numOptions;
    EventArgsOption options[MAX_OPTIONS];
    CallData callId;
    wchar_t callSource[MAX_SOURCE];
    wchar_t callerId[MAX_SOURCE];
    wchar_t devicePath[MAX_DEVICE_PATH];

    };
```

#### 2.3.1.2: eDeviceEventKind enum

```
DEVICE_EVENT_KIND_DOCKED = 1
DEVICE_EVENT_KIND_UNDOCKED = 2
DEVICE_EVENT_KIND_TALKPRESS = 3
DEVICE_EVENT_KIND_UNKNOWN = 4
DEVICE_EVENT_KIND_NO_DEVICE = 5
DEVICE_EVENT_KIND_BASEPRESS = 6
DEVICE_EVENT_KIND_IDLE_TALKPRESS = 7
```

#### 2.3.1.3: EventArgsOption struct

```
struct EventArgsOption
{
    enum {
    MAX_OPTION = 64,
    MAX_VALUE = 64,
    };
    char option[MAX_OPTION];
    char value[MAX_VALUE];
};
```

#### 2.3.1.4: CallData struct

```
struct CallData
{
    int32_t id;
    int32_t conferenceID;
    bool inConference;
    void * callContext;
};
```

### *2.3.2: bool OnCallRequest(CallRequestEventArgs const& pcscArgs)*

The Call Manager triggers an *OnCallRequest* event when a call is requested by the device either from a dialing key handset pad, selecting a contact from the contact list or selecting a call log from the call history. This is an event that the plug-in can register to get notification from a Plantronics device. When there are multiple sessions active the Call Manager will use the User preferences and only notify the default softphone.

#### 2.3.2.1: CallRequestEventArgs struct

```
struct CallRequestEventArgs
{
    ContactData contact;
};
```

#### 2.3.2.2: ContactData Struct

```
struct ContactData
{
    enum {
    MAX_NAME = 128,
    MAX_URI = 256,
    MAX_PHONE = 64,
    MAX_EMAIL = 64
};

    wchar_t name[MAX_NAME];
    wchar_t friendlyName[MAX_NAME];
    int32_t id;
    wchar_t sipUri[MAX_URI];
    wchar_t phone[MAX_PHONE];
    wchar_t workPhone[MAX_PHONE];
    wchar_t mobilePhone[MAX_PHONE];
    wchar_t homePhone[MAX_PHONE];
    wchar_t email[MAX_EMAIL];
};
```

## 2.4: ICallInfo Interface

*ICallInfo* provides information about individual calls currently managed by the Call Manager.

```
INTF ICallInfo{
    virtual int32_t getCallId() = 0;
    virtual bool getSessionId(char * sessionid, int32_t len) = 0;
    virtual bool getSource(wchar_t * source, int32_t len) = 0;
    virtual bool isActive() = 0;
    };
```

### *2.4.1: int32_t getCallId()*

This method specifies the call ID associated with the call.

### *2.4.2: bool getSessionId(char * sessionid, int32_t len)*

This method specifies the session ID associated with the call. The parameter *len* is the size of the buffer.

### *2.4.3: bool getSource(wchar_t * source, int32_t len)*

This method specifies the source of the call (plug-in name) associated with the call. The parameter *len* is the size of the buffer.

### *2.4.4: bool isActive()*

This method specifies whether the call is active or on hold.

## 2.5: ICallInfoGroup Interface

Use the *ICallInfoGroup* interface to iterate over all the calls managed by the Call Manager.

```
INTF ICallInfoGroup{
    virtual ICallInfo * getCall(uint32_t index) = 0;
    virtual uint32_t numCalls() = 0;
    };
```

### *2.5.1: ICallInfo * getCall(uint32_t index)*

This method returns a pointer to the call at a given *index*. If there is no call in the group, this method returns `nullptr`.

### *2.5.2: uint32_t numCalls()*

This method returns the number of calls in the group.

## 2.6: ICallManagerState Interface

The *ICallManagerState* interface provides call state information about calls currently managed by the Call Manager.

```
INTF ICallManagerState : public IDisposable
    {
    virtual bool HasActiveCall() = 0;
    virtual int32_t  numberOfCallsOnHold() = 0;
    virtual bool getCalls(ICallInfoGroup** group) = 0;
    };
```

### *2.6.1: bool HasActiveCall()*

This method specifies if the Call Manager currently has an active call.

### *2.6.2: int32_t numberOfCallsOnHold()*

This method specifies the number of held calls in the Call Manager.

### *2.6.3: bool getCalls(ICallInfoGroup** group)*

This method provides a list of incoming, active, and held calls in the Call Manager.

## 2.7: IContact Interface

Use the *IContact* interface to provide information about a contact from, for example, an address book.

```
INTF IContact : public IDisposable
    {
    virtual const wchar_t* getName()=0;
    virtual void setName(const wchar_t* pName)=0;
    virtual const wchar_t* getFriendlyName()=0;
    virtual void setFriendlyName(const wchar_t* pFName)=0;
    virtual int32_t getID()=0;
    virtual void setID(int32_t id)=0;
    virtual const wchar_t* getSipUri()=0;
    virtual void setSipUri(const wchar_t* pSip)=0;
    virtual const wchar_t* getPhone()=0;
    virtual void setPhone(const wchar_t* pPhone)=0;
    virtual const wchar_t* getEmail()=0;
    virtual void setEmail(const wchar_t* pEmail)=0;
    virtual const wchar_t* getWorkPhone()=0;
    virtual void setWorkPhone(const wchar_t* pWPhone)=0;
    virtual const wchar_t* getMobilePhone()=0;
    virtual void setMobilePhone(const wchar_t* pMPhone)=0;
    virtual const wchar_t* getHomePhone()=0;
    virtual void setHomePhone(const wchar_t* pHPhone)=0;
    };
```

### *2.7.1: const wchar_t* getName()*

This method allows reading a contact's name.

### *2.7.2: void setName(const wchar_t* pName)*

This method allows setting a contact's name.

### *2.7.3: const wchar_t* getFriendlyName()*

This method gets the friendly name of the contact, which is something a human can easily read and remember instead of a code or a numeric address/identifier.

### *2.7.4: void setFriendlyName(const wchar_t* pFName)*

This method sets the friendly name of the contact, which is something a human can easily read and remember instead of a code or a numeric address/identifier.

### *2.7.5: int32_t getID()*

This method fetches an integer that uniquely identifies a contact.

### *2.7.6: void setID(int32_t id)*

This method allows setting an integer that uniquely identifies a contact.

### *2.7.7: const wchar_t* getSipUri()*

This method allows reading a contact's SIP Uniform Resource Identifier (URI), which is the Session Initiation Protocol (SIP) addressing schema to call another person via SIP. In other words, a SIP URI is a user's SIP phone number. The SIP URI resembles an e-mail address.

### *2.7.8: void setSipUri(const wchar_t* pSip)*

This method allows setting a contact's SIP Uniform Resource Identifier (URI), which is the Session Initiation Protocol (SIP) addressing schema to call another person via SIP. In other words, a SIP URI is a user's SIP phone number. The SIP URI resembles an e-mail address.

### *2.7.9: const wchar_t* getPhone()*

This method allows reading a contact's phone number.

### *2.7.10: void setPhone(const wchar_t* pPhone)*

This method allows setting a contact's phone number.

### *2.7.11: const wchar_t* getEmail()*

This method allows reading a contact's email address.

### *2.7.12: void setEmail(const wchar_t* pEmail)*

This method allows setting a contact's email address.

### *2.7.13: const wchar_t* getWorkPhone()*

This method allows reading a contact's work phone number.

### *2.7.14: void setWorkPhone(const wchar_t* pWPhone)*

This method allows setting a contact's work phone number.

### *2.7.15: const wchar_t* getMobilePhone()*

This method allows reading a contact's mobile phone number.

### *2.7.16: void setMobilePhone(const wchar_t* pMPhone)*

This method allows setting a contact's mobile phone number.

### *2.7.17: const wchar_t* getHomePhone()*

This method allows reading a contact's home phone number.

### *2.7.18: void setHomePhone(const wchar_t* pHPhone)*

This method allows setting a contact's home phone number.

## 2.8: ISession Interface

Inherit the *ISession* interface to get information about the current session.

```
INTF ISession : public IDisposable{
    virtual SMResult getSessionId(wchar_t* id, uint32_t len) = 0
    virtual SMResult getPluginId(int32_t* pluginId) = 0;
    virtual SMResult getPluginName(wchar_t* pName, int32_t len) = 0;
    virtual SMResult registerCallback(pCallEvents callback) = 0;
    virtual SMResult unregisterCallback(pCallEvents callback) = 0;
    virtual SMResult getCallCommand(ppCallCommand ppCommand) = 0;
    virtual SMResult getCallEvents(ppCallEvents ppEvents) = 0;
    virtual SMResult getActiveDevice(ppDevice ppDev)= 0;
    virtual SMResult getActiveDeviceName(wchar_t* pDeviceName,
        int32_t len) = 0;
    virtual SMResult enableAudio(bool bEnable) = 0;
```

With *SMResult* return code:

#### 2.8.1.1: SMResult enum

```
SM_RESULT_SUCCESS = 0
SM_RESULT_FAIL = 1
SM_RESULT_INVALID_ARGUMENT = 2
SM_RESULT_ALREADY_REGISTERED = 3
SM_RESULT_NOT_FOUND = 4
SM_RESULT_NOT_SUPPORTED = 5
SM_RESULT_NOT_INITIALIZED = 6
```

### *2.8.2: SMResult getSessionId(wchar_t* id, uint32_t len)*

This method returns a GUID that uniquely identifies a Spokes session. The parameter *len* is the size of the buffer.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.8.3: SMResult getPluginId(int32_t* pluginId)*

This method returns a string that uniquely identifies a Spokes plug-in by its identification (ID) number.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.8.4: SMResult getPluginName(wchar_t* pName, int32_t len)*

This method returns a string that uniquely identifies a Spokes plug-in by its name. The parameter *len* is the size of the buffer.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.8.5: SMResult registerCallback(pCallEvents callback)*
### *SMResult unregisterCallback(pCallEvents callback)*

These methods register or unregister an *ICallEvents* interface for receiving events from Plantronics SDK.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.8.6: SMResult getCallCommand(ppCallCommand ppCommand)*

This method returns a *ppCallCommand* that Spokes plug-in uses to notify Spokes about incoming call and other softphone related call commands.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.8.7: SMResult enableAudio(bool bEnable)*

Use the *enableAudio()* method to request that Spokes enable audio (meaning, bring up the RF Link) for cordless headsets.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.8.8: SMResult getActiveDevice(ppDevice ppDev)*

This method returns a *ppDevice* that represents the Plantronics device to which Spokes is currently connected. Spokes plug-ins can use the *ppDevice* interface to send device specific commands to the device and register for device specific events.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.8.9: SMResult getActiveDeviceName(wchar_t* pDeviceName, int32_t len)*

This method returns a *pDeviceName* that allows it to send and retrieve contact information to supported Plantronics devices. The parameter *len* is the size of the buffer.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

## 2.9: ISessionManager Interface

The Spokes Session Manager is responsible for handling all incoming client sessions. The client applications can range from Softphone (SP) and Media Player (MP) modules to other third party applications that want to interface with devices

directly. The Spokes Session Manager will expose interfaces for call handling, device state and events (both as Plantronics devices and as supported ATDs (Alternative Telephony Device)).

The Session Manager provides the following functionality:

- Handles all incoming client requests
- Maintains a list of client sessions
- Interfaces with the Call Manager and the Device Listener and propagates events and commands from and to the Spokes clients.

```
INTF ISessionManager :  public IDisposable
{
    virtual SMResult registerSession(const wchar_t* pPluginName,
        ppSession ppSess) noexcept = 0;
    virtual SMResult unregisterSession(pSession pSess) noexcept = 0;
    virtual SMResult setPluginStatus(int32_t pluginId,
        ePluginState pluginState) noexcept = 0;
    virtual SMResult getActive(int32_t pluginId, bool& bActive) noexcept = 0;
    virtual SMResult setOption(eSessionMgrOption eDOption, int32_t
        lValue) noexcept = 0;
    virtual SMResult getOption(eSessionMgrOption eDOption,
        int32_t* plValue, int16_t iMaxLength, int16_t* piLength) noexcept = 0;
    virtual SMResult registerCallback(pSessionManagerEvents
        callback) noexcept = 0;
    virtual SMResult unregisterCallback(pSessionManagerEvents
        callback) noexcept = 0;
    virtual SMResult getUserPreference(IUserPreference** userPref) noexcept = 0;
    virtual SMResult getCallManagerState(ICallManagerState **
        callManagerState) noexcept = 0;
    virtual SMResult getActiveDevice(ppDevice ppDev)) noexcept = 0;
virtual SMResult getDeviceForPath(const wchar_t* devicePath, ppDevice ppDev)
        NOEXCEPT = 0;
};
```

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

#### 2.9.1.1: ePluginState enum

```
PLUGIN_RUNNING = 1
PLUGIN_ACTIVE = 2
PLUGIN_USER_SIGNIN = 3
PLUGIN_USER_SIGNOUT = 4
PLUGIN_NOT_ACTIVE = 5
```

### *2.9.2: SMResult registerSession(const wchar_t* pPluginName, ppSession ppSess)*

Registers and returns a session object for a given plug-in name. Spokes plug- ins register with the Spokes Session Manager by calling registerSession() once loaded by Spokes. The registerSession() method returns a virtual int32_t object that the plug-in will use to interface with Spokes.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.9.3: SMResult unregisterSession(pSession pSess)*

When exiting, Spokes plug-ins should call *unregisterSession()* to unregister a session from the Session Manager.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.9.4: SMResult setPluginStatus(int32_t pluginId, ePluginState pluginState)*

This method sets the plug-in status to active. Once a plug-in has done all its initialization, it calls *setActive(),* which notifies Spokes that it is running.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.9.5: SMResult getActive(int32_t pluginId, bool& bActive)*

Gets the plug-in active status. Once a plug-in has done all its initialization, it can call *getActive()* to get a list of active plug-ins.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.9.6: SMResult setOption(eSessionMgrOption eDOption, int32_t lValue)*

This method sets an option to a given value.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum. *eSessionMgrOption* enum values are listed below.

#### 2.9.6.1: eSessionMgrOption enum

The options for the `eSessionMgrOption` enum are:

```
SESSIONMANAGER_OPT_UNKNOWN = 0
SESSIONMANAGER_SUPPRESS_DIALTONE = 1
SESSIONMANAGER_LOAD_PLUGINS = 2
```

### *2.9.7: SMResult getOption(eSessionMgrOption eDOption, int32_t* plValue, int16_t iMaxLength, int16_t* piLength)*

Gets the value for a given option.

See Also: See Also: *SMResult* enum values are listed on page **Error! Bookmark not defined.**. Options for *eSessionMgrOption* enum values are shown above.

### *2.9.8: SMResult registerCallback(pSessionManagerEvents callback) SMResult unregisterCallback(pSessionManagerEvents callback)*

Registers or unregisters a callback for ISessionManagerEvents CallStateChanged and DeviceStateChanged.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.9.9: SMResult * getUserPreference (IUserPreference** userPref)*

This method gets the user preference and returns the current choices made by users. User preferences can be used by various plug-ins.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.9.10: SMResult getCallManagerState(ICallManagerState **callManagerState)*

The *getCallManagerState* method returns the current state of the Call Manager. It provides a list of active and held calls.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.9.11: SMResult getActiveDevice(ppDevice ppDev) noexcept*

Gets the current active device. The parameter *ppDev* is an output pointer to receive the active device.

See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

### *2.9.12: SMResult getDeviceForPath(const wchar_t* devicePath, ppDevice ppDev)*

Given a path, this method gets the device on that path. The parameter *devicePath* is the path to the device. The parameter *ppDev* is an output pointer to receive a device for the path provided. This method returns SM_RESULT_SUCCESS for success or other codes for failure.

## 2.10: ISessionManagerEvents Interface

Inherit the *ISessionManagerEvents* interface to be notified of state changes on the device or the call.

```
INTF ISessionManagerEvents{
    virtual bool OnCallStateChanged(CallStateEventArgs const&
        cseArgs) = 0;
    virtual bool OnDeviceStateChanged(DeviceStateEventArgs const&
        devArgs) = 0;
    };
```

### *2.10.1: bool OnCallStateChanged(CallStateEventArgs const& cseArgs)*

Notifies on call state changes, and returns success status. An onCallStateChanged event is triggered by the Session Manager when one of the Spokes plug-ins has a ringing or active call or when an active call is ended or terminated. This is the event that plug-ins register to get notification about call status across other plug-ins.

See Also: Section 2.3.1.1: CallStateEventArgs struct.

### *2.10.2: bool OnDeviceStateChanged(DeviceStateEventArgs const& devArgs)*

Notifies on device state changes like device arrival and removal, and returns success status. Notifies the plug-in about device removal and arrival. Once a Plantronics device is plugged in, the Session Manager notifies the plug-in about the arrival of the device. The *getActiveDevice()* method provides a reference to the current device to which the Call Manager is connected.

#### 2.10.2.1: DeviceStateEventArgs struct

```
enum {
     MAX_DEVPATH     = 128
     };
    eDeviceState deviceState;
    wchar_t devicePath[MAX_DEVPATH];
```

## 2.11: IDeviceAttributes Interface

An object with this interface is owned by its Device object, which has the sole responsibility for managing its lifetime.

```
INTF IDeviceAttributes
{
    virtual DeviceCaps const* GetDeviceCaps() noexcept = 0;
    virtual ICapabilities const* GetInputCaps() noexcept = 0;
    virtual ICapabilities const* GetOutputCaps() noexcept = 0;
    virtual ICapabilities const* GetFeatureCaps() noexcept = 0;
    virtual int32_t GetDelayTime() noexcept = 0;
    virtual void SetDelayTime(int32_t value) noexcept = 0;
    virtual bool HasUsage(USAGE usage) const noexcept = 0;
    virtual bool HasUsage(USAGE usagePage, USAGE usage) const noexcept = 0;
    virtual bool HasBtnUsage(USAGE usage, ReportType enumReportType) const
        noexcept = 0;
    virtual bool HasValUsage(USAGE usage, ReportType enumReportType) const
        noexcept = 0;
    virtual bool HasBtnUsage(USAGE usagePage, USAGE usage, ReportType
        enumReportType) const noexcept = 0;
    virtual bool HasValUsage(USAGE usagePage, USAGE usage, ReportType
        enumReportType) const noexcept = 0;
    virtual IDeviceUsage* GetDeviceUsage(ReportType eReportType)noexcept  = 0;
    virtual bool IsConstant(USAGE usage) const noexcept = 0;
}
```

### *2.11.1: DeviceCaps const* GetDeviceCaps()*

Returns a non-owning pointer to the device capabilities object, whose lifetime is managed solely by this object. So, the user should not keep this pointer, or should not use it after this object (actually Device object of this object) is destroyed.

### *2.11.2: ICapabilities const* GetInputCaps()*

Returns a non-owning pointer to the input capabilities object, whose lifetime is managed solely by this object. So, the user should not keep this pointer, or should not use it after this object (actually Device object of this object) is destroyed.

### *2.11.3: ICapabilities const* GetOutputCaps()*

Returns a non-owning pointer to the output capabilities object, whose lifetime is managed solely by this object. So, the user should not keep this pointer, or should not use it after this object (actually Device object of this object) is destroyed.

### *2.11.4: ICapabilities const* GetFeatureCaps()*

Returns a non-owning pointer to the feature object, whose lifetime is managed solely by this object. So, the user should not keep this pointer, or should not use it after this object (actually Device object of this object) is destroyed.

### *2.11.5: int32_t GetDelayTime()*

Returns the delay time, in milliseconds

### *2.11.6: void SetDelayTime(int32_t value)*

Sets the delay time, in milliseconds

### *2.11.7: bool HasUsage(USAGE usage)*

Returns true if the given usage (on the default usage page) exists, either as a button or as a value usage, on any report type.

### *2.11.8: bool HasUsage(USAGE usagePage, USAGE usage)*

Returns true if the given usage on the given usage page exists, either as a button or as a value usage, on any report type.

### *2.11.9: bool HasBtnUsage(USAGE usage, ReportType enumReportType)*

Returns true if the given usage (on the default usage page) for the given report type exists as a button usage.

### *2.11.10: bool HasValUsage(USAGE usage, ReportType enumReportType)*

Returns true if the given usage (on the default usage page) for the given report type exists as a value usage.

### *2.11.11: bool HasBtnUsage(USAGE usagePage, USAGE usage, ReportType enumReportType)*

Returns true if the given usage on the given usage page for the given report type exists as a button usage.

### *2.11.12: bool HasValUsage(USAGE usagePage, USAGE usage, ReportType enumReportType)*

Returns true if the given usage on the given usage page for the given report type exists as a value usage.

### *2.11.13: IDeviceUsage* GetDeviceUsage(ReportType eReportType)*

Creates and returns an owning pointer to the device usage object for the given report type. The user should call *Release()* on the object on end-of-life.

### *2.11.14: bool IsConstant(USAGE usage)*

Returns true if the given usage is constant (as per the report descriptor)

## 2.12: IDeviceUsage Class

Holds an array of usages of interest to the user, provides functions to work with those usages, but also with all usages of the device. This includes reading and writing reports.

Also holds an internal buffer for report(s).

```
class IDeviceUsage : public IDisposable
{
public:
    virtual void AddUsage(const Usage& usage) = 0;
    virtual void AddUsage(uint16_t usagePage, uint16_t usage) = 0;
    virtual void AddUsage(uint16_t usagePage, uint16_t usage, unsigned long
        value) = 0;
    virtual Usage const* FindUsage(const USAGE usage) noexcept = 0;
    virtual DMResult SetUsage() noexcept = 0;
    virtual DMResult SetOutputUsage(const Usage& usage) noexcept = 0;
    virtual DMResult SetOutputUsage(uint16_t usagePage, uint16_t usage)
        noexcept = 0;
    virtual DMResult SetOutputUsage(uint16_t usagePage, uint16_t usage, unsigned
        long value) noexcept = 0;
    virtual DMResult SetOutputUsage(uint8_t reportID, uint16_t usagePage,
        uint16_t usage, unsigned long value) noexcept = 0;
    virtual DMResult SetValue(int32_t val) noexcept = 0;
    virtual DMResult SetValue(const Usage& usage) noexcept = 0;
    virtual DMResult SetOutputValue(const Usage& usage) noexcept = 0;
    virtual DMResult SetOutputValue(uint16_t usagePage, uint16_t usage)
        noexcept = 0;
    virtual DMResult SetOutputValue(uint16_t usagePage, uint16_t usage, unsigned
        long value) noexcept = 0;
    virtual DMResult GetReportID(uint16_t i_usagePage, uint16_t i_usage, uint8_t
        &o_reportID) noexcept = 0;
    virtual DMResult GetReportID(uint16_t i_usage, uint8_t &o_reportID)
        noexcept = 0;
    virtual int32_t GetMaxUsageListLength(USAGE usagePage) noexcept = 0;
    virtual DMResult GetUsages(USAGE usagePage, USAGE usages[], uint32_t&
        maxUsageListLen) noexcept = 0;
    virtual DMResult GetUsages(USAGE usagePage, USAGE usages[], uint32_t&
        maxUsageListLen, uint8_t const * pReportBuffer, int32_t
        reportBufferSize) noexcept = 0;
    virtual DMResult GetUsage() noexcept = 0;
    virtual uint32_t GetValue() noexcept = 0;
    virtual DMResult GetValue(USAGE usagePage, USAGE usage, uint32_t &data)
        noexcept = 0;
    virtual DMResult GetValue(USAGE usagePage, USAGE usage, uint8_t* buffer,
        int32_t bufferSize, uint32_t& data) noexcept = 0;
    virtual DMResult GetValueArray(uint8_t* values, uint16_t len) noexcept = 0;
    virtual DMResult GetValueArray(uint16_t usagePage, USAGE usage, uint8_t*
```

```
        values, uint16_t len) noexcept = 0;
    virtual uint32_t GetValueArraySize() const noexcept = 0;
    virtual uint32_t GetValueArraySize(uint16_t usagePage, USAGE usage) const
        noexcept = 0;
    virtual DMResult GetData(USAGE usagePage, USAGE usage, uint32_t& data)
        noexcept = 0;
    virtual DMResult GetData(USAGE usagePage, USAGE usage, uint8_t const *
        buffer, int32_t bufferSize, uint32_t& data) noexcept = 0;
    virtual DMResult GetMakeList(pUSAGE previousUsageList, pUSAGE
        currentUsageList, pUSAGE makeList, int32_t usageListLen) noexcept = 0;
    virtual DMResult GetBreakList(pUSAGE previousUsageList, pUSAGE
        currentUsageList, pUSAGE breakList, int32_t usageListLen) noexcept = 0;
    virtual DMResult GetMakeBreakList(pUSAGE previousUsageList, pUSAGE
        currentUsageList, pUSAGE makeList, pUSAGE breakList, uint32_t
        usageListLen) noexcept = 0;
    virtual DMResult WriteReport() = 0;
    virtual DMResult WriteReport(uint8_t const *buffer, int32_t len)
        noexcept = 0;
    virtual DMResult WriteReportRaw(uint8_t const *buffer, int32_t len)
        noexcept = 0;
    virtual DMResult WriteReportCT(uint8_t const *buffer, int32_t len)
        noexcept = 0;
    virtual DMResult WriteReportWithDelay() noexcept = 0;
    virtual DMResult GetReport(uint8_t** reportBuffer, uint32_t *length)
        noexcept = 0;
    virtual DMResult ReadFeature(uint8_t* buffer, int32_t len) noexcept = 0;
    virtual DMResult ReadFeature() noexcept = 0;
    virtual DMResult ReadInput(uint8_t* buffer, int32_t len) noexcept = 0;
    virtual DMResult ReadInput() noexcept = 0; /** Device usage interface.
}
```

### *2.12.1: void AddUsage(const Usage& usage)*

Adds a usage to the internal array of usages.

### *2.12.2: void AddUsage(uint16_t usagePage, uint16_t usage)*

Adds a usage to the internal array of usages with the default value.

### *2.12.3: void AddUsage(uint16_t usagePage, uint16_t usage, unsigned long value)*

Adds a usage to the internal array of usages, with the given value.

### *2.12.4: Usage const* FindUsage(const USAGE usage)*

Returns a non-owning pointer to the found usage. The usage is looked for in the internal array.

If the usage is not found, returns `nullptr`.

### *2.12.5: DMResult SetUsage()*

Set the values of all the usages from the internal array to the internal report buffer. Initializes the internal report buffer if it is not initialized.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.6: DMResult SetOutputUsage(const Usage& usage)*

Set the value of the given usage to the internal report buffer, bypassing the internal usage array. Initializes the internal report buffer if it is not initialized, as per the first element of the usage array.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.7: DMResult SetOutputUsage(uint16_t usagePage, uint16_t usage)*

Set the given usage to the internal report buffer, bypassing the internal usage array, using the default value. Initializes the internal report buffer if it is not initialized, as per the first element of the usage array.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.8: DMResult SetOutputUsage(uint16_t usagePage, uint16_t usage, unsigned long value)*

Set the given usage to the internal report buffer, bypassing the internal usage array, using the given value. Initializes the internal report buffer if it is not initialized, as per the first element of the usage array.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.9: DMResult SetOutputUsage(uint8_t reportID, uint16_t usagePage, uint16_t usage, unsigned long value)*

Set the given usage to the internal report buffer, bypassing the internal usage array, using the given value. Initializes the internal report buffer if it is not initialized, as per the given reportID.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.10: DMResult SetValue(int32_t val)*

Sets the first usage in the internal usage array to the internal report buffer, using the given value. Bypasses the value of the usage in the internal array. Assumes the internal report buffer is initialized.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.11: DMResult SetValue(const Usage& usage)*

Sets the given usage in the internal usage array to the internal report buffer, using the given value. Bypasses the internal usage array. Assumes the internal report buffer is initialized.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.12: DMResult SetOutputValue(const Usage& usage)*

Sets the given usage in the internal usage array to the internal report buffer, using the given value. Bypasses the internal usage array. Initializes the internal report buffer (if not already initialized).

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.13: DMResult SetOutputValue(uint16_t usagePage, uint16_t usage)*

Sets the given usage in the internal usage array to the internal report buffer, using the default value. Bypasses the internal usage array. Initializes the internal report buffer (if not already initialized).

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.14: DMResult SetOutputValue(uint16_t usagePage, uint16_t usage, unsigned long value)*

Sets the given usage in the internal usage array to the internal report buffer, using the given value. Bypasses the internal usage array. Initializes the internal report buffer (if not already initialized).

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.15: DMResult GetReportID(uint16_t i_usagePage, uint16_t i_usage, uint8_t &o_reportID)*

Returns the report ID which contains the given usage (and usage page).

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.16: DMResult GetReportID(uint16_t i_usage, uint8_t &o_reportID)*

Returns the report ID which contains the given usage for default PLT usage page.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.17: int32_t GetMaxUsageListLength(USAGE usagePage)*

Returns the maximum length of the usage list array (to be used with GetUsages functions).

### *2.12.18: DMResult GetUsages(USAGE usagePage, USAGE usages[], uint32_t& maxUsageListLen)*

Reads the active usages for the given usage page from the internal report buffer. Bypasses the internal usage array.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.19: DMResult GetUsages(USAGE usagePage, USAGE usages[], uint32_t& maxUsageListLen, uint8_t const * pReportBuffer, int32_t reportBufferSize)*

Reads the active usages for the given usage page from the given report buffer. Bypasses the internal usage array.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.20: DMResult GetUsage()*

Reads the active usages from the internal report buffer and copies the values to the internal usage array.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.21: uint32_t GetValue()*

Returns the usage value of the first usage in the internal usage array from the internal report buffer. Bypasses the internal usage array.

### *2.12.22: DMResult GetValue(USAGE usagePage, USAGE usage, uint32_t &data)*

Gives the usage value of the given usage from the internal report buffer. Bypasses the internal usage array.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.23: DMResult GetValue(USAGE usagePage, USAGE usage, uint8_t* buffer, int32_t bufferSize, uint32_t& data)*

Gives the usage value of the given usage from the given report buffer. Bypasses the internal usage array.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.24: DMResult GetValueArray(uint8_t* values, uint16_t len)*

Gives the value array of the first usage in the internal usage array from the internal report buffer. The value array is read to the given buffer with the given length. Length can be calculated by calling *GetValueArraySize()* function.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.25: DMResult GetValueArray(uint16_t usagePage, USAGE usage, uint8_t* values, uint16_t len)*

Gives the value array of the given usage from the internal report buffer. The value array is read to the given buffer with the given length. Length can be calculated by calling *GetValueArraySize()* function.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.26: uint32_t GetValueArraySize()*

Gives the required size (dimension) of the value array of the first usage in the internal usage array.

### *2.12.27: uint32_t GetValueArraySize(uint16_t usagePage, USAGE usage)*

Gives the required size (dimension) of the value array of the given usage in the internal usage array.

### *2.12.28: DMResult GetData(USAGE usagePage, USAGE usage, uint32_t& data)*

Gives the usage data value of the given usage from the internal report buffer. The data index is read from device attributes (capabilities).

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.29: DMResult GetData(USAGE usagePage, USAGE usage, uint8_t const * buffer, int32_t bufferSize, uint32_t& data)*

Gives the usage data value of the given usage from the given report buffer. The data index is read from device attributes (capabilities).

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.30: DMResult GetMakeList(pUSAGE previousUsageList, pUSAGE currentUsageList, pUSAGE makeList, int32_t usageListLen)*

Given the array of usages that were ON in a previous read, and those from a current read, gives a list of the usages that are in the current and weren't in the previous ("make" list/array). It is assumed that all arrays are of the same dimension, and that the elements of the arrays that are not populated with usage ID have the value 0. The elements of the make list which are not populated with usage ID will have the value set to 0.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.31: DMResult GetBreakList(pUSAGE previousUsageList, pUSAGE currentUsageList, pUSAGE breakList, int32_t usageListLen)*

Given the array of usages that were ON in a previous read, and those from a current read, gives a list of the usages that are were the previous and aren't in the current. It is assumed that all arrays are of the same dimension, and that the elements of the arrays that are not populated with usage ID have the value 0. The elements of the break list which are not populated with usage ID will have the value set to 0.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.32: DMResult GetMakeBreakList(pUSAGE previousUsageList, pUSAGE currentUsageList, pUSAGE makeList, pUSAGE breakList, uint32_t usageListLen)*

Given the array of usages that were ON in a previous read, and those from a current read, gives two lists (arrays):

- a list of the usages that are in the current and weren't in the previous (make list)
- a list of the usages that are were the previous and aren't in the current (break list)

It is assumed that all arrays are of the same dimension, and that the elements of the arrays that are not populated with usage ID have the value 0. The elements of the break and make lists which are not populated with usage ID will have the value set to 0.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.33: DMResult WriteReport()*

Writes the report in the internal report buffer to the device.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.34: DMResult WriteReport(uint8_t const *buffer, int32_t len)*

Writes the given report buffer to the device, using default length (*len*). Excess bytes are filled with zeros.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.35: DMResult WriteReportRaw(uint8_t const *buffer, int32_t len)*

Writes the given report buffer to the device with the given length (*len*).

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.36: DMResult WriteReportCT(uint8_t const *buffer, int32_t len)*

Writes the given report buffer to the device with the given length (*len*) using alternate method.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.37: DMResult WriteReportWithDelay()*

Writes the report in the internal report buffer to the device, with a delay.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.38: DMResult GetReport(uint8_t** reportBuffer, uint32_t *length)*

Gives a non-owning pointer to the array of chars which holds the report buffer, and also the dimension of the array (number of elements).

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.39: DMResult ReadFeature(uint8_t* buffer, int32_t len)*

Reads a feature report from the device and puts it in the provided buffer. This buffer should be obtained by calling *GetReport()* member function or properly initialized by some other means. The parameter *len* is the size of the buffer.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.40: DMResult ReadFeature()*

Reads a feature report from the device and puts it in the internal report buffer.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.41: DMResult ReadInput(uint8_t* buffer, int32_t len)*

Reads an input report from the device and puts it in the provided buffer. This buffer should be obtained by calling *GetReport()* member function or properly initialized by some other means. The parameter *len* is the size of the buffer.

See Also: Section 4.1.1.1: DMResult enum.

### *2.12.42: DMResult ReadInput()*

Reads an input report from the device and puts it in the internal report buffer.

See Also: Section 4.1.1.1: DMResult enum.

# Chapter 3: Using Device Manager Interfaces

The Device Manager provides device-level interfaces to events, commands, and properties, and handles the interaction between devices and applications. Through this interface, applications can detect actions taken on the device, such as button presses, and also monitor status information (for example, proximity signals and battery capacity). The Device Manager also enables applications to send commands to devices, such as ringing a headset.

The Device Manager interfaces with a human interface device (HID) layer to communicate with USB devices. The HID layer provides USB device communication and management by calling Win32 APIs using Platform Invoke services (Pinvoke) and is exposed only to the Device Manager.

Plantronics devices may provide more functionality (such as support for multiple communication endpoints VoIP, PSTN, Mobile) and these devices may require device-specific commands and events. The Device Manager exposes all of the device-specific events and commands supported by these devices.

```
INTF IDeviceManager : public IDisposable
{
    virtual DMResult getDevice(eVendorID vendorID, eProductID productID,
        ppDevice ppDev) noexcept = 0;
    virtual DMResult getDevice(eVendorID vendorID, ppDevice ppDev)
        noexcept = 0;
    virtual DMResult getDevice(const wchar_t* devicePath, ppDevice ppDev)
        noexcept = 0;
    virtual DMResult getDevices(IDeviceGroup** devices) noexcept = 0;
    virtual DMResult getHIDDevices(uint16_t i_usagePage, IDeviceGroup** devices)
        noexcept = 0;
    virtual DMResult getDevices(eVendorID vendorID, IDeviceGroup** devices)
        noexcept = 0;
    virtual DMResult getDevices(eVendorID vendorID, eProductID productID,
        IDeviceGroup** devices) noexcept = 0;
    virtual DMResult setOption(eDMOption eDOption, int32_t lValue) noexcept = 0;
    virtual DMResult getOption(eDMOption eDOption, int32_t* plValue, int16_t
        iMaxLength, int16_t* piLength) noexcept = 0;
    virtual DMResult registerCallback(pDeviceManagerCallback pDMCallbacks)
        noexcept = 0;
    virtual DMResult unregisterCallback(pDeviceManagerCallback pDMCallbacks)
        noexcept = 0;
    virtual DMResult registerCallback(IGenericDeviceCallback *pDMCallbacks) NOEXCEPT = 0;
    virtual DMResult unregisterCallback(IGenericDeviceCallback *pDMCallbacks)
        NOEXCEPT = 0;
    virtual DMResult setDeviceFilterEnabled(bool) noexcept = 0;
    virtual bool getDeviceFilterEnabled() noexcept = 0;
    virtual IDeviceBaseExt * getDeviceExtension(IDevice* pDev) NOEXCEPT = 0;
```

## 3.1: IDeviceManagerCallback

Inherit this interface to receive *DeviceManager* callbacks. Use it in conjunction with *IDeviceManager::registerCallback*. When finished, use *IDeviceManager::unregisterCallback*.

```
INTF IDeviceManagerCallback
{
    virtual void onDeviceAdded(IDevice *pDev, wchar_t const *path) = 0;
    virtual void onDeviceRemoved(wchar_t const *path) = 0;
    virtual void onSystemSuspending() = 0;
    virtual void onSystemResuming() = 0;
```

### *3.1.1: void onDeviceAdded(IDevice *pDev, wchar_t const *path)*

This method is called when a device is added.

The parameter *pDev* is a pointer to the *IDevice* interface of the device instance created for the added device. The parameter *path* is the path to the added device.

### *3.1.2: void onDeviceRemoved(wchar_t const *path)*

This method is called when a device is removed from a system.

The parameter *path* is the path to the removed device.

### *3.1.3: void onSystemSuspending()*

This method is called before the system goes to sleep.

### *3.1.4: void onSystemResuming()*

This method is called after the system returns from sleep.

Returns whether a device in DFU mode is connected.

## 3.2: IDevice Interface

The *IDevice* interface exposes common device behavior.

```
INTF IDevice : public IDisposable
{
    virtual int32_t getProductID() const noexcept = 0;
    virtual int32_t getVendorID() const noexcept = 0;
    virtual int32_t getVersionNumber() const noexcept = 0;
    virtual DMResult getDevicePath(wchar_t* path, int32_t len) const
        noexcept = 0;
    virtual DMResult getInternalName(wchar_t* name, int32_t len) const
        noexcept = 0;
    virtual DMResult getProductName(wchar_t* name, int32_t len) const
        noexcept = 0;
    virtual DMResult getManufacturerName(wchar_t* name, int32_t len) const
        noexcept = 0;
    virtual DMResult getSerialNumber(wchar_t* serialNumber, int32_t len) const
        noexcept = 0;
    virtual bool isAttached() const noexcept = 0;
```

```
virtual bool attach() noexcept = 0;
virtual bool detach() noexcept = 0;
virtual bool isSupported(uint16_t usage) const noexcept = 0;
virtual DMResult getDeviceAttributes(IDeviceAttributes **ppAttributes) const
    noexcept = 0;
virtual DMResult getHostCommand(ppHostCommand ppCommand) const noexcept = 0;
virtual DMResult getDeviceEvents(ppDeviceEvents ppEvents) const
    noexcept = 0;
virtual DMResult getDeviceListener(ppDeviceListener ppListener) const
    noexcept = 0;
virtual DMResult setOption(eDeviceOption eDOption, int32_t lValue)
    noexcept = 0;
virtual DMResult getOption(eDeviceOption eDOption, int32_t* plValue, int16_t
    iMaxLength, int16_t* piLength) noexcept = 0;
virtual DMResult registerCallback(pDeviceCallback pDevCallbacks)
    noexcept = 0;
virtual DMResult unregisterCallback(pDeviceCallback pDevCallbacks)
    noexcept = 0;
virtual DMResult getDeviceType(uint32_t *io_cntType, DeviceType *o_pType)
    NOEXCEPT = 0;
virtual IDeviceManager *getDeviceManager() NOEXCEPT = 0;
```

#### 3.2.1.1: enum DeviceType

This is a classification of devices based on how it carries sound to the host computer. A device can be of more than one type. The defined types carry sound over Bluetooth, DECT, and/or wire/USB.

```
dtBluetooth
dtDECT
dtWireline
```

### *3.2.2: int32_t getProductID()*

Gets and returns USB Product ID assigned to the device by Plantronics.

### *3.2.3: int32_t getVendorID()*

Gets and returns the USB Vendor ID for Plantronics (0x47F).

### *3.2.4: int32_t getVersionNumber()*

Gets and returns the USB firmware version assigned to the device by Plantronics.

### *3.2.5: DMResult getDevicePath(wchar_t* path, int32_t len)*

Gets the Human Interface Device (HID) path of the device. This is the path the Device Manager uses to read/write to the device.

The parameter *path* is a pointer to the buffer that contains the device path. The parameter *len* is the size of the buffer.

Returns DM_RESULT_SUCCESS on success, DM_RESULT_BUFFER_TOO_SMALL if the buffer is too small.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.6: DMResult getInternalName(wchar_t* name, int32_t len)*

Gets the internal device name.

The parameter *name* points to the buffer that contains the internal name. The parameter *len* is the size of the buffer.

Returns DM_RESULT_SUCCESS on success, DM_RESULT_BUFFER_TOO_SMALL if the buffer is too small.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.7: DMResult getProductName(wchar_t* name, int32_t len)*

Gets name under which the device is marketed.

The parameter *name* points to the buffer that contains the product name. The parameter *len* is the size of the buffer.

Returns DM_RESULT_SUCCESS on success, DM_RESULT_BUFFER_TOO_SMALL if the buffer is too small.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.8: DMResult getManufacturerName(wchar_t* name, int32_t len)*

Gets manufacturer name (Plantronics Inc.)

The parameter *name* points to the buffer that contains the manufacturer name. The parameter *len* is the size of the buffer.

Returns DM_RESULT_SUCCESS on success, DM_RESULT_BUFFER_TOO_SMALL if the buffer is too small.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.9: DMResult getSerialNumber(wchar_t* serialNumber, int32_t len)*

Gets serial number assigned to the USB port.

The parameter *serialNumber* points to the buffer that contains the serial number. The parameter *len* is the size of the buffer.

Returns DM_RESULT_SUCCESS on success, DM_RESULT_BUFFER_TOO_SMALL if the buffer is too small.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.10: bool isAttached()*

Indicates whether this *IDevice* object is currently attached.

Returns *true* if this *IDevice* object is currently attached, *false* otherwise.

### *3.2.11: bool attach()*

Connects to the device and starts listening for input reports from the device.

Returns *true* on successful attach, *false* otherwise.

### *3.2.12: bool detach()*

Disconnects from the device and stops listening for input reports from the device.

Returns *true* on successful detach, *false* otherwise.

### *3.2.13: bool isSupported(uint16_t usage)*

Determines whether an attached device supports the specified usage.

Returns true if device supports the specific *usage*, false otherwise

### *3.2.14: DMResult getDeviceAttributes(IDeviceAttributes **ppAttributes)*

Gets an *IDeviceAttributes* object, which provides metadata about the device.

The parameter *ppAttributes* is a pointer to an instance of *IDeviceAttributes* interface.

Returns DM_RESULT_SUCCESS on success, DM_RESULT_ILLEGAL_ARGUMENT if *ppAttributes* argument is nullptr.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.15: DMResult getHostCommand(ppHostCommand ppCommand)*

Gets an *IHostCommand* object, which provides methods and properties that can be used to send specific requests to the device from the host application.

The parameter *ppCommand* is a pointer to an instance of *IHostCommand* interface.

Returns DM_RESULT_SUCCESS on success, DM_RESULT_ILLEGAL_ARGUMENT if *ppCommand* argument is nullptr.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.16: DMResult getDeviceEvents(ppDeviceEvents ppEvents)*

Gets an *IDeviceEvents* object, which provides a way to register and unregister for device events.

The parameter *ppEvents* is a pointer to an instance of *IDeviceEvents* interface

Returns DM_RESULT_SUCCESS on success, DM_RESULT_ILLEGAL_ARGUMENT if *ppEvents* argument is nullptr.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.17: DMResult getDeviceListener(ppDeviceListener ppListener)*

Gets an *IDeviceListener* object, which provides a unified interface for accessing a device.

The parameter *ppListener* is a pointer to an instance of *IDeviceListener* interface.

Returns DM_RESULT_SUCCESS on success, DM_RESULT_ILLEGAL_ARGUMENT if *ppListener* argument is nullptr.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.18: DMResult setOption(eDeviceOption eDOption, int32_t lValue) DMResult getOption(eDeviceOption eDOption, int32_t* plValue, int16_t iMaxLength, int16_t* piLength)*

Set and/or Get options for device level properties.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.19: DMResult registerCallback(pDeviceCallback pDevCallbacks) DMResult unregisterCallback(pDeviceCallback pDevCallbacks)*

Register or unregister an *IDeviceCallback*.

See Also: Section 4.1.1.1: DMResult enum.

### *3.2.20: DMResult getDeviceType(uint32_t *io_cntType, DeviceType *o_pType)*

Returns the type(s) for the given device. Most devices are of only one type, however, some complex devices (e.g., Savi 7xx) can carry sound over more than one type.

The parameter *io_cntType* is the size of the array *0_pType*, allocated by the caller, on output, and the number of entries written to the *o_pType* array. The parameter *o_pType* is the array of device types to be filled.

### *3.2.21: IDeviceManager *getDeviceManager()*

Returns the (pointer to the) Device Manager that created the device object to which this interfaces. This should be a valid pointer, as all device objects are created by Device Manager as long as the device object is valid. If device objects becomes invalid, then you may expect an error (null pointer).

## 3.1: IDeviceBaseExt Interface

Inherit this class to get a list of all connected devices..

```
INTF IDeviceBaseExt
{
   virtual DMResult getAllConnectedDevices() NOEXCEPT = 0;
};
```

### *3.1.1: virtual DMResult getAllConnectedDevices()*

Gets all devices connected via USB.

## 3.2: IDeviceCallback Interface

Inherit this class to register for Device events.

```
INTF IDeviceCallback
{
```

```
    virtual void onDataReceived(ReportDataEventArgs const& args) = 0;
}
```

### *3.2.1: void onDataReceived(ReportDataEventArgs const& args)*

Called by *ReportReader* when data is received from a device.

The parameter *args* is for data from the device.

## 3.3: IDeviceGroup Interface

Provides access to devices in a group of devices.

```
INTF IDeviceGroup : public IDisposable {
    virtual IDevice *getDevice() noexcept = 0;
    virtual uint32_t numDevices() noexcept = 0;
}
```

### *3.3.1: IDevice *getDevice()*

Returns an owning pointer to the first device from the group and removes it from the group. If there is no device in the group (the group is empty), returns nullptr.

### *3.3.2: uint32_t numDevices()*

Returns the number of devices in the group. Each successful call to *getDevice()* will decrement this number.

# Chapter 4: Using Device Listener Interfaces

Plantronics devices may provide more functionality (such as support for multiple communication endpoints VoIP, PSTN, Mobile) and these devices may require device-specific commands and events. The Device Manager exposes all of the device-specific events and commands supported by these devices. The Device Listener provides a consistent interface across all devices so that applications need not handle device-specific events and commands.

The Device Listener provides the following functionality:

- Interfaces with the Device Manager and registers for all device-specific events
- Exposes a common interface for call control commands and call control events to the Call Manager
- Handles device-specific details for sending custom data, display, alert, etc.
- Filters device events based on the active call state
- Exposes a common interface for device state, commands, and events to the Session Manager

The Device Listener might also expose additional commands and events for applications to interface with ATDs (Alternate Telephony Devices) – such as a mobile phone connected to a headset with multi-point connection to the host.

## 4.1: IDeviceListener Interface

The *IDeviceListener* interface provides a consistent interface across all Plantronics devices so that applications need not handle device-specific events and commands. Some Plantronics devices support multiple (VOIP, PTSN, and mobile ) interfaces.

```
INTF IDeviceListener{
    virtual IDeviceListenerQuery *getQuery() noexcept = 0;
    virtual DMResult ring(bool enable) noexcept = 0;
    virtual DMResult getAudioState(eAudioState &state) const noexcept = 0;
    virtual DMResult setAudioState(eAudioState value) noexcept = 0;
    virtual DMResult getMute(bool &mute)) const noexcept = 0;
    virtual DMResult setMute(bool value) noexcept = 0;
    virtual DMResult getIntellistand(bool &enabled) const noexcept = 0;
    virtual DMResult setIntellistand(bool value) noexcept = 0;
    virtual DMResult getAudioMixerState(int32_t &state) const noexcept = 0;
    virtual DMResult setAudioMixerState(int32_t value) noexcept = 0;
    virtual DMResult getDefaultLine(eLineType &line) const noexcept = 0;
    virtual DMResult setDefaultLine(eLineType value) noexcept = 0;
    virtual DMResult inCall(bool bOn) noexcept = 0;
    virtual DMResult setActiveLink(eLineType lineType, bool
        bActive) noexcept = 0;
    virtual DMResult isLineActive(eLineType lineType, bool &active)
        const noexcept = 0;
    virtual DMResult hold(eLineType lineType, bool bHold) noexcept = 0;
    virtual DMResult getHoldState(eLineType lineType, bool &hold)
        noexcept = 0;
    virtual DMResult getAudioLinkState(eAudioLinkState &state) const
        noexcept = 0;
    virtual DMResult getAudioLocation(eAudioLocation &location) const noexcept =
```

```
        0;
    virtual DMResult registerCallback(IDeviceListenerCallback*) noexcept =
        0;
    virtual DMResult unregisterCallback(IDeviceListenerCallback*)
        noexcept = 0;
    };
```

#### 4.1.1.1: DMResult enum values

```
DM_RESULT_SUCCESS = 0
DM_RESULT_FAIL = 1
DM_RESULT_BUFFER_TOO_SMALL = 2
DM_RESULT_ILLEGAL_ARGUMENT = 3
DM_RESULT_NOT_FOUND = 4
DM_RESULT_NOT_SUPPORTED = 5
DM_RESULT_DATA_NOT_AVAILABLE = 6
DM_RESULT_BUSY = 7
DM_RESULT_USAGE_NOT_FOUND = 8
```

### *4.1.2: IDeviceListenerQuery *getQuery()*

Returns the interface query pointer.

### *4.1.3: DMResult ring(bool enable)*

Rings the device if ringing is enabled. The enable parameter tells the device to enable or disable ringing.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.4: DMResult getAudioState(eAudioState &state)*

Gets the current audio state of the device.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.5: DMResult setAudioState(eAudioState value)*

Sets the current audio state of the device to the specified state. *DMResult* values are shown above.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.6: DMResult getMute(bool &mute)*

Gets the mute state of device, true if muted, false otherwise.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.7: DMResult setMute(bool value)*

Sets the mute state of the device to either muted or not.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.8: DMResult getIntellistand(bool &enabled)*

Gets auto-answer functionality state, which is enabled or not.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.9: DMResult setIntellistand(bool value)*

Sets auto-answer functionality state to enabled or not.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.10: DMResult getAudioMixerState(int32_t &state) const noexcept = 0;*

On a device that supports multiple interfaces, gets the current audio mixer state.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.11: DMResult setAudioMixerState(int32_t value)*

On a device that supports multiple interfaces, sets the current audio mixer state and returns DM_RESULT_SUCCESS on success.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.12: DMResult getDefaultLine(eLineType &line)*

On a device that supports multiple types of transport lines, gets the default line, which may be PSTN, VoIP, or Mobile.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed below.

#### 4.1.12.1: eLineType enum

```
LINE_TYPE_PSTN = 0
LINE_TYPE_VOIP = 1
LINE_TYPE_MOBILE = 2
```

### *4.1.13: DMResult setDefaultLine(eLineType value)*

On a device that supports multiple interfaces, sets the default interface and returns DM_RESULT_SUCCESS on success.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 41.

### *4.1.14: DMResult inCall(bool bOn)*

When a call is active, indicates that the device is handling a call. The parameter bOn implies that the host application uses *inCall(true)* a connected Plantronics device that it is handling a VOIP call. Using *inCall(false)* lets the device know that the VOIP call ended.

See Also: Section 4.1.1.1: DMResult enum.

### *4.1.15: DMResult setActiveLink(eLineType lineType, bool bActive)*

On a device that supports multiple interfaces, allows you to set the specified interface (PSTN, VOIP, or Mobile) as active. When *bActive (true)*, the interface is active and brings up the radio link for the specified LineType. *bActive (false)* will drop the radio link for the specific line type. Returns the result of the operation.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 41.

### *4.1.16: DMResult isLineActive(eLineType lineType, bool &active)*

On a device that supports multiple interfaces, allows you to check whether the specified interface (PSTN, VOIP, or Mobile )is active. Returns true if the line is active, false otherwise.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 41.

### *4.1.17: DMResult hold(eLineType lineType, bool bHold)*

On a device that supports multiple interfaces, puts the specified line on hold or off (resume).

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 41.

### *4.1.18: DMResult getHoldState(eLineType lineType, bool &hold)*

On a device that supports multiple interfaces, gets the current hold state for the specified interface.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 41.

### *4.1.19: DMResult getAudioLinkState(eAudioLinkState &state)*

Gets the current audio link state.

See Also: Section 4.1.1.1: DMResult enum. *eAudioLinkState* enum values are listed on page 58.

### *4.1.20: DMResult getAudioLocation(eAudioLocation &location)*

On a mobile device, gets the audio location (AG or headset).

See Also: Section 4.1.1.1: DMResult enum.

#### 4.1.20.1: eAudioLocation enum values

```
AUDIO_LOCATION_HEADSET = 0
AUDIO_LOCATION_HANDSET = 1
```

### *4.1.21: DMResult registerCallback(IDeviceListenerCallback*)*
### *DMResult unregisterCallback(IDeviceListenerCallback*)*

Registers or unregisters a callback listener for this device.

See Also: Section 4.1.1.1: DMResult enum.

## 4.2: IDeviceListenerCallback Interface

Inherit the *IDeviceListenerCallback* interface to listen for events on the base and on the headset. The receiver of DeviceListener notifications should implement this interface and register with IDeviceListener::registerCallback, in order to receive the notifications.

```
INTF IDeviceListenerCallback {
    virtual void onHeadsetButtonPressed(DeviceListenerEventArgs
        const&) = 0;
    virtual void onHeadsetStateChanged(DeviceListenerEventArgs
        const&)    = 0;
    virtual void onBaseButtonPressed(DeviceListenerEventArgs
        const&) = 0;
    virtual void onBaseStateChanged(DeviceListenerEventArgs const&)
        = 0;
    virtual void onATDStateChanged(DeviceListenerEventArgs const&)
        = 0;
    };
```

### 4.2.1.1: DeviceListenerEventArgs struct

```
struct DeviceListenerEventArgs
{
    enum
    {
        MAX_DEVICE_PATH = 256
    };
    eDeviceEventType deviceEventType;
    eHeadsetButton headsetButton;
    eHeadsetStateChange headsetStateChange;
    eBaseButton baseButton;
    int32_t dialedKey;
    eBaseStateChange baseStateChange;
    eATDStateChange ATDStateChange;
    wchar_t devicePath[MAX_DEVICE_PATH];
};
```

## *4.2.2: void onHeadsetButtonPressed(DeviceListenerEventArgs const&)*

Callback that listens for the pressing of a button on the headset.

### 4.2.2.1: eHeadsetButton enum

```
HEADSET_BUTTON_UNKNOWN = 0
HEADSET_BUTTON_VOLUME_UP = 1
HEADSET_BUTTON_VOLUME_DOWN = 2
HEADSET_BUTTON_VOLUME_UP_HELD = 3
HEADSET_BUTTON_VOLUME_DOWN_HELD = 4
HEADSET_BUTTON_MUTE = 5
```

```
HEADSET_BUTTON_MUTE_HELD = 6
HEADSET_BUTTON_TALK = 7
HEADSET_BUTTON_AUDIO = 8
HEADSET_BUTTON_PLAY = 9
HEADSET_BUTTON_PAUSE = 10
HEADSET_BUTTON_NEXT = 11
HEADSET_BUTTON_PREVIOUS = 12
HEADSET_BUTTON_FAST_FORWARD = 13
HEADSET_BUTTON_REWIND = 14
HEADSET_BUTTON_STOP = 15
HEADSET_BUTTON_FLASH = 16
HEADSET_BUTTON_SMART = 17

//Handset buttons
HEADSET_BUTTON_OFF_HOOK = 18
HEADSET_BUTTON_ON_HOOK = 19
HEADSET _BUTTON_KEY_0 = 20
HEADSET_BUTTON_KEY_1 = 21
HEADSET_BUTTON_KEY_2 = 22
HEADSET_BUTTON_KEY_3 = 23
HEADSET_BUTTON_KEY_4 = 24
HEADSET_BUTTON_KEY_5 = 25
HEADSET_BUTTON_KEY_6 = 26
HEADSET_BUTTON_KEY_7 = 27
HEADSET_BUTTON_KEY_8 = 28
HEADSET_BUTTON_KEY_9 = 29
HEADSET_BUTTON_KEY_STAR = 30
HEADSET_BUTTON_KEY_POUND = 31
HEADSET_BUTTON_SPEAKER = 32
HEADSET_BUTTON_REJECT = 33
```

### *4.2.3: void onHeadsetStateChanged(DeviceListenerEventArgs const&)*

Callback that listens for a state change on the headset. Possible state changes are listed below in the section eHeadsetStateChange enum.

#### 4.2.3.1: eHeadsetStateChange enum

```
HS_STATE_CHANGE_UNKNOWN = 0
HS_STATE_CHANGE_MONO_ON = 1
HS_STATE_CHANGE_MONO_OFF = 2
HS_STATE_CHANGE_STEREO_ON = 3
HS_STATE_CHANGE_STEREO_OFF = 4
HS_STATE_CHANGE_MUTE_ON = 5
HS_STATE_CHANGE_MUTE_OFF = 6
HS_STATE_CHANGE_BATTERY_LEVEL = 7
HS_STATE_CHANGE_IN_RANGE = 8
HS_STATE_CHANGE_OUT_OF_RANGE = 9
HS_STATE_CHANGE_DOCKED = 10
```

```
HS_STATE_CHANGE_UNDOCKED = 11
HS_STATE_CHANGE_IN_CONFERENCE = 12
HS_STATE_CHANGE_DON=13
HS_STATE_CHANGE_DOFF=14
HS_STATE_CHANGE_SERIAL_NUMBER = 15
HS_STATE_CHANGE_NEAR = 16
HS_STATE_CHANGE_FAR = 17
HS_STATE_CHANGE_DOCKED_CHARGING = 18
HS_STATE_CHANGE_PROXIMITY_UNKNOWN = 19
HS_STATE_CHANGE_PROXIMITY_ENABLED = 20
HS_STATE_CHANGE_PROXIMITY_DISABLED = 21
HS_STATE_CHANGE_CONNECTED = 29
HS_STATE_CHANGE_DISCONNECTED = 30
```

## *4.2.4: void onBaseButtonPressed(DeviceListenerEventArgs const&)*

Callback that listens for a button to be pressed on the base unit. The possible options are listed below in section eBaseButton enum.

### 4.2.4.1: eBaseButton enum

```
BASE_BUTTON_UNKNOWN = 0
BASE_BUTTON_PSTN_TALK = 1
BASE_BUTTON_VOIP_TALK = 2
BASE_BUTTON_SUBSCRIBE = 3
BASE_BUTTON_PSTN_TALK_HELD = 4
BASE_BUTTON_VOIP_TALK_HELD = 5
BASE_BUTTON_SUBSCRIBE_HELD = 6
BASE_BUTTON_PSTN_TALK_AND_SUBSCRIBE_HELD = 7
BASE_BUTTON_PSTN_TALK_AND_VOIP_TALK_HELD = 8
BASE_BUTTON_MAKE_CALL = 9
BASE_BUTTON_MOBILE_TALK = 10
BASE_BUTTON_MOBILE_TALK_HELD = 11
BASE_BUTTON_PSTN_TALK_AND_MOBILE_TALK_HELD = 12
BASE_BUTTON_VOIP_TALK_AND_MOBILE_TALK_HELD = 13
BASE_BUTTON_DIAL_PAD = 14
BASE_BUTTON_MAKE_CALL_FROM_CALL_LOG = 15
```

## *4.2.5: void onBaseStateChanged(DeviceListenerEventArgs const&)*

Callback that listens for a state change on the base unit. Possible state changes are listed below in section eBaseStateChange enum.

### 4.2.5.1: eBaseStateChange enum

```
BASE_STATE_CHANGE_UNKNOWN = 0
BASE_STATE_CHANGE_PSTN_LINK_ESTABLISHED = 1
BASE_STATE_CHANGE_PSTN_LINK_DOWN = 2
BASE_STATE_CHANGE_VOIP_LINK_ESTABLISHED = 3
```

```
BASE_STATE_CHANGE_VOIP_LINK_DOWN = 4
BASE_STATE_CHANGE_AUDIO_MIXER = 5
BASE_STATE_CHANGE_RF_LINK_WIDE_BAND = 6
BASE_STATE_CHANGE_RF_LINK_NARROW_BAND = 7
BASE_STATE_CHANGE_MOBILE_LINK_ESTABLISHED = 8
BASE_STATE_CHANGE_MOBILE_LINK_DOWN = 9
BASE_STATE_CHANGE_INTERFACE_STATE_CHANGED = 10
BASE_STATE_CHANGE_AUDIO_LOCATION_CHANGED = 11
BASE_STATE_CHANGE_SERIAL_NUMBER = 12
BASE_STATE_CHANGE_PRODUCTION_NUMBER = 13
BASE_STATE_CHANGE_PRODUCTION_BUILD = 14
BASE_STATE_CHANGE_PRODUCTION_PART_NUMBER = 15
BASE_STATE_CHANGE_MOBILE_CONNECTED = 16
BASE_STATE_CHANGE_MOBILE_DISCONNECTED = 17
```

### *4.2.6: void onATDStateChanged(DeviceListenerEventArgs const&)*

Callback that listens for a state change on the ATD device. Possible state changes are listed in section eATDStateChange enum.

#### 4.2.6.1: eATDStateChange enum

```
ATD_STATE_CHANGE_UNKNOWN = -1
ATD_STATE_CHANGE_MOBILE_CALL_ENDED = 0
ATD_STATE_CHANGE_MOBILE_INCOMING = 1
ATD_STATE_CHANGE_MOBILE_OUTGOING = 2
ATD_STATE_CHANGE_MOBILE_ON_CALL = 3
ATD_STATE_CHANGE_PSTN_INCOMING_CALL_RING_ON = 4
ATD_STATE_CHANGE_PSTN_INCOMING_CALL_RING_OFF = 5
ATD_STATE_CHANGE_DESKPHONE_HEADSET = 6
ATD_STATE_CHANGE_MOBILE_CALLER_ID = 7
```

## 4.3: IDeviceListenerQuery Interface

The *IDeviceListenerQuery* interface obtains an interface from the *IDeviceListener* hierarchy. It is needed because `dynamic_cast<>` does not work across DLL (.so) boundaries.

```
INTF IDeviceListenerQuery{
    virtual bool query(IDeviceListener **p) = 0;
    virtual bool query(IDisplayDeviceListener **p) = 0;
    };
```

### *4.3.1: bool query(IDeviceListener **p)*

The method obtains an *IDeviceListener* interface.

### *4.3.2: bool query(IDisplayDeviceListener **p)*

The method obtains an *IDisplayDeviceListener* interface.

## 4.4: IDisplayDeviceCall Interface

The *IDisplayDeviceCall* is an interface for getting and setting call parameters for calls on Display devices.

```
INTF IDisplayDeviceCall : public IDisposable
{
    virtual DM_GUID getSessionId() const = 0;
    virtual DMResult setSessionId(DM_GUID guid) = 0;
    virtual int32_t getCallId() const = 0;
    virtual DMResult setCallId(int32_t callId) = 0;

    virtual DDCallState getCallState() const = 0;
    virtual DMResult setCallState(DDCallState callState) = 0;

    virtual DDSoftphoneID getSoftphoneID() const = 0;
    virtual DMResult setSoftphoneID(DDSoftphoneID spID) = 0;
    virtual wchar_t const *getCallerId() const = 0;
    virtual DMResult setCallerId(wchar_t const *callerId) = 0;
    virtual wchar_t const *getFriendlyName() const = 0;
    virtual DMResult setFriendlyName(wchar_t const *name) = 0;
```

### *4.4.1: DM_GUID getSessionId()*

Gets the session ID of the plug-in that initiated the call to the display device. A session ID is a unique number that a Web site's server assigns to a specific user for the duration of that user's visit (session).

### *4.4.2: DMResult setSessionId(DM_GUID guid)*

Sets the session ID of the plug-in that initiated the call to the display device. A session ID is a unique number that a Web site's server assigns to a specific user for the duration of that user's visit (session).

See Also: Section 4.1.1.1: DMResult enum.

### *4.4.3: int32_t getCallId()*

Returns the call ID assigned to the call.

### *4.4.4: DMResult setCallId(int32_t callId)*

Sets the call ID assigned to the call.

See Also: Section 4.1.1.1: DMResult enum.

### *4.4.5: DDCallState getCallState()*

Returns the call state of the call.

*DDCallState* options are listed below.

#### 4.4.5.1: DDCallState enum

```
DD_CALL_STATE_UNKNOWN = 0
DD_CALL_STATE_ENDED = 1
DD_CALL_STATE_INCOMING = 2
DD_CALL_STATE_OUTGOING = 3
DD_CALL_STATE_ACTIVE = 4
DD_CALL_STATE_HELD = 5
```

### *4.4.6: DMResult setCallState(DDCallState callState)*

Sets the call state of the call.

See Also: Section 4.1.1.1: DMResult enum. *DDCallState* enum values are listed on page 48.

### *4.4.7: DDSoftphoneID getSoftphoneID()*

This method gets the identifier for the softphone being used for the call.

See Also: 4.4.8.1: DDSoftphoneID enum.

### *4.4.8: DMResult setSoftphoneID(DDSoftphoneID spID)*

This method specifies the ID for the softphone being used.

Possible values for the softphone ID are shown below in *DDSoftphoneID* enum.

See Also: Section 4.1.1.1: DMResult enum.

#### 4.4.8.1: DDSoftphoneID enum

```
DD_SOFTPHONE_ID_UNKNOWN = 0
DD_SOFTPHONE_ID_MS_OFFICE_COMMUNICATOR = 1
DD_SOFTPHONE_ID_AVAYA_ONE_X_COMMUNICATOR = 2
DD_SOFTPHONE_ID_AVAYA_ONE_X_AGENT = 3
DD_SOFTPHONE_ID_AVAYA_IP_SOFTPHONE = 4
DD_SOFTPHONE_ID_AVAYA_IP_AGENT = 5
DD_SOFTPHONE_ID_SKYPE = 6
DD_SOFTPHONE_ID_SHORE_TEL_COMMUNICATOR = 8
DD_SOFTPHONE_ID_CISCO_IP_COMMUNICATOR = 9
DD_SOFTPHONE_ID_CSF = 10
DD_SOFTPHONE_ID_NECSP350 = 11
DD_SOFTPHONE_ID_MS_LYNC_2013 = 12
DD_SOFTPHONE_ID_CISCO_WEBEX = 13
DD_SOFTPHONE_ID_CISCO_JABBER = 14
DD_SOFTPHONE_ID_SWYX= 15
```

### *4.4.9: wchar t const *getCallerId()*

This method discovers the caller ID, which identifies and displays the number or ID of incoming calls.

### *4.4.10: DMResult setCallerId(wchar_t const *callerId)*

This method specifies the caller ID, which identifies and displays the number or ID of incoming calls.

See Also: Section 4.1.1.1: DMResult enum.

### *4.4.11: wchar_t const *getFriendlyName()*

This method gets the friendly name of the device, which is something a human can easily read and remember instead of a code or a numeric address/identifier.

### *4.4.12: DMResult setFriendlyName(wchar_t const *name)*

This method sets the friendly name for the device, which is something a human can easily read and remember instead of a code or a numeric address/identifier.

See Also: Section 4.1.1.1: DMResult enum.

## 4.5: IDisplayDeviceListener Interface

This *IDisplayDeviceListener* interface provides methods for setting fields that affect the device's display.

```
INTF IDisplayDeviceListener{
    virtual DMResult setLocale(uint32_t lcid) = 0;
    virtual DMResult setPresence(DDSoftphoneID spID, DDPresence
        presence) = 0;
    virtual DMResult setDefaultSoftphone(DDSoftphoneID spID) = 0;
    virtual DMResult setDateTime(DM_DateTime const& dt) = 0;
    virtual DMResult setDateTimeFormat(uint32_t lcid) = 0;
    virtual DMResult setSoftphoneName(DDSoftphoneID spID, wchar_t
        const *spName) = 0;
    virtual DMResult setMultiCallState(int32_t nCount,
        IDisplayDeviceCall **ddCall) = 0;
    virtual DDSoftphoneID getSoftphoneID() = 0;
    virtual wchar_t const *getMakeCallName() = 0;
```

### *4.5.1: DMResult setLocale(uint32_t lcid)*

This method sets the locale of the device. Locale is a set of parameters that define the user's language and region. The lcid parameter is the locale id supported by Display Devices.

See Also: Section 4.1.1.1: DMResult enum.

### *4.5.2: DMResult setPresence(DDSoftphoneID spID, DDPresence presence)*

This method sets the presence of the device. Presence is the availability status of the softphone.

Options for the  and 4.4.8.1: DDSoftphoneID enum.

DDPresence enum are listed below.

See Also: Section 4.1.1.1: DMResult enum and 4.4.8.1: DDSoftphoneID enum.

DDPresence enum values

```
DD_PRESENCE_UNKNOWN = 0
DD_PRESENCE_AVAILABLE = 1
DD_PRESENCE_AWAY = 2
DD_PRESENCE_BUSY = 3
DD_PRESENCE_DND = 4
DD_PRESENCE_IN_CALL = 5
DD_PRESENCE_CLOSED = 6
```

## *4.5.3: DMResult setDefaultSoftphone(DDSoftphoneID spID)*

This method sets the default softphone identifier for the device.

See Also: Section 4.1.1.1: DMResult enum and 4.4.8.1: DDSoftphoneID enum.

## *4.5.4: DMResult setDateTime(DM_DateTime const& dt)*

This method sets the date and time for the device.

The DM_DateTime struct is listed below.

See Also: Section 4.1.1.1: DMResult enum.

### 4.5.4.1: DM_DateTime struct

```
struct DM_DateTime {
    /// The full year A.D. number
    uint16_t year;
    /// The number of the month in a year, 1-12
    uint8_t month;
    /// The number of day in a month, 1-31
    uint8_t day;
    /// The number of hour in a day, 0-23
    uint8_t hour;
    /// The number of minute in a hour, 0-59
    uint8_t minute;
    /// The number of seconds in a minute, 0-59 in general,
    /// can be more due to leap seconds
    uint8_t secs;
};
```

## *4.5.5: DMResult setDateTimeFormat(uint32_t lcid)*

This method sets the format to be used to display the date and time. *lcid* is the locale that provides the date time format. The *lcid* parameter is the locale ID for Display Devices.

See Also: Section 4.1.1.1: DMResult enum.

### *4.5.6: DMResult setSoftphoneName(DDSoftphoneID spID, wchar_t const spName)*

This method sets the name of the softphone and connects it with the softphone identifier.

See Also: Section 4.1.1.1: DMResult enum and 4.4.8.1: DDSoftphoneID enum.

### *4.5.7: DMResult setMultiCallState(int32_t nCount, IDisplayDeviceCall **ddCall)*

Sets the state of (all active/held) calls for the device. These will be displayed on the device.

See Also: Section 4.1.1.1: DMResult enum.

### *4.5.8: DDSoftphoneID getSoftphoneID()*

This method discovers the ID of the softphone.

See Also: Section 4.4.8.1: DDSoftphoneID enum.

### *4.5.9: wchar_t const *getMakeCallName()*

Returns the string of the name with which to make a call.

# Chapter 5: Using Host Command Interfaces

This section contains interfaces of the *IHostCommand* hierarchy. Host command interfaces send requests to the attached device from the host application.

## 5.1: IHostCommand Interface

The *IHostCommand* interface provides methods and properties that send specific requests to the attached device from the host application. This is the root of the *HostCommand* interface hierarchy. All other interfaces are optional.

```
INTF IHostCommand{
    virtual IHostCommandQuery *getQuery() noexcept = 0;
    virtual DMResult setRing(bool bEnable) noexcept = 0;
    virtual DMResult getAudioState(eAudioState &state) const
        noexcept= 0;
    virtual DMResult setAudioState(eAudioState audioType) noexcept = 0;
    virtual DMResult getMute(bool &mute) const noexcept = 0;
    virtual DMResult registerCommand(eRegistrationType sign) noexcept = 0;
    virtual DMResult getVersion(eVersionType version, wchar_t*
        buffer, uint32_t bufferLength) const noexcept = 0;
    virtual bool isSupported(eFeatureType feature) const noexcept = 0;
```

### *5.1.1: IHostCommandQuery *getQuery()*

This method returns the query pointer.

### *5.1.2: DMResult setRing(bool bEnable)*

This method controls the headset ringer. When *bEnable* is true, sends a command to the device to start the ringer of the device. On devices that do not have a ringer, this call will be a NOP. When *bEnable* is false, sends a command to the device to stop the ringer.

The *setRing* usage may be required to set the device into a ringing state, regardless of whether the device has a local audible ringer.

Applications should use the *setRing* method to indicate an incoming call. This should be done in addition to playing a ring-tone through the headset audio stream. This is done because wireless headsets may turn off the audio channel to conserve the battery power. If audio is disabled, no ring-tone will be heard until the radio link is established. Use of the *setRing* method allows a ring-tone to be signaled out-of-band, and thus allows the wearer to hear the ring even if the audio link is not established. Using the *setRing* method in addition to the audio-channel ring-tone will ensure that the ring is heard regardless of the audio link state.

See Also: Section 4.1.1.1: DMResult enum.

### *5.1.3: DMResult getAudioState(eAudioState &state)*

This method gets the audio stream state of the attached device. This method returns a constant value from *eAudioState* enum that represents current the audio state.

See Also: Section 4.1.1.1: DMResult enum. *eAudioState* enum values are listed below.

## *5.1.4: DMResult setAudioState(eAudioState audioType)*

This method enables or disables the audio stream to the device. For a wireless device this method also controls the audio link. The link is up when audio is enabled by setting the *AudioState* to *AudioType.MonoOn* and the link is down when the audio is disabled by setting the *AudioState* to *AudioType.MonoOff*.

Applications should use this method to conserve battery power on wireless devices. (For example, a telephony application would keep the audio disabled while there are no active calls and then enable the audio when a call becomes active.)

See Also: Section 4.1.1.1: DMResult enum. The supported values for *eAudioState* enums are shown below.

### 5.1.4.1: eAudioState enums

```
AUDIO_STATE_UNKNOWN= 0
AUDIO_STATE_MONOON = 1
AUDIO_STATE_MONOOFF = 2
AUDIO_STATE_STEREOON = 3
AUDIO_STATE_STEREOOFF = 4
AUDIO_STATE_MONOONWAIT = 5
AUDIO_STATE_STEREOONWAIT = 6
```

## *5.1.5: DMResult getMute(bool &mute)*

This method gets the mute state of the microphone in the device. It returns true if the device is muted, or false if it is not muted.

See Also: Section 4.1.1.1: DMResult enum.

## *5.1.6: DMResult registerCommand(eRegistrationType sign)*

This method signs a host application in or out. Signing on lets the device know that the host application is running.

Host applications are required to Sign On and Sign Off when they start and shutdown in order to notify the device that they are running and are interested in listening to device events. If the host application does not register, some devices may not send input reports on a specific HID usage page.

Options for *See* Also: Section 4.1.1.1: DMResult enum.

eRegistrationType enum are listed below.

See Also: Section 4.1.1.1: DMResult enum.

### 5.1.6.1: eRegistrationType enum values

```
REGISTRATION_SIGNON = 1
REGISTRATION_SIGNOFF = 2
REGISTRATION_EXCLUSIVE = 4
REGISTRATION_CALLMANAGER = 8
REGISTRATION_ATTACH = 0x10
REGISTRATION_DETACH = 0x20
REGISTRATION_DETACH_SIGNOFF = 0x40
REGISTRATION_EXSIGN = REGISTRATION_EXCLUSIVE | REGISTRATION_SIGNON
```

```
REGISTRATION_EXSIGNCM = REGISTRATION_EXCLUSIVE | REGISTRATION_SIGNON |
REGISTRATION_CALLMANAGER
```

## *5.1.7: DMResult getVersion(eVersionType version, wchar_t* buffer, uint32_t bufferLength)*

This method allows the host application to request the USB firmware version, the base firmware version, and the headset firmware version of a particular Plantronics device. The buffer includes a version string, or is empty if the specified version is not applicable for the current device. This method returns DM_RESULT_SUCCESS on success, DM_RESULT_BUFFER_TOO_SMALL if the buffer is too small, and DM_RESULT_FAIL on fail.

Options for *See* Also: Section 4.1.1.1: DMResult enum.

eVersionType enum are listed below.

See Also: Section 4.1.1.1: DMResult enum.

### 5.1.7.1: eVersionType enum values

```
VERSION_TYPE_USB = 1
VERSION_TYPE_BASE = 2
VERSION_TYPE_REMOTE = 3
VERSION_TYPE_BLUETOOTH = 4
VERSION_TYPE_TUNING = 5,
VERSION_TYPE_PIC = 6
```

## *5.1.8: bool isSupported(eFeatureType feature) const*

This method determines whether the attached device supports the specified feature. This method returns true if the feature is supported, false otherwise.

Values for the *eFeatureType* enum are listed below.

### 5.1.8.1: eFeatureType enum

```
FEATURE_TYPE_ANTI_STARTLE = 1
FEATURE_TYPE_G616  = 2
FEATURE_TYPE_AUDIO_LIMITING = 3
FEATURE_TYPE_DFU = 4
FEATURE_TYPE_AUTO_ANSWER= 5
FEATURE_TYPE_CORDED = 6
FEATURE_TYPE_BATTERY_LEVEL = 7
FEATURE_TYPE_HEADSET_CONFERENCE_MODE= 8
FEATURE_TYPE_DONGLE = 9
FEATURE_TYPE_MAKE_CALL = 10
FEATURE_TYPE_WEARABLE = 11
FEATURE_TYPE_TWA_PERIOD = 12
FEATURE_TYPE_MULTIPLE_INTERFACE_SUPPORT =13
FEATURE_TYPE_BT_DFU =14
FEATURE_TYPE_BT_HEADSET = 15
FEATURE_TYPE_PERIPHERAL_DEVICE = 16
```

```
FEATURE_TYPE_OC_MUTE_LOCAL = 17
FEATURE_TYPE_DISPLAY_DEVICE = 18
```

## 5.2: IHostCommandExt Interface

The *IHostCommandExt* interface provides methods that send specific requests to the attached device from the host application.

```
INTF IHostCommandExt{
    virtual DMResult getBatteryLevel(eBatteryLevel &level) const noexcept = 0;
    virtual DMResult getHeadsetsInConference(int32_t &count) const noexcept= 0;
    virtual DMResult setHeadsetMute(bool bMute) noexcept = 0;
    virtual DMResult getHeadsetType(eHeadsetType &hs) const noexcept = 0;
    virtual DMResult sendVersionRequest()noexcept = 0;
    virtual DMResult setActiveLink(eLineType lineType, bool
        bActive) noexcept = 0;
    virtual DMResult isLineActive(eLineType i_lineType, bool
        &o_active) const noexcept = 0;
    virtual DMResult isLineConnected(eLineType i_lineType, bool &o_active)
        const NOEXCEPT = 0;
    virtual DMResult hold(eLineType lineType, bool bHold) noexcept = 0;
    virtual DMResult getHoldState(eLineType i_lineType, bool
        &o_hold) noexcept = 0;
    virtual DMResult getAudioLinkState(eAudioLinkState &state)
        const noexcept = 0;
    virtual DMResult isHeadsetDocked(bool& isHsDocked) const noexcept = 0;
    virtual DMResult getHeadsetProductName(wchar_t* buffer, int32_t
        length) noexcept = 0;
    virtual DMResult getAudioLocation(eAudioLocation
        &location) const noexcept = 0;
    virtual DMResult getBatteryInfo(eBTBatteryStatus*
        btBatteryStatus, int32_t* life, eBTChargingStatus*
        btChargingStatus) noexcept = 0;
    virtual DMResult sendSerialNumberReq(Plantronics::DM::eDeviceType
        devType) noexcept = 0;
    virtual DMResult getSerialNumber(Plantronics::DM::eDeviceType devType,
        uint8_t* serialNumber, int32_t serialNumberBufferSize) const
        noexcept = 0;
    virtual isConstant(uint16_t usage, bool) const noexcept = 0;
    virtual DMResult getProximity() noexcept = 0;
    virtual DMResult enableProximity(bool bEnable) noexcept = 0;
    virtual DMResult getHeadsetState(eHeadsetState
        &state) const noexcept = 0;
    virtual DMResult HeadsetPairing(bool on) noexcept = 0;
    };
```

## *5.2.1: eBatteryLevel getBatteryLevel()*

This method gets the battery level of the device, and returns the value from the *eBatteryLevel* enum or `BATTERY_LEVEL_EMPTY` for devices that do not have a battery.

Values for the *eBatteryLevel* enum are listed below.

### 5.2.1.1: eBatteryLevel enum

```
BATTERY_LEVEL_EMPTY = 0
BATTERY_LEVEL_LOW =
BATTERY_LEVEL_MEDIUM = 2
BATTERY_LEVEL_HIGH = 3
BATTERY_LEVEL_FULL = 4
```

## *5.2.2: DMResult getHeadsetsInConference(int32_t &count)*

This method gets the number of headsets currently in conference.

See Also: Section 4.1.1.1: DMResult enum.

## *5.2.3: DMResult setHeadsetMute(bool bMute)*

This method sets the mute state of headset. When *bMute* is true,the headset is muted, when *bMute* is false, the headset is unmuted.

See Also: Section 4.1.1.1: DMResult enum.

## *5.2.4: DMResult getHeadsetType(eHeadsetType &hs)*

This method gets the headset type as a value from the *eHeadsetType* enum, the values for which are listed below.

See Also: Section 4.1.1.1: DMResult enum.

### 5.2.4.1: eHeadsetType enum values

```
HEADSET_TYPE_INVALID = 0xFF
HEADSET_TYPE_UNKNOWN = 0
HEADSET_TYPE_THEO_428 = 1
HEADSET_TYPE_EROS_428 = 2
HEADSET_TYPE_HELIOS_480_MONOAURAL = 3
HEADSET_TYPE_HELIOS_480_BINAURAL = 4
HEADSET_TYPE_THEO_480 = 5
HEADSET_TYPE_EROS_480 = 6
HEADSET_TYPE_UNKNOWN_480 = 7
HEADSET_TYPE_HERMIT_480 = 8
HEADSET_TYPE_HERMIT_LITE = 9
HEADSET_TYPE_HERMIT_4804MM = 10
HEADSET_TYPE_HERMIT_LITE_4MM = 11
HEADSET_TYPE_UNKNOWN_428ROM = 0x80     // unknown 428 top ROM
HEADSET_TYPE_THEO_428ROM    // 428 Theo ROM
HEADSET_TYPE_EROS_428ROM    // 428 Eros ROM
```

```
HEADSET_TYPE_HELIOS_MONOAURAL_480ROM   // Helios mono (can only be
    480) ROM
HEADSET_TYPE_HELIOS_BINAURAL_480ROM    // Helios binaural (can only
    be 480) ROM
HEADSET_TYPE_THEO_480ROM    // 480 Theo ROM
HEADSET_TYPE_EROS_480ROM    // 480 Eros ROM
HEADSET_TYPE_UNKNOWN_480ROM            // unknown 480 top ROM
HEADSET_TYPE_HERMIT_480ROM   // Hermit (can only be 480) ROM
HEADSET_TYPE_HERMIT_LITE_480ROM     // Hermit Lite (can only
    be 480) ROM
HEADSET_TYPE_HERMIT_4MM480ROM       // Hermit with 4mm mic (can
    only be 480) ROM
HEADSET_TYPE_HERMIT_LITE_4MM480ROM   // Hermit Lite with 4mm mic
    (can only be 480) ROM
```

## *5.2.5: DMResult sendVersionRequest()*

This method gets the firmware version from the device.

See Also: Section 4.1.1.1: DMResult enum.

## *5.2.6: DMResult setActiveLink(eLineType lineType, bool bActive)*

This method sets the currently active link. The *lineType* parameter specifies the line for which to change the active state (VOIP, PSTN or Mobile) from the eLineType enum. When *bActive* is true, activates the specified line type, when *bActive* is false, deactivates specified line type. This method returns DM_RESULT_SUCCESS on success, DM_RESULT_FAIL on fail.

See Also: Section 4.1.1.1: DMResult enum. The *eLineType* enum values are listed below.

### 5.2.6.1: eLineType enum values

```
LINE_TYPE_PSTN = 0
LINE_TYPE_VOIP = 1
LINE_TYPE_MOBILE = 2
```

## *5.2.7: DMResult isLineActive(eLineType i_lineType, bool &o_active)*

This method checks to determine if specified line is active. The *lineType* parameter specifies the type of line for which the active state should be checked (VOIP, PSTN or Mobile). This method returns true if the specified line type is active, false otherwise.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed above.

## *5.2.8: DMResult isLineConnected(eLineType i_lineType, bool &o_active)*

This method checks to determine if the line specified in *i_lineType* is connected. The *i_lineType* parameter specifies the line to check active state (VOIP, PSTN or Mobile) from the eLineType enum. When the *&o_active* parameter is true, the specified line is active, when false, the specified line type is inactive. This method returns DM_RESULT_SUCCESS on success, DM_RESULT_FAIL otherwise.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 57.

## *5.2.9: DMResult hold(eLineType lineType, bool bHold)*

This method puts the specified line on hold. The *lineType* parameter specifies the line to hold (VOIP, PSTN or Mobile) from the eLineType enum. When the *bHold* parameter is true, the specified line is placed on hold, when false, the specified line is taken off of hold. This method returns DM_RESULT_SUCCESS on success, DM_RESULT_FAIL otherwise.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 57.

## *5.2.10: DMResult getHoldState(eLineType i_lineType, bool &o_hold)*

This method gets the hold state of the specified line. The *lineType* parameter specifies the line to check for its hold state (VOIP, PSTN or Mobile) from the eLineType enum, the values for which are shown on page 57. This method returns true if the specified line type is on hold, false otherwise.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 57.

## *5.2.11: DMResult getAudioLinkState(eAudioLinkState &state)*

This method gets the current audio link state.

See Also: Section 4.1.1.1: DMResult enum. *eAudioLinkState* enum values are listed below.

### 5.2.11.1: eAudioLinkState enum values

```
AUDIO_LINK_STATE_IDLE = 0
AUDIO_LINK_STATE_PENDING = 1
AUDIO_LINK_STATE_ESTABLISHED = 2
AUDIO_LINK_STATE_FAILED = 3
AUDIO_LINK_STATE_RELEASE_PENDING=4
AUDIO_LINK_STATE_LOST = 5
```

## *5.2.12: DMResult isHeadsetDocked(bool& isHsDocked)*

This method checks to determine if the headset is docked. The parameter *isHsDocked* returns true if the headset is docked.

See Also: Section 4.1.1.1: DMResult enum.

## *5.2.13: DMResult getHeadsetProductName(wchar_t* buffer, int32_t length)*

This method gets the headset product name.

See Also: Section 4.1.1.1: DMResult enum.

## *5.2.14: DMResult getAudioLocation(eAudioLocation &location)*

This method gets the location of the audio, headset or handset, from the *eAudioLocation* enum (values listed below).

See Also: Section 4.1.1.1: DMResult enum.

### 5.2.14.1: eAudioLocation enum values

```
AUDIO_LOCATION_HEADSET = 0
AUDIO_LOCATION_HANDSET = 1
```

## *5.2.15: DMResult getBatteryInfo(eBTBatteryStatus* btBatteryStatus, int32_t* life, eBTChargingStatus* btChargingStatus)*

This method gets battery info, such as the battery status, the life of the battery, and the charging status of the battery.

See Also: Section 4.1.1.1: DMResult enum. Options for the *eBTBatteryStatus* enum and the *eBTChargingStatus* enum are listed below.

### 5.2.15.1: eBTBatteryStatus enum values

```
BT_BATTERY_STATUS_UNKNOWN = 0
BT_BATTERY_STATUS_CRITICAL = 1
BT_BATTERY_STATUS_LOW = 2
BT_BATTERY_STATUS_MEDIUM = 3
BT_BATTERY_STATUS_HIGH = 4
BT_BATTERY_STATUS_FULL = 5
BT_BATTERY_STATUS_NOT_BATTERY_POWERED = 6
```

## *5.2.16: DMResult sendSerialNumberReq(eDeviceType devType)*

This method gets the serial number for the specified device type.

Options for *eDeviceType* enum are listed below.

See Also: Section 4.1.1.1: DMResult enum.

### 5.2.16.1: eDeviceType enum values

```
DEVICE_TYPE_BASE = 1
DEVICE_TYPE_HEADSET = 2
```

## *5.2.17: DMResult getSerialNumber(Plantronics::DM::eDeviceType devType, uint8_t* serialNumber, int32_t serialNumberBufferSize)*

This method attempts to retrieve the persisted device type (base or headset) serial number. Serial number is a GUID (or alike) number.

The parameter *devType* specifies the device type for which to send get the serial number. The parameter *serialNumber* is a pointer to the serial number buffer. The parameter *serialNumberBufferSize* should be exactly 16.

Returns DM_RESULT_SUCCESS if successful, DM_RESULT_DATA_NOT_AVAILABLE if data not available (not requested probably), or DM_RESULT_BUFFER_TOO_SMALL if buffer sent is too small.

See Also: Section 4.1.1.1: DMResult enum. *eDeviceType* enum values are listed above.

## *5.2.18: bool isConstant(uint16_t usage)*

This method checks to determine if the specified usage is constant.

## *5.2.19: DMResult getProximity()*

This method sends a request for proximity to device. The value is returned as a *HeadsetStateChanged* event.

Options for the *eProximity* enum are listed below.

See Also: Section 4.1.1.1: DMResult enum.

#### 5.2.19.1: eProximity enum values

```
PROXIMITY_UNKNOWN = 0
PROXIMITY_NEAR = 1
PROXIMITY_FAR = 2
PROXIMITY_DISABLED = 3
PROXIMITY_ENABLED = 4
```

### *5.2.20: DMResult enableProximity(bool bEnable)*

This method sets the proximity state. True enables proximity, false disables proximity.

See Also: Section 4.1.1.1: DMResult enum.

### *5.2.21: DMResult getHeadsetState(eHeadsetState &state)*

This method gets headset state from the *eHeadsetState* enum *(below)*.

#### 5.2.21.1: eHeadsetState enum

```
HEADSET_STATE_UNKNOWN = 0
HEADSET_STATE_IN_RANGE = 1
HEADSET_STATE_OUT_OF_RANGE = 2
HEADSET_STATE_DOCKED = 3
HEADSET_STATE_UNDOCKED = 4
HEADSET_STATE_IN_CONFERENCE = 5
HEADSET_STATE_DON = 6
HEADSET_STATE_DOFF = 7
HEADSET_STATE_BTRF_POWER_MODE = 8
HEADSET_STATE_DOCKED_CHARGING = 9
HEADSET_STATE_PRODUCT_NAME = 10
HEADSET_STATE_BATTERY_INFO = 11
HEADSET_STATE_SERIAL_NUMBER = 12
HEADSET_STATE_PROXIMITY = 13
HEADSET_STATE_CONNECTED = 14
HEADSET_STATE_DISCONNECTED = 15
HEADSET_STATE_VOICE_DETECTED = 16
HEADSET_STATE_SILENCE_DETECTED = 17
HEADSET_STATE_VISIBLE_MUTE_ALERT_ON = 18
HEADSET_STATE_VISIBLE_MUTE_ALERT_OFF = 19
```

## 5.3: IHostCommandOption Interface

This interface is used to get and set options for hostcommand level properties for HID 2.0.

```
INTF IHostCommandOption{
    virtual DMResult setOption(eCommandOption eCOption, int32_t lValue) NOEXCEPT = 0;
    virtual DMResult getOption(eCommandOption eCOption, int32_t* plValue,
        int16_t iMaxLength, int16_t* piLength) NOEXCEPT = 0;};
```

### *5.3.1: virtual DMResult setOption(eCommandOption eCOption, int32_t lValue*

Set options for hostcommand level properties.

### *5.3.2: virtual DMResult getOption(eCommandOption eCOption, int32_t* plValue, int16_t iMaxLength, int16_t* piLength*

Get options for hostcommand level properties.

#### 5.3.2.1: eCommandOption enum values

```
COMMAND_OPT_UNKNOWN = 0,
COMMAND_OPT_GET_ALL_CONNECTED_DEVICES = 1,
COMMAND_OPT_REMOVE_ALL_CONNECTED_DEVICES = 2
```

## 5.4: IHostCommandQuery Interface

Use the *IHostCommandQuery* interface to obtain an interface from the *IHostCommand* hierarchy. It is used because `dynamic_cast<>` does not work across DLL (.so) boundaries.

```
INTF IHostCommandQuery
{
    virtual bool query(IHostCommand **p) noexcept = 0;
    virtual bool query(IHostCommandExt **p) noexcept = 0;
    virtual bool query(IDeviceSettings  **p) noexcept = 0;
    virtual bool query(IAdvanceSettings **p) noexcept = 0;
    virtual bool query(IHostCommandOption **p) NOEXCEPT = 0;
    };
```

### *5.4.1: bool query(IHostCommand **p)*

This method obtains an *IHostCommand* interface.

### *5.4.2: virtual bool query(IHostCommandExt **p)*

This method obtains an *IHostCommandExt* interface.

### *5.4.3: bool query(IDeviceSettings  **p)*

This method obtains an *IDeviceSettings* interface.

### *5.4.4: bool query(IAdvanceSettings **p)*

This method obtains an *IAdvanceSettings* interface.

### *5.4.5: bool query(IHostCommandOption **p)*

This method obtains an *IHostCommandOption* interface.

## 5.5: IAdvanceSettings Interface

Use the *IAdvanceSettings* interface to set and get various "advanced" (less common) device option settings.

```
INTF IAdvanceSettings{
    virtual DMResult getDialTone(bool& value) const noexcept = 0;
    virtual DMResult setDialTone(bool value) noexcept = 0;
    virtual DMResult getAudioSensing(bool& value)
        const noexcept = 0;
    virtual DMResult setAudioSensing(bool value) noexcept = 0;
    virtual DMResult getDialToneActive(bool& value)
        const noexcept = 0;
    virtual DMResult getAnswerOnDon(bool& bEnable) const noexcept = 0;
    virtual DMResult setAnswerOnDon(bool value) noexcept = 0;

    virtual DMResult getAutoTransfer(eSensorControl&
        value) const noexcept = 0;
    virtual DMResult setAutoTransfer(eSensorControl value) = noexcept 0;
    virtual DMResult getAutoDisconnect(eSensorControl& value)
        const noexcept = 0;
    virtual DMResult setAutoDisconnect(eSensorControl value) noexcept = 0;
    virtual DMResult getAutoReject(eSensorControl&
        value) const noexcept = 0;
    virtual DMResult setAutoReject(eSensorControl value) noexcept = 0;
    virtual DMResult getLockHookswitch(eSensorControl& value)
        const noexcept = 0;
    virtual DMResult setLockHookswitch(eSensorControl value) noexcept = 0;
    virtual DMResult getAutoPause(eSensorControl& value) const noexcept = 0;
    virtual DMResult setAutoPause(eSensorControl value) noexcept = 0;
    virtual DMResult getVoicePrompt(eSensorControl&
        value) const noexcept  = 0;
    virtual DMResult setVoicePrompt(eSensorControl value) noexcept = 0;
    };
```

### *5.5.1: DMResult getDialTone(bool& value)*

This method retrieves the value of the dial tone setting. If this feature is enabled, when a computer system radio link is established, it is assumed the user is trying to place a VoIP call and a dial tone is provided.

See Also: Section 4.1.1.1: DMResult enum.

### *5.5.2: DMResult setDialTone(bool value)*

This method enables or disables the dial tone when a computer system radio link is established.

See Also: Section 4.1.1.1: DMResult enum.

### 5.5.3: DMResult getAudioSensing(bool& value)

This method finds out if audio sensing is enabled. Audio Sensing is a special feature supported by Plantronics wireless devices that can detect an audio signal at the USB port and automatically establish a computer system radio link between the USB adapter and wireless headset without the user having to press the computer system call control button.

See Also: Section 4.1.1.1: DMResult enum.

### 5.5.4: DMResult setAudioSensing(bool value)

This method enables or disables audio sensing. Audio Sensing is a special feature supported by Plantronics wireless devices that can detect an audio signal at the USB port and automatically establish a computer system radio link between the USB adapter and wireless headset without the user having to press the computer system call control button.

See Also: Section 4.1.1.1: DMResult enum.

### 5.5.5: DMResult getDialToneActive(bool& value)

The method finds out if the dial tone is enabled when the device is active.

See Also: Section 4.1.1.1: DMResult enum.

### 5.5.6: DMResult getAnswerOnDon(bool& bEnable)

This method finds out if the call will be automatically answered when the headset is placed on the user's ear.

See Also: Section 4.1.1.1: DMResult enum.

### 5.5.7: DMResult setAnswerOnDon(bool value)

This method specifies whether or not a call should be automatically answered when a user puts on their headset.

See Also: Section 4.1.1.1: DMResult enum.

### 5.5.8: DMResult getAutoTransfer(eSensorControl& value)

This method finds out if auto transfer is enabled, Auto transfer enables calls to be automatically transferred between a mobile phone and a headset.

See Also: Section 4.1.1.1: DMResult enum. Options for the *eSensorControl* enum are listed below.

#### 5.5.8.1: eSensorControl enum values

```
SENSOR_CONTROL_HEADSET_NOT_CONNECTED = 0
SENSOR_CONTROL_UNDEFINED = 1
SENSOR_CONTROL_DISABLED = 2
SENSOR_CONTROL_ENABLED = 3
```

### 5.5.9: DMResult setAutoTransfer(eSensorControl value)

This method sets the auto transfer value.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed above.

### *5.5.10: DMResult getAutoDisconnect(eSensorControl& value)*

This method finds out if the connections will automatically end when the headset is removed from the ear.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.11: DMResult setAutoDisconnect(eSensorControl value)*

This method sets the value for the auto-disconnect option, determining whether or not the connection will end when the headset is removed from the ear.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.12: DMResult getAutoReject(eSensorControl& value)*

This method discovers whether or not the auto reject feature is enabled.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.13: DMResult setAutoReject(eSensorControl value)*

This method sets the auto reject feature to a specified value.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.14: DMResult getLockHookswitch(eSensorControl& value)*

This method discovers whether or not the electronic hookswitch is locked. The electronic hookswitch cable electronically and automatically takes the desk phone handset off hook and enables remote call control with a headset.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.15: DMResult setLockHookswitch(eSensorControl value)*

This method sets the lock hookswitch to a specific value.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.16: DMResult getAutoPause(eSensorControl& value)*

This method discovers whether the device will automatically pause streaming music when a call is received.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.17: DMResult setAutoPause(eSensorControl value)*

This method specifies whether music is automatically paused when an incoming call is received.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.18: DMResult getVoicePrompt(eSensorControl& value)*

The method discovers whether the device will respond to a voice command.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

### *5.5.19: DMResult setVoicePrompt(eSensorControl value)*

This method sets the value for whether or not a device will respond to a voice command.

See Also: Section 4.1.1.1: DMResult enum. The *eSensorControl* enum values are listed on page 63.

## 5.6: IATDCommand Interface

Inherit the *IATDCommand* interface to dial a call over a modem.

```
INTF IATDCommand{
    virtual DMResult AnswerMobileCall() noexcept = 0;
    virtual DMResult EndMobileCall() noexcept = 0;
    virtual DMResult Redial() = 0;
    virtual DMResult MakeMobileCall(wchar_t const *callerId,
        int32_t length) noexcept = 0;
    virtual DMResult GetMobileCallStatus() noexcept = 0;
    virtual wchar_t const *getCallerId() noexcept = 0;
    virtual DMResult MuteMobileCall(bool mute) noexcept = 0;
    };
```

### *5.6.1: DMResult AnswerMobileCall()*

This method tells the Device Manager to answer a mobile call.

See Also: Section 4.1.1.1: DMResult enum.

### *5.6.2: DMResult EndMobileCall()*

This method tells the Device Manager to end a mobile call.

See Also: Section 4.1.1.1: DMResult enum.

### *5.6.3: DMResult Redial()*

This method tells the Device Manager to redial the last number.

See Also: Section 4.1.1.1: DMResult enum.

### *5.6.4: DMResult MakeMobileCall(wchar_t const *callerId, int32_t length)*

This method tells the Device Manager to make a mobile call with call ID enabled.

See Also: Section 4.1.1.1: DMResult enum.

### *5.6.5: DMResult GetMobileCallStatus()*

This method tells the Device Manager to get the status of the mobile call.

See Also: Section 4.1.1.1: DMResult enum.

### *5.6.6: wchar_t const *getCallerId()*

This method gets the ID of the calling device.

### *5.6.7: DMResult MuteMobileCall(bool mute)*

This method mutes or unmutes the mobile call.

See Also: Section 4.1.1.1: DMResult enum.

## 5.7: IDeviceSettings Interface

The *IDeviceSettings* interface is used to read and set device configuration parameters.

```
INTF IDeviceSettings {
    virtual DMResult getVOIPRing(eDeviceRingTone &tone) const
        noexcept = 0;
    virtual DMResult setVOIPRing(eDeviceRingTone value) noexcept = 0;
    virtual DMResult getVOIPBandwidth(eAudioBandwidth &audio)
        const noexcept = 0;
    virtual DMResult setVOIPBandwidth(eAudioBandwidth value) noexcept = 0;
    virtual DMResult getVOIPRingVolume(eVolumeLevel &level)
        const noexcept = 0;
    virtual DMResult setVOIPRingVolume(eVolumeLevel value) noexcept = 0;
    virtual DMResult getVOIPToneControl(eToneLevel &level)
        const noexcept = 0;
    virtual DMResult setVOIPToneControl(eToneLevel value) noexcept = 0;

    virtual DMResult getPSTNRing(eDeviceRingTone& value) const noexcept = 0;
    virtual DMResult setPSTNRing(eDeviceRingTone value) noexcept = 0;
    virtual DMResult getPSTNBandwidth(eAudioBandwidth& value)
        const noexcept = 0;
    virtual DMResult setPSTNBandwidth(eAudioBandwidth value) noexcept = 0;
    virtual DMResult getPSTNRingVolume(eVolumeLevel& value)
        const noexcept = 0;
    virtual DMResult setPSTNRingVolume(eVolumeLevel value) noexcept = 0;
    virtual DMResult getPSTNToneControl(eToneLevel &level)
        const noexcept = 0;
    virtual DMResult setPSTNToneControl(eToneLevel value) noexcept = 0;

    virtual DMResult getMobileRing(eDeviceRingTone &tone)
        const noexcept = 0;
    virtual DMResult setMobileRing(eDeviceRingTone value) noexcept = 0;
```

```
virtual DMResult getMobileRingVolume(eVolumeLevel &level)
    const noexcept = 0;
virtual DMResult setMobileRingVolume(eVolumeLevel value) noexcept = 0;
virtual DMResult getMobileBandwidth(eAudioBandwidth &audio)
    const noexcept
virtual DMResult setMobileBandwidth(eAudioBandwidth value) noexcept = 0;
virtual DMResult getMuteTone(eDeviceRingTone &tone)
    const noexcept = 0;
virtual DMResult setMuteTone(eDeviceRingTone value) noexcept = 0;

virtual DMResult getToneVolume(eVolumeLevel &level)
    const noexcept = 0;
virtual DMResult setToneVolume(eVolumeLevel value) noexcept = 0;
virtual DMResult getMuteToneVolume(eVolumeLevel &level)
    const noexcept = 0;
virtual DMResult setMuteToneVolume(eVolumeLevel value) noexcept = 0;
virtual DMResult getActiveCallRing(eActiveCallRing &ring)
    const noexcept = 0;
virtual DMResult setActiveCallRing(eActiveCallRing value) noexcept = 0;
virtual DMResult getAntiStartleEnabled(bool& value) const noexcept = 0;
virtual DMResult setAntiStartleEnabled(bool value) noexcept = 0;

virtual DMResult getAudioLimit(eAudioLimit& value) const noexcept = 0;
virtual DMResult setAudioLimit(eAudioLimit value) noexcept = 0;
virtual DMResult getG616Enabled(bool& value) const noexcept = 0;
virtual DMResult setG616Enabled(bool value) noexcept = 0;
virtual DMResult getOTAEnabled(bool &enabled) const noexcept =
    0;
virtual DMResult setOTAEnabled(bool value) noexcept = 0;
virtual DMResult getIntellistandEnabled(bool &enabled)
    const noexcept = 0;
virtual DMResult setIntellistandEnabled(bool value) noexcept = 0;

virtual DMResult getPowerMode(ePowerLevel &level) const
    noexcept = 0;
virtual DMResult setPowerMode(ePowerLevel value) noexcept = 0;
virtual DMResult getData(eCustomData dataType) noexcept = 0;
virtual DMResult setData(eCustomData dataType, uint8_t* buffer,
    int32_t bufferLength) noexcept = 0;
virtual DMResult isPasswordProtected(bool &protec) const
    noexcept = 0;
virtual DMResult getDECTMode(bool &mode) const noexcept = 0;
virtual DMResult getTWAPeriod(eTWAPeriod &period) const
    noexcept = 0;
virtual DMResult setTWAPeriod(eTWAPeriod value) noexcept = 0;
virtual DMResult getPhoneLine(eLineType &type) const
    noexcept = 0;
virtual DMResult setPhoneLine(eLineType value) noexcept = 0;
```

```
    virtual DMResult getBTInterfaceEnabled(bool &enabled)
        const noexcept = 0;
    virtual DMResult setBTInterfaceEnabled(bool value) noexcept = 0;
    virtual DMResult getBTAutoConnectEnabled(bool &enabled)
        const noexcept = 0;
    virtual DMResult setBTAutoConnectEnabled(bool value) noexcept = 0;
    virtual DMResult restoreDefaultSettings() noexcept = 0;
    virtual DMResult getACLPollingEnabled(bool &enabled)
        const noexcept = 0;
    virtual DMResult setACLPollingEnabled(bool value) noexcept = 0;
    virtual DMResult getBTVoiceCommandEnabled(bool &enabled)
        const noexcept = 0;
    virtual DMResult setBTVoiceCommandEnabled(bool value) noexcept = 0;
    virtual DMResult getVolumeControlOrientation(eVolumeControlOrientation&
        value) const noexcept = 0;
    virtual DMResult setVolumeControlOrientation(eVolumeControlOrientation
        value) noexcept = 0;
    virtual DMResult getA2DPEnabled(bool& value) const noexcept = 0;
    virtual DMResult setA2DPEnabled(bool value) noexcept = 0;
};
```

## *5.7.1: DMResult getVOIPRing(eDeviceRingTone &tone) const noexcept*

This method gets the Voice Over Internet Protocol (VOIP) ring tone from the *eDeviceRingTone* enum *(below)*. When a VoIP call arrives, the handset will use a distinctive VoIP ring.

*DMResult* values are listed on page 40.

### 5.7.1.1: eDeviceRingTone enum

```
DEVICE_RING_TONE_TYPE_1 = 0
DEVICE_RING_TONE_TYPE_2 = 1
DEVICE_RING_TONE_TYPE_3 = 2
DEVICE_RING_TONE_OFF = 3
```

## *5.7.2: DMResult setVOIPRing(eDeviceRingTone value)*

This method sets the VOIP ring tone to a specified value. When a VoIP call arrives, the handset will use a distinctive VoIP
See Also: Section 4.1.1.1: DMResult enum. The *eDeviceRingTone* enum values are above.

## *5.7.3: DMResult getVOIPBandwidth(eAudioBandwidth &audio) const noexcept*

This method gets the VOIP bandwidth of the device. The system will support both narrowband and wideband audio from the computer system. Wideband audio delivers heightened speech clarity and life-like fidelity. However, wideband audio consumes more battery power and has more stringent access criteria which reduces the number of systems that can be deployed in a small area.

See Also: Section 4.1.1.1: DMResult enum. Options for *eAudioBandwidth* enum are listed below.

#### 5.7.3.1: eAudioBandwidth enum values

```
AUDIO_BANDWIDTH_UNKNOWN = 0
AUDIO_BANDWIDTH_NARROW_BAND = 1
AUDIO_BANDWIDTH_WIDE_BAND = 2
```

### *5.7.4: DMResult setVOIPBandwidth(eAudioBandwidth value)*

This method sets the VOIP bandwidth for the device to a specified value. The system will support both narrowband and wideband audio from the computer system. Wideband audio delivers heightened speech clarity and life-like fidelity. However, wideband audio consumes more battery power and has more stringent access criteria which reduces the number of systems that can be deployed in a small area.

See Also: Section 4.1.1.1: DMResult enum. *eAudioBandwidth* enum values are listed above.

### *5.7.5: DMResult getVOIPRingVolume(eVolumeLevel &level) const noexcept*

This method gets the VOIP ring volume.

See Also: Section 4.1.1.1: DMResult enum. Options for the *eVolumeLevel* enum are listed below.

#### 5.7.5.1: eVolumeLevel enum values

```
VOLUME_LEVEL_OFF = 0
VOLUME_LEVEL_LOW = 1
VOLUME_LEVEL_STANDARD = 2
```

### *5.7.6: DMResult setVOIPRingVolume(eVolumeLevel value)*

This method sets the VOIP ring volume to a specified value.

See Also: Section 4.1.1.1: DMResult enum. *eVolumeLevel* enum values are listed above.

### *5.7.7: DMResult getVOIPToneControl(eToneLevel &level) const noexcept*

This method gets the VOIP tone control from the *eToneLevel* enum (below).

See Also: Section 4.1.1.1: DMResult enum.

#### 5.7.7.1: eToneLevel enum values

```
TONE_LEVEL_MAX_BASS = 0
TONE_LEVEL_MID_BASS = 1
TONE_LEVEL_MIN_BASS = 2
TONE_LEVEL_NO_BOOST = 3
TONE_LEVEL_MIN_TREBLE = 4
TONE_LEVEL_MID_TREBLE = 5
TONE_LEVEL_MAX_TREBLE = 6
```

### *5.7.8: DMResult setVOIPToneControl(eToneLevel value)*

This method sets the VOIP tone control to a specified value.

See Also: Section 4.1.1.1: DMResult enum. The *eToneLevel* enum values are listed above.

## *5.7.9: DMResult getPSTNRing(eDeviceRingTone& value)*

This method gets the Public Switched Telephone Network (PSTN) ring.

Values for the *eDeviceRingTone* enum are listed below.

See Also: Section 4.1.1.1: DMResult enum.

### 5.7.9.1: eDeviceRingTone enum values

```
DEVICE_RING_TONE_TYPE_1 = 0
DEVICE_RING_TONE_TYPE_2 = 1
DEVICE_RING_TONE_TYPE_3 = 2
```

## *5.7.10: DMResult setPSTNRing(eDeviceRingTone value)*

This method sets the Public Switched Telephone Network (PSTN) ring for the device to a specified value.

See Also: Section 4.1.1.1: DMResult enum. The *eDeviceRingTone* enum values are listed on page 70.

## *5.7.11: DMResult getPSTNBandwidth(eAudioBandwidth& value)*

This method gets the Public Switched Telephone Network (PSTN) bandwidth.

See Also: Section 4.1.1.1: DMResult enum. The *eAudioBandwidth* enum values are listed on page 69.

## *5.7.12: DMResult setPSTNBandwidth(eAudioBandwidth value)*

This method sets the Public Switched Telephone Network (PSTN) bandwidth to a specified value.

See Also: Section 4.1.1.1: DMResult enum. The *eAudioBandwidth* enum values are listed on page 69.

## *5.7.13: DMResult getPSTNRingVolume(eVolumeLevel& value)*

This method gets the Public Switched Telephone Network (PSTN) ring volume.

See Also: Section 4.1.1.1: DMResult enum. The *eVolumeLevel* enum values are listed on page 69.

## *5.7.14: DMResult setPSTNRingVolume(eVolumeLevel value)*

This method sets the Public Switched Telephone Network (PSTN) ring volume to a specified value.

See Also: Section 4.1.1.1: DMResult enum. The *eVolumeLevel* enum values are listed on page 69.

## *5.7.15: DMResult getPSTNToneControl(eToneLevel &level) const noexcept*

This method gets the Public Switched Telephone Network (PSTN) tone control.

See Also: Section 4.1.1.1: DMResult enum. The *eToneLevel* enum values are listed on page 69.

### *5.7.16: DMResult setPSTNToneControl(eToneLevel value)*

This method sets the Public Switched Telephone Network (PSTN) tone control to a specified value.

See Also: Section 4.1.1.1: DMResult enum. The *eToneLevel* enum values are listed on page 69.

### *5.7.17: DMResult getMobileRing(eDeviceRingTone &tone) const noexcept*

This method gets the mobile ring tone.

See Also: Section 4.1.1.1: DMResult enum. The *eDeviceRingTone* enum values are listed on page 70.

### *5.7.18: DMResult setMobileRing(eDeviceRingTone value)*

This method sets the mobile ring tone.

See Also: Section 4.1.1.1: DMResult enum. The *eDeviceRingTone* enum values are listed on page 70.

### *5.7.19: DMResult getMobileRingVolume(eVolumeLevel &level) const noexcept*

This method gets the mobile ring volume.

See Also: Section 4.1.1.1: DMResult enum. The *eVolumeLevel* enum values are listed on page 69.

### *5.7.20: DMResult setMobileRingVolume(eVolumeLevel value)*

This method sets the mobile ring volume to a specified value.

See Also: Section 4.1.1.1: DMResult enum. The *eVolumeLevel* enum values are listed on page 69.

### *5.7.21: DMResult getMobileBandwidth(eAudioBandwidth &audio) const noexcept*

This method gets the mobile bandwidth.

See Also: Section 4.1.1.1: DMResult enum. The *eAudioBandwidth* enum values are listed on page 69.

### *5.7.22: DMResult setMobileBandwidth(eAudioBandwidth value)*

This method sets the mobile bandwidth to a specified value.

See Also: Section 4.1.1.1: DMResult enum. The *eAudioBandwidth* enum values are listed *on* page 69.

### *5.7.23: DMResult getMuteTone(eDeviceRingTone &tone) const noexcept*

This method gets the mute tone.

See Also: Section 4.1.1.1: DMResult enum. The *eDeviceRingTone* enum values are listed on page 70.

### *5.7.24: DMResult setMuteTone(eDeviceRingTone value)*

This method sets the mute tone of the device to a specific value.

See Also: Section 4.1.1.1: DMResult enum. The *eDeviceRingTone* enum values are listed on page 70.

### *5.7.25: DMResult getToneVolume(eVolumeLevel &level) const noexcept*

This method gets the volume level of the specified device.

See Also: Section 4.1.1.1: DMResult enum. The *eVolumeLevel* enum values are listed on page 69.

### *5.7.26: DMResult setToneVolume(eVolumeLevel value)*

This method sets the tone volume of the device to a specific value.

See Also: Section 4.1.1.1: DMResult enum. The *eVolumeLevel* enum values are listed on page 69.

### *5.7.27: DMResult getMuteToneVolume(eVolumeLevel &level) const noexcept*

This method gets the mute tone volume.

See Also: Section 4.1.1.1: DMResult enum. The *eVolumeLevel* enum values are listed on page 69.

### *5.7.28: DMResult setMuteToneVolume(eVolumeLevel value)*

This method sets the mute tone volume to a specific value.

See Also: Section 4.1.1.1: DMResult enum. The *eVolumeLevel* enum values are listed on page 69.

### *5.7.29: DMResult getActiveCallRing(eActiveCallRing &ring) const noexcept*

This method gets the setting for second incoming call tone in the presence of an Active Call.

See Also: Section 4.1.1.1: DMResult enum. Options for the *eActiveCallRing* enum are listed below.

#### 5.7.29.1: eActiveCallRing enum values

```
ACTIVE_CALL_RING_IGNORE = 0
ACTIVE_CALL_RING_RING_ONCE = 1
ACTIVE_CALL_RING_RING_CONTINUOUS = 2
```

### *5.7.30: DMResult setActiveCallRing(eActiveCallRing value)*

This method gets the setting for second incoming call tone in the presence of an Active Call to a specific *eActiveCallRing* enum value.

See Also: Section 4.1.1.1: DMResult enum. The *eActiveCallRing* enum values are listed on page 72.

### *5.7.31: DMResult getAntiStartleEnabled(bool& value)*

This method gets whether or not the anti-startle feature is enabled. Anti-startle provides advanced hearing protection against sudden loud sounds. When Anti-startle is enabled, the system identifies and eliminates sudden loud sounds and rapidly reduces them to a comfortable level. When Anti-startle is disabled, the headset limits sound levels at 118 dBA to protect your hearing.

See Also: Section 4.1.1.1: DMResult enum.

## *5.7.32: DMResult setAntiStartleEnabled(bool value)*

This method sets the anti-startle feature. Anti-startle provides advanced hearing protection against sudden loud sounds. When anti-startle is enabled, the system identifies and eliminates sudden loud sounds and rapidly reduces them to a comfortable level. When anti-startle is disabled, the headset limits sound levels at 118 dBA to protect your hearing.

See Also: Section 4.1.1.1: DMResult enum.

## *5.7.33: DMResult getAudioLimit(eAudioLimit& value)*

This method retrieves the current setting for audio limit. Audio Limiting provides advanced hearing protection for daily noise exposure. When Audio-Limiting is enabled, the system monitors and controls sound to ensure audio levels do not exceed 80 dBA or 85 dBA (whichever is selected) specified by current and imminent EU legislation. When Audio-Limiting is disabled, the headset limits sound levels at 118 dBA to protect your hearing.

Options for the *eAudioLimit* enum are listed below.

See Also: Section 4.1.1.1: DMResult enum.

### 5.7.33.1: eAudioLimit enum values

```
AUDIO_LIMIT_OFF = 0
AUDIO_LIMIT_EIGHTY_FIVE = 1
AUDIO_LIMIT_EIGHTY = 2
```

## *5.7.34: DMResult setAudioLimit(eAudioLimit value)*

This method sets the audio limit to a specific value. Audio Limiting provides advanced hearing protection for daily noise exposure. When Audio-Limiting is enabled, the system monitors and controls sound to ensure audio levels do not exceed 80 dBA or 85 dBA (whichever is selected) specified by current and imminent EU legislation. When Audio-Limiting is disabled, the headset limits sound levels at 118 dBA to protect your hearing.

See Also: Section 4.1.1.1: DMResult enum. The *eAudioLimit* enum values are listed above.

## *5.7.35: DMResult getG616Enabled(bool& value)*

This method finds out if the device has G616 enabled. G616 Acoustic Limiting provides additional hearing protection against acoustic shock. When G616 Acoustic Limiting is enabled, the system provides additional acoustic shock protection. Sound levels are limited to 102 dBA as recommended in the G616:2006 guideline issued by the Australian Communications Industry Forum (ACIF). When G616 Audio-Limiting is disabled, the headset limits sound levels at 118 dBA to protect your hearing.

See Also: Section 4.1.1.1: DMResult enum.

## *5.7.36: DMResult setG616Enabled(bool value)*

This method enables G616 on the device. G616 Acoustic Limiting provides additional hearing protection against acoustic shock. When G616 Acoustic Limiting is enabled, the system provides additional acoustic shock protection. Sound levels are limited to 102 dBA as recommended in the G616:2006 guideline issued by the Australian Communications Industry Forum (ACIF). When G616 Audio-Limiting is disabled, the headset limits sound levels at 118 dBA to protect your hearing.

See Also: Section 4.1.1.1: DMResult enum.

## *5.7.37: DMResult getOTAEnabled(bool &enabled)*

This method finds out if OTA is enabled on this device. OTA is the ability to update phone firmware over the network.

See Also: Section 4.1.1.1: DMResult enum.

## *5.7.38: DMResult setOTAEnabled(bool value)*

This method sets OTA on this device. OTA is the ability to update phone firmware over the network.

See Also: Section 4.1.1.1: DMResult enum.

## *5.7.39: DMResult getIntellistandEnabled(bool &enabled)*

This method finds out if Intellistand is enabled on this device. When Intellistand is disabled, the user is in control of picking up an incoming call or triggering a dial tone in the headset by pressing the headset control button. When Intellistand is enabled, the headset transfersthe dial tone or the incoming call transferred to the headset automatically when the headset is removed from the charging cradle.

See Also: Section 4.1.1.1: DMResult enum.

## *5.7.40: DMResult setIntellistandEnabled(bool value)*

This method enables or disables Intellistand functionality. When Intellistand is disabled, the user is in control of picking up an incoming call or triggering a dial tone in the headset by pressing the headset control button. When Intellistand is enabled, the headset transfersthe dial tone or the incoming call transferred to the headset automatically when the headset is removed from the charging cradle.

See Also: Section 4.1.1.1: DMResult enum.

## *5.7.41: DMResult getPowerMode(ePowerLevel &level) const noexcept*

This method finds out the setting for the power mode of the device.

See Also: Section 4.1.1.1: DMResult enum. Options for the *ePowerLevel* enum are listed below.

### 5.7.41.1: ePowerLevel enum values

```
POWER_LEVEL_FIXED_LOW = 0
POWER_LEVEL_ADAPTIVE_MEDIUM = 1
POWER_LEVEL_ADAPTIVE_HIGH = 2
```

## *5.7.42: DMResult setPowerMode(ePowerLevel value)*

This method sets the power mode on the device to a specified value. A lower power level will extend the battery life of the device.

See Also: Section 4.1.1.1: DMResult enum. The *ePowerLevel* enum values are listed above.

## *5.7.43: DMResult getData(eCustomData dataType)*

This method gets custom data from the device.

Options for the *eCustomData* enum are listed below.

See Also: Section 4.1.1.1: DMResult enum.

#### 5.7.43.1: eCustomData enum values

```
CUSTOM_DATA_PASSWORD = 1
CUSTOM_DATA_FEATURE_LOCK = 2
CUSTOM_DATA_RAW = 3
CUSTOM_DATA_RAW_INPUT = 4
CUSTOM_DATA_RAW_OUTPUT = 5
CUSTOM_DATA_COMMUNICATION_TEST = 6
CUSTOM_DATA_HEADSET_SERIAL_NUMBER = 7
CUSTOM_DATA_BASE_SERIAL_NUMBER = 8
CUSTOM_DATA_RAW_HID_PIPE_DATA = 9
CUSTOM_DATA_BR_INPUT_DATA = 10
CUSTOM_DATA_BR_OUTPUT_DATA = 11
CUSTOM_DATA_HEADSET_PRODUCTION_SERIAL_NUMBER = 12
CUSTOM_DATA_BASE_PRODUCTION_SERIAL_NUMBER = 13
CUSTOM_DATA_HEADSET_PRODUCTION_BUILD_NUMBER = 14
CUSTOM_DATA_BASE_PRODUCTION_BUILD_NUMBER = 15
CUSTOM_DATA_HEADSET_PRODUCTION_PART_NUMBER = 16
CUSTOM_DATA_BASE_PRODUCTION_PART_NUMBER = 17
```

### *5.7.44: DMResult setData(eCustomData dataType, uint8_t* buffer, int32_t bufferLength)*

This methods sets custom data on the device.

See Also: Section 4.1.1.1: DMResult enum. The *eCustomData* enum values are listed above.

### *5.7.45: DMResult isPasswordProtected(bool &protec) const noexcept*

This method finds out if the device is protected by a password.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.46: DMResult getDECTMode(bool &mode) const noexcept*

This method finds out if DECT mode is enabled on the device. The Digital Enhanced Cordless Telecommunications (DECT) standard fully specifies a means for a portable unit, such as a cordless telephone, to access a fixed telecoms network via radio.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.47: DMResult getTWAPeriod(eTWAPeriod &period) const noexcept*

This method gets the TWA period setting for the device. Time Weighted Average(TWA) is the guideline used by the Occupational Safety and Health Administration to measure noise levels in the workplace. Exposure to loud noise over a long time can cause hearing damage. If the TWA noise level, which is the average sound level over 8 hours, is exceeding 85 dB(A), a hearing conservation program is required. The Time Weighted Average (TWA) measurement prevents average daily sound exposure from exceeding 85 dBA.

See Also: *DMResult* values are listed on page 40. Options for the *eTWAPeriod* enum are listed below.

#### 5.7.47.1: eTWAPeriod enum

```
TWA_PERIOD_UNKNOWN = 0
TWA_PERIOD_TWO_HOURS = 1
TWA_PERIOD_FOUR_HOURS = 2
TWA_PERIOD_SIX_HOURS = 3
TWA_PERIOD_EIGHT_HOURS = 4
```

### *5.7.48: DMResult setTWAPeriod(eTWAPeriod value)*

This method sets the TWA period for the device. The Time Weighted Average (TWA) measurement prevents average daily sound exposure from exceeding 85 dBA.

See Also: Section 4.1.1.1: DMResult enum. The *eTWAPeriod* enum values are listed above.

### *5.7.49: DMResult getPhoneLine(eLineType &type) const noexcept*

This method detects the type of phone line the device is set to use by default. When a user makes an outgoing call, the call is made by default using the last line selected. If the last line used is unavailable, outgoing calls are made over the first available line, in this order: softphone, landline, or mobile phone.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 57.

### *5.7.50: DMResult setPhoneLine(eLineType value)*

This method sets the phone line type to a specified value, such as softphone, landline, or mobile.

See Also: Section 4.1.1.1: DMResult enum. *eLineType* enum values are listed on page 57.

### *5.7.51: DMResult getBTInterfaceEnabled(bool &enabled)*

This method finds out which Bluetooth interface is enabled.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.52: DMResult setBTInterfaceEnabled(bool value)*

This method sets the Bluetooth interface to a specified value.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.53: DMResult getBTAutoConnectEnabled(bool &enabled)*

This method finds out if Bluetooth is automatically connected.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.54: DMResult setBTAutoConnectEnabled(bool value)*

This method sets whether or not a Bluetooth device is automatically connected.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.55: DMResult restoreDefaultSettings()*

This method restores the device's default settings.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.56: DMResult getACLPollingEnabled(bool &enabled)*

This method finds out if ACL polling is enabled. ACL (Asynchronous Connectionless Link) is a control data link. SCO (Synchronous Connection Oriented Link) is a voice data link.

While Bluetooth is the technology used to connect devices from different manufacturers without wires, the devices still have to talk to each other while providing whatever service they are designed for. Headsets are audio devices for speech, talking and listening; but when a headset is not doing its job, it is still communicating with the other device referred to as the audio gateway (AG). This would be a cell phone or USB dongle for instance.

Once the headset has completed its pairing to the AG, it goes into a standby mode of sorts. As long as the headset and AG are within range they are connected by the ACL. The headset and AG are checking in with each other to verify range, check for control signals like incoming calls, redial or voice control requests.

If the AG receives a call and the headset ACL is good the AG will signal the headset. The headset will beep to let the user know a call has arrived. Pressing the call control on the headset lets the AG know it is okay to open the SCO and voice data will be exchanged between the headset and AG.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.57: DMResult setACLPollingEnabled(bool value)*

This method enables or disables ACL polling.

ACL (Asynchronous Connectionless Link) is a control data link, and , and SCO (Synchronous Connection Oriented Link) is a voice data link.

While Bluetooth is the technology used to connect devices from different manufacturers without wires, the devices still have to talk to each other while providing whatever service they are designed for. Headsets are audio devices for speech, talking and listening; but when a headset is not doing its job, it is still communicating with the other device referred to as the audio gateway (AG). This would be a cell phone or USB dongle for instance.

Once the headset has completed its pairing to the AG, it goes into a standby mode of sorts. As long as the headset and AG are within range they are connected by the ACL. The headset and AG are checking in with each other to verify range, check for control signals like incoming calls, redial or voice control requests.

If the AG receives a call and the headset ACL is good the AG will signal the headset. The headset will beep to let the user know a call has arrived. Pressing the call control on the headset lets the AG know it is okay to open the SCO and voice data will be exchanged between the headset and AG.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.58: DMResult getBTVoiceCommandEnabled(bool &enabled) const*

This methods finds out if Bluetooth voice command is enabled. If voice commands are enabled, the user can give the headset a verbal command, and it will whisper the answer or do what you say.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.59: DMResult setBTVoiceCommandEnabled(bool value)*

This method enables or disables Bluetooth voice command. If voice commands are enabled, the user can give the headset a verbal command, and it will whisper the answer or do what you say.

See Also: Section 4.1.1.1: DMResult enum.

### *5.7.60: DMResult getVolumeControlOrientation(eVolumeControlOrientation& value)*

This method gets the volume control orientation.

See Also: Section 4.1.1.1: DMResult enum. *eVolumeControlOrientation* enum values are listed below.

#### 5.7.60.1: eVolumeControlOrientation enum values

```
VOLUME_CONTROL_ORIENTATION_UNKNOWN = 0
VOLUME_CONTROL_ORIENTATION_RIGHT = 1
VOLUME_CONTROL_ORIENTATION_LEFT = 2
```

### *5.7.61: DMResult setVolumeControlOrientation(eVolumeControlOrientation value)*

This method sets the volume control orientation, whether the volume control is on the left, right, or whether the orientation is unknown.

See Also: *DMResult* values are listed on page 40. *eVolumeControlOrientation* enum values are listed above.

### *5.7.62: DMResult getA2DPEnabled(bool& value)*

This method finds out if A2DP is enabled. The Advanced Audio Distribution Profile (A2DP) is a Bluetooth profile that allows for the wireless transmission of stereo audio from an A2DP source (typically a phone or computer) to an A2DP receiver (a set of Bluetooth headphones or stereo system).

See Also: *DMResult* values are listed on page 40.

### *5.7.63: DMResult setA2DPEnabled(bool value)*

This method enables or disables A2DP. The Advanced Audio Distribution Profile (A2DP) is a Bluetooth profile that allows for the wireless transmission of stereo audio from an A2DP source (typically a phone or computer) to an A2DP receiver (a set of Bluetooth headphones or stereo system).

See Also: *DMResult* values are listed on page 40.

## 5.8: IDeviceSettingsExt Interface

Use this interface to capture all HID 2.0 device settings that are not already exposed through *IDeviceSettings* and *IAdvanceSettings*.

```
INTF IDeviceSettingsExt
{
    //Get and set Acoustics Reporting for AAL Acoustics incident and TWA reporting and
```

```
  Conversation dynamics reporting
virtual DMResult getAcousticsReporting(eAcousticsReportType type, bool& value) const
    NOEXCEPT = 0;
virtual DMResult setAcousticsReporting(eAcousticsReportType type, bool value)
    NOEXCEPT = 0;
//Get and set AAL Acoustics incident Threshold
virtual DMResult getAALReportingThreshold(AALReportingThreshold& value) const
    NOEXCEPT = 0;
virtual DMResult setAALReportingThreshold(AALReportingThreshold value) NOEXCEPT = 0;
//Get and set AAL TWA reporting time period
virtual DMResult getTWAReportingPeriod(uint32_t& value) const NOEXCEPT = 0;
virtual DMResult setTWAReportingPeriod(uint32_t value) NOEXCEPT = 0;
//Get and set AAL TWA reporting time period
virtual DMResult getConversationDynamicsPeriod(uint32_t& value) const NOEXCEPT = 0;
virtual DMResult setConversationDynamicsPeriod(uint32_t value) NOEXCEPT = 0;
virtual DMResult getPartitionInfo(uint16_t & partition, uint16_t & position,
    uint16_t & version, uint16_t & number) = 0;
virtual DMResult removePartition(const uint16_t&) = 0;
virtual DMResult setLanguage(uint16_t lang) = 0;
virtual DMResult getLanguage(uint16_t & lang) = 0;
virtual DMResult getAnswerIgnoreVoiceCommand(bool & on) = 0;
virtual DMResult setAnswerIgnoreVoiceCommand(bool on) = 0;
virtual DMResult getCallerNameAnnouncement(bool & on) = 0;
virtual DMResult setCallerNameAnnouncement(bool on) = 0;
virtual DMResult getMuteOffVoicePrompt(bool & on) = 0;
virtual DMResult setMuteOffVoicePrompt(bool on) = 0;
virtual DMResult getMutePromptInterval(uint16_t & interval) = 0;
virtual DMResult setMutePromptInterval(uint16_t interval) = 0;
virtual DMResult getCallButtonLock(bool & on) = 0;
virtual DMResult setCallButtonLock(bool on) = 0;
virtual DMResult getSupportedLanguages(uint16_t *, unsigned int & capacity) = 0;
virtual DMResult setSignalStrengthEvents(SignalStrengthParams param) = 0;
virtual DMResult getCurrentSignalStrengthEvents(uint8_t id, uint8_t& conId, uint8_t&
    strength, SignalStrengthType& type) = 0;
virtual DMResult setPairingMode(bool enable) = 0;
virtual DMResult getPairingMode(bool& enable) = 0;
virtual DMResult getConnectionStatus(std::vector<uint8_t>& downPort,
    std::vector<uint8_t>& connectedPort, uint8_t& origPort) = 0;
virtual DMResult getSignalStrengthConfiguration(uint8_t connId, SignalStrengthParams&
     param) = 0;
virtual DMResult getAudioStatus(uint8_t& codec, uint8_t& port, uint8_t& speakerGain,
     uint8_t& micGain) = 0;
virtual DMResult getSCOTone(bool & tone) = 0;
virtual DMResult setSCOTone(bool tone) = 0;
virtual DMResult setFindHeadsetLEDAlert(bool enable) = 0;
virtual DMResult getFindHeadsetLEDAlert(bool& enabled) = 0;
virtual DMResult setAudioTransmitGain(uint8_t gain) = 0;
virtual DMResult setSpeakerVolume(bool action, uint16_t volume) = 0;
```

```
    virtual DMResult getWearingSensorEnabled(bool& enabled) = 0;
    virtual DMResult setWearingSensorEnabled(bool enabled) = 0;
    virtual DMResult setVRCallRejectAnswer(bool enable) = 0;
    virtual DMResult getVRCallRejectAnswer(bool& enabled) = 0;
    virtual DMResult getHeadsetConnectedState(bool& enabled) const NOEXCEPT = 0;
    virtual DMResult getHTopSelector(eHTopSelectorType& value) const NOEXCEPT = 0;
    virtual DMResult setHTopSelector(eHTopSelectorType value) NOEXCEPT = 0;
    virtual DMResult setAutoMuteCall(bool enabled) NOEXCEPT = 0;
    virtual DMResult getAutoMuteCall(bool& enabled) const NOEXCEPT = 0;
    virtual DMResult setMuteAlert(eMuteAlertValues value) NOEXCEPT = 0;
    virtual DMResult getMuteAlert(eMuteAlertValues &value) const NOEXCEPT = 0;
    virtual DMResult getVoiceSilenceDetectionMode(eVoiceSilenceDetectionMode& mode)
        const NOEXCEPT = 0;
    virtual DMResult setOLIFeature(bool enabled) NOEXCEPT = 0;
    virtual DMResult getOLIFeature(bool& enabled) const NOEXCEPT = 0;
};
```

### *5.8.1: DMResult getMutePromptInterval(uint16_t & interval)*

For product DA80, this command returns the current mute reminder interval (in seconds, 1 through 15 minutes) set on the device.

### *5.8.2: DMResult setMutePromptInterval(uint16_t interval)*

For product DA80, this “Get“ function returns the value of the last “Set“ operation.

### *5.8.3: DMResult getHTopSelector(eHTopSelectorType& value)*

For products DA80/DA70, the command returns the current setting for an H-Top Selector set on the device.

### *5.8.4: DMResult setHTopSelector(eHTopSelectorType value)*

For products DA80/DA70, this “Get“ function returns the value of the last “Set“ operation.

### *5.8.5: DMResult getMuteAlert(eMuteAlertValues &value)*

For product DA80, this command always returns MUTE_ALERT_VALUE_TIME_INTERVAL_REMINDER.

### *5.8.6: DMResult getHeadsetConnectedState(bool& enabled)*

For product DA80, this command returns *true* when top is connected, or *false* when disconnected.

# Chapter 6: Using Device Events Interfaces

Use Device Events to subscribe to and receive notifications from devices.

## 6.1: IDeviceEvents Interface

The *IDeviceEvents* interface provides functions to register for notifications using *IDeviceEventsCallback* and the function to receive and parse/handle the report received from the device.

```
INTF IDeviceEvents{
    virtual IDeviceEventsQuery *getQuery() noexcept = 0;
    virtual DMResult regCbDeviceEvents(IDeviceEventsCallback*) noexcept = 0;
    virtual DMResult unregCbDeviceEvents(IDeviceEventsCallback*) noexcept =
        0;
    virtual DMResult setInputReport(uint8_t const * reportBuffer,
        int32_t length) noexcept = 0;
    };
```

### *6.1.1: IDeviceEventsQuery *getQuery()*

This method returns the query pointer.

### *6.1.2: DMResult regCbDeviceEvents(IDeviceEventsCallback*)*

This method registers the device events receiver.

See Also: Section 4.1.1.1: DMResult enum.

### *6.1.3: DMResult unregCbDeviceEvents(IDeviceEventsCallback*)*

This method unregisters the device events receiver.

See Also: Section 4.1.1.1: DMResult enum.

### *6.1.4: DMResult setInputReport(uint8_t const * reportBuffer, int32_t length)*

This method parses the input report received from the device. Receives an array of bytes of the specified length and returns DM_RESULT_SUCCESS on success, DM_RESULT_FAIL otherwise.

See Also: Section 4.1.1.1: DMResult enum.

## 6.2: IDeviceEventsCallback Interface

Inherit the *IDeviceEventsCallback* interface to receive Device Events and to register the Device Events using *IDeviceEvents*.

```
INTF IDeviceEventsCallback{
    virtual void onTalkButtonPressed(DeviceEventArgs const& args) = 0;
    virtual void onButtonPressed(DeviceEventArgs const& args) = 0;
    virtual void onMuteStateChanged(DeviceEventArgs const& args) =
```

```
        0;
    virtual void onAudioStateChanged(DeviceEventArgs const& args) =
        0;
    virtual void onFlashButtonPressed(DeviceEventArgs const& args)
        = 0;
    virtual void onSmartButtonPressed(DeviceEventArgs const& args)
        = 0;
    };
```

### *6.2.1: void onTalkButtonPressed(DeviceEventArgs const& args)*

This method is called when a talk button press is detected. *DeviceEventArgs* are outlined below.

#### 6.2.1.1: DeviceEventArgs

```
eHeadsetButton buttonPressed;
eAudioState audioState;
bool mute;
```

### *6.2.2: void onButtonPressed(DeviceEventArgs const& args)*

This method is called when a button press is detected. *DeviceEventArgs* are outlined above.

### *6.2.3: void onMuteStateChanged(DeviceEventArgs const& args)*

This method is called when device mute state change is detected. *DeviceEventArgs* are outlined above.

### *6.2.4: void onAudioStateChanged(DeviceEventArgs const& args)*

This method is called when an audio state change is detected. *DeviceEventArgs* are outlined above.

### *6.2.5: void onFlashButtonPressed(DeviceEventArgs const& args)*

This method is called when flash on a button is detected. *DeviceEventArgs* are outlined above.

### *6.2.6: void onSmartButtonPressed(DeviceEventArgs const& args)*

This method is called when a "smart" button press is detected. This, in general, means a "long" button is pressed. *DeviceEventArgs* are outlined above.

## 6.3: IDeviceEventsExt Interface

```
The IDeviceEventsExt interface provides functions to register for notifications using
IDeviceEventsExtCallback.
```

```
INTF IDeviceEventsExt{
    virtual DMResult
        regCbDeviceEventsExt(IDeviceEventsExtCallback*) noexcept = 0;
    virtual DMResult
```

```
        unregCbDeviceEventsExt(IDeviceEventsExtCallback*) noexcept = 0;
    };
```

### 6.3.1: DMResult regCbDeviceEventsExt(IDeviceEventsExtCallback*)

This method registers device events "ext" receiver.

See Also: Section 4.1.1.1: DMResult enum.

### 6.3.2: DMResult unregCbDeviceEventsExt(IDeviceEventsExtCallback*)

This method unregisters device events "ext" receiver.

See Also: Section 4.1.1.1: DMResult enum.

## 6.4: IDeviceEventsExtCallback Interface

Inherit the *IDeviceEventsExtCallback* interface to receive Device Events "Ext" and then register these events using *IDeviceEventsExt*.

```
INTF IDeviceEventsExtCallback{
    virtual void onBatteryLevelChanged(BatteryLevelEventArgs
        const& args) = 0;
    virtual void onHeadsetStateChanged(HeadsetStateEventArgs
        const& args) = 0;
    };
```

### 6.4.1: void onBatteryLevelChanged(BatteryLevelEventArgs const& args)

This method is called when the status of the battery on the device changes.

### 6.4.2: void onHeadsetStateChanged(HeadsetStateEventArgs const& args)

This method is called when the state of the headset changes, for example, if it is put on (donned) or taken off (doffed).

#### 6.4.2.1: HeadsetStateEventArgs struct

```
enum {
        MAX_HEADSET_NAME = 64,
        MAX_SERNO_SIZE   = 16,
        MAX_PATH_SIZE    = 256
    };
int32_t headsetsInConference;
char headsetName[MAX_HEADSET_NAME];
eHeadsetState headsetState;
eProximity proximity;
eBTBatteryStatus batteryStatus;
eBTChargingStatus batteryChargStatus;
int32_t batteryLife;
```

```
uint8_t serialNumber[MAX_SERNO_SIZE];
wchar_t devicePath[MAX_PATH_SIZE];
```

## 6.5: IDeviceEventsQuery Interface

The *IDeviceEventsQuery* interface obtains an interface from the *IDeviceEvents* hierarchy. It is used because `dynamic_cast<>` does not work across DLL (.so) boundaries.

```
INTF IDeviceEventsQuery{
    virtual bool query(IDeviceEvents **p) noexcept = 0;
    virtual bool query(IDeviceEventsExt **p) noexcept = 0;
    virtual bool query(IBaseEvents **p) noexcept = 0;
    virtual bool query(IHIDPipeEvents **p) noexcept = 0;
    virtual bool query(IMobilePresenceEvents **p) noexcept = 0;
    virtual bool query(IMOCEventsCallback **p) noexcept = 0;
    virtual bool query(IMOCEvents **p) noexcept = 0;
    virtual bool query(IMOCEventsExtCallback **p) noexcept = 0;
    virtual bool query(IMOCEventsExt **p) noexcept = 0;
    virtual bool query(IAudioSenseEventsCallback **p) noexcept = 0;
    virtual bool query(IAudioSenseEvents **p) noexcept = 0;
    };
```

### *6.5.1: bool query(IDeviceEvents **p)*

This method obtains an *IDeviceEvents* interface.

### *6.5.2: bool query(IDeviceEventsExt **p)*

This method obtains an *IDeviceEventsExt* interface.

### *6.5.3: bool query(IBaseEvents **p)*

This method obtains an *IBaseEvents* interface.

### *6.5.4: bool query(IHIDPipeEvents **p)*

This method obtains an *IHIDPipeEvents* interface.

### *6.5.5: bool query(IMobilePresenceEvents **p)*

This method obtains an *IMobilePresenceEvents* interface.

### *6.5.6: bool query(IMOCEventsCallback **p*

This method obtains an *IIMOCEventsCallback* interface.

### *6.5.7: bool query(IMOCEvents **p*

This method obtains an *IMOCEvents* interface.

### *6.5.8: bool query(IMOCEventsExtCallback **p)*

This method obtains an *IIMOCEventsExtCallback* interface.

### *6.5.9: bool query(IMOCEventsExt **p)*

This method obtains an *IIMOCEventsExt* interface.

## 6.6: IBaseEvents Interface

The *IBaseEvents* interface provides functions to register for notifications using *IBaseEventsCallback*.

```
INTF IBaseEvents{
    virtual DMResult regCbBaseEvents(IBaseEventsCallback*) noexcept = 0;
    virtual DMResult unregCbBaseEvents(IBaseEventsCallback*) noexcept = 0;
    };
```

### *6.6.1: DMResult regCbBaseEvents(IBaseEventsCallback*)*

This method registers the base events receiver.

See Also: Section 4.1.1.1: DMResult enum.

### *6.6.2: DMResult unregCbBaseEvents(IBaseEventsCallback*)*

This method unregisters the base events receiver.

See Also: Section 4.1.1.1: DMResult enum.

## 6.7: IBaseEventsCallback Interface

Inherit the *IBaseEventsCallback* interface to receive Base Events and then register Base Events using *IBaseEvents*.

```
INTF IBaseEventsCallback{
    virtual void onBaseEventReceived(BaseEventArgs const& args) =
        0;
    };
```

### *6.7.1: void onBaseEventReceived(BaseEventArgs const& args)*

This method is called when a base event is detected. The *BaseEventArgs* struct is shown below.

#### 6.7.1.1: BaseEventArgs

```
struct BaseEventArgs
{
    enum {
        MAX_REPORT_SIZE = 64,
        MAX_PASSWORD_SIZE = 30,
        MAX_SERNO_SIZE = 16,
```

```
            MAX_FEATURE_LOCKS = 32
        };
        uint32_t count;
        uint8_t report[MAX_REPORT_SIZE];
        uint8_t password[MAX_PASSWORD_SIZE];
        uint8_t serialNumber[MAX_SERNO_SIZE];
        eBaseEventType baseEvent;
        eFeatureLock featureLock[MAX_FEATURE_LOCKS];
    };
```

The *BaseEventType* enum referenced above is shown in the next section.

#### 6.7.1.2: BaseEventType enum

BASE_EVENT_TYPE_UNKNOWN = 0
BASE_EVENT_TYPE_FEATURE_MASK = 1
BASE_EVENT_TYPE_PASSWORD = 2
BASE_EVENT_TYPE_RFPOWER_MODE = 3
BASE_EVENT_TYPE_RFLINK_TYPE = 4
BASE_EVENT_TYPE_VOIP_TALK = 5
BASE_EVENT_TYPE_PSTN_TALK = 6
BASE_EVENT_TYPE_MOBILE_TALK = 7
BASE_EVENT_TYPE_VOIP_TALK_HELD = 8
BASE_EVENT_TYPE_PSTN_TALK_HELD = 9
BASE_EVENT_TYPE_MOBILE_TALK_HELD = 10
BASE_EVENT_TYPE_PSTN_LINK_ESTABLISHED = 11
BASE_EVENT_TYPE_PSTN_LINK_DOWN = 12
BASE_EVENT_TYPE_VOIP_LINK_ESTABLISHED = 13
BASE_EVENT_TYPE_VOIP_LINK_DOWN = 14
BASE_EVENT_TYPE_MOBILE_LINK_ESTABLISHED = 15
BASE_EVENT_TYPE_MOBILE_LINK_DOWN = 16
BASE_EVENT_TYPE_PSTN_INCOMING_CALL_RING_ON = 17
BASE_EVENT_TYPE_PSTN_INCOMING_CALL_RING_OFF = 18
BASE_EVENT_TYPE_INTERFACE_STATE_CHANGE = 19
BASE_EVENT_TYPE_PSTN_TALK_AND_VOIP_TALK_HELD = 20
BASE_EVENT_TYPE_PSTN_TALK_AND_MOBILE_TALK_HELD = 21
BASE_EVENT_TYPE_VOIP_TALK_AND_MOBILE_TALK_HELD = 22
BASE_EVENT_TYPE_PSTN_TALK_AND_SUBSCRIBE_HELD = 23
BASE_EVENT_TYPE_SUBSCRIBE = 24
BASE_EVENT_TYPE_SUBSCRIBE_HELD = 25
BASE_EVENT_TYPE_DIALPAD = 26
BASE_EVENT_TYPE_BT_AUDIO_LOCATION = 27
BASE_EVENT_TYPE_VOIP_TALK_AND_SUBSCRIBE_HELD = 28
BASE_EVENT_TYPE_MOBILE_TALK_AND_SUBSCRIBE_HELD = 29
BASE_EVENT_TYPE_SERIAL_NUMBER= 30
BASE_EVENT_TYPE_DESKPHONE_HEADSET = 31
BASE_EVENT_TYPE_MOBILE_CONNECTED = 32
BASE_EVENT_TYPE_MOBILE_DISCONNECTED = 33

## 6.8: IMobilePresenceEvents Interface

The *IMobilePresenceEvents* interface provides functions to register for notifications through *IHIDPipeEventsCallback*.

```
INTF IMobilePresenceEvents{
    virtual DMResult
        regCbMobilePresenceEvents(IMobilePresenceEventsCallback*) noexcept =
            0;
    virtual DMResult
        unregCbMobilePresenceEvents(IMobilePresenceEventsCallback*)
            noexcept = 0;
    };
```

### *6.8.1: DMResult regCbMobilePresenceEvents(IMobilePresenceEventsCallback*)*

This method registers the Mobile presence events receiver.

See Also: Section 4.1.1.1: DMResult enum.

### *6.8.2: DMResult unregCbMobilePresenceEvents(IMobilePresenceEventsCallback*)*

This method unregisters the Mobile presence events receiver.

See Also: Section 4.1.1.1: DMResult enum.

## 6.9: IMobilePresenceEventsCallback Interface

Inherit the *IMobilePresenceEventsCallback* interface to receive Mobile presence Events and then register these events through *IMobilePresenceEvents*.

```
INTF IMobilePresenceEventsCallback{
    virtual void onPresenceChanged(MobilePresenceEventArgs const&
        args) = 0;
    };
```

### *6.9.1: void onPresenceChanged(MobilePresenceEventArgs const& args)*

This method detects a change in a mobile presence and sends an event that corresponds to the change.

# Chapter 7: Interfaces Intended for Internal Use Only

The interfaces in this section are visible in the SDK, but are intended for internal use only, and, as such, are undocumented. Use at your own risk.

| Internal Interface Name | |
|---|---|
| *IAudioIntelligenceEvents* | ✗ |
| *IAudioIntelligenceEventsCallback* | ✗ |
| *IAudioSense* | ✗ |
| *IAudioSenseEvents* | ✗ |
| *IAudioSenseEventsCallback* | ✗ |
| *IBTDeviceSettings* | ✗ |
| *IConnectedDevice* | ✗ |
| *IDelphiCommand* | ✗ |
| *IDeviceCookie* | ✗ |
| *IDeviceExtendedInfo* | ✗ |
| *IDisplayDeviceCommand* | ✗ |
| *IExtendedVersion* | ✗ |
| *IGenericDeviceCallback* | ✗ |
| *IHIDPipeEvents* | ✗ |
| *IHIDPipeEventsCallback* | ✗ |

| | |
|---|---|
| *IMOCCommand* | ✗ |
| *IMOCCommandExt* | ✗ |
| *IMOCEvents* | ✗ |
| *IMOCEventsCallback* | ✗ |
| *IMOCEventsExt* | ✗ |
| *IMOCEventsExtCallback* | ✗ |
| *IUserPreference* | ✗ |
| *IVOIPCall* | ✗ |

# Chapter 8: Using the Plantronics SDK REST Service

Representational State Transfer (REST) is an HTTP-based architecture style for designing distributed applications. This chapter describes how to use the Plantronics SDK REST Service to integrate Plantronics SDK functionality into your application.

The REST interface is a subset of the native interface. It includes only the APIs discussed in the rest of this chapter.

The following topics are included in this chapter:

- About the Plantronics SDK REST Service
- Using Device Services
- Getting ATD Events
- Using Session Manager Services
- Using Call Services
- Using Configuration Services
- Handling Incoming Data: the REST Response Format
- Code Snippets

If you are migrating from Spokes 2.7 to Plantronics SDK v3.6, you may want to read Chapter 9: Appendix A: Plantronics SDK v3.6 Feature Parity.

## 8.1: About the Plantronics SDK REST Service

The REST service plug-in hosts a REST Service to expose certain features of the Device Manager SDK and the Plantronics SDK. The Plantronics SDK REST Service can be used by business applications (Web based or thick clients) to interface with the Spokes runtime engine via a RESTful interface on the client computer system.

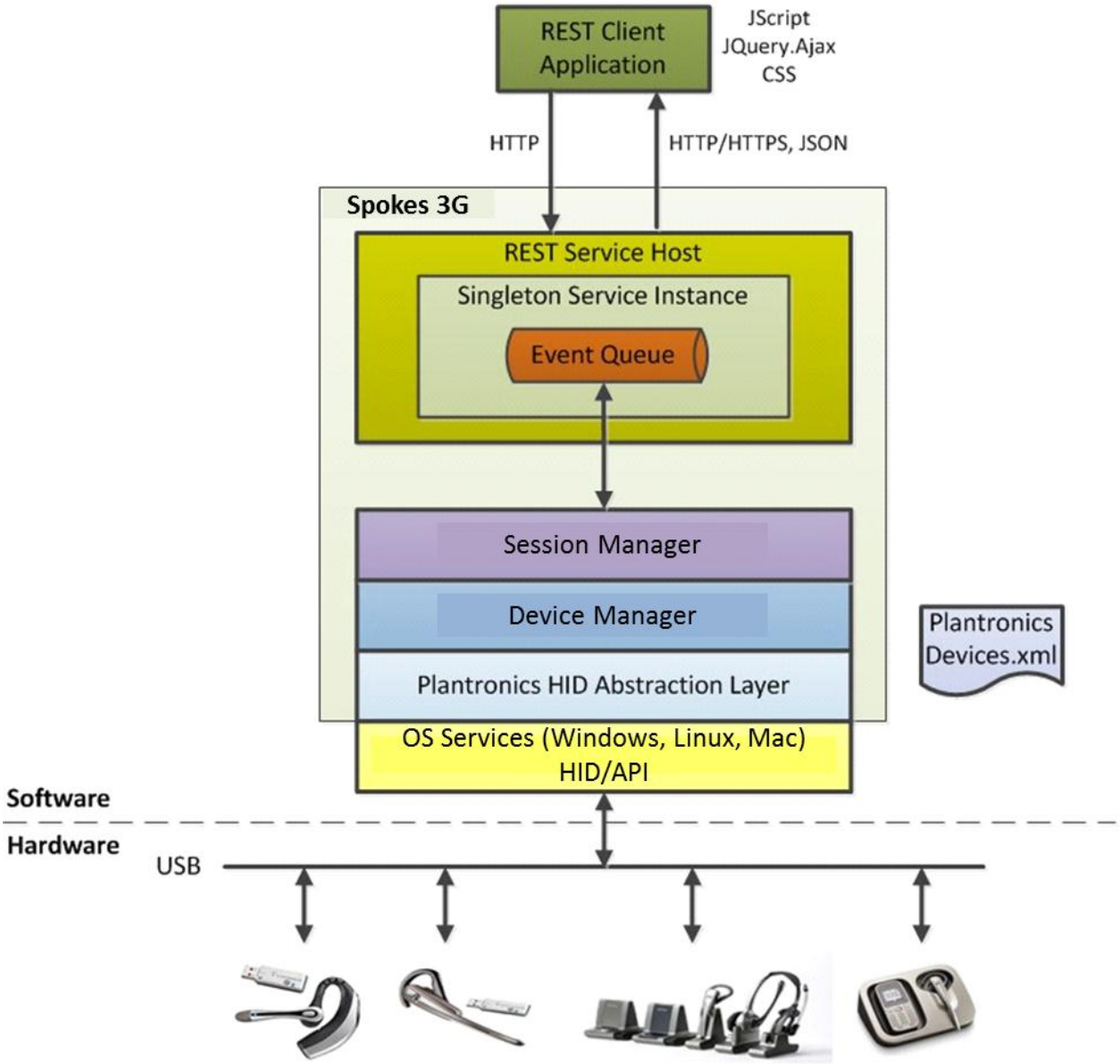
### *8.1.1: About REST*

REST (REpresentational State Transfer) is an architecture style for designing distributed applications. With REST, rather than using a distributed application framework such as CORBA, XML RPC, or SOAP to build services, simple HTTP is used to make calls between a client and a service. REST is a lightweight alternative to other mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL), as there are no client side proxies or stubs needed, and the request response is handled via basic HTTP GET/POST/PUT or DELETE. RESTful applications identify resources with well-defined URLs and use simple HTTP requests for all basic CRUD (Create (POST) / Retrieve (GET) / Update (PUT) / Delete (DELETE)) operations.

Like Web Services, a REST service is:

- Platform-independent: The client and server can be running on different platforms/OS.
- Standards-based: It can run on top of HTTP/HTTPS.
- Can easily be used in the presence of firewalls.

Unlike Web Services and WS-* standards, REST offers no built-in end-to-end message level security, encryption, session management, transaction, and QoS guarantees. In RESTful services, username/password tokens are often used for security. For encryption, REST services can use HTTPS. The "ST" in "REST" stands for "State Transfer", and in a good REST design, operations are self-contained. Each client request carries with it (transfers) all the information (state) that the service needs in order to complete the request.




## *8.1.2: Understanding the REST Service Architecture*

**Figure 2: REST Service Architecture**

The Plantronics SDK software stack has the functional modules shown in the following table.

**Table 1: Modules in Plantronics SDK Software Stack**

| Module | Description |
|---|---|
| REST Client Application | Any client application (such as a browser) that initiates REST requests via HTTP and processes the JSON responses. Application code can be written in any popular language, including JScript, JQuery.Ajax, CSS, and so on. Third party business applications, web clients (via JQuery Ajax post back), or thick clients (via simple HTTPWebRequest and HTTPWebResponse) can interface out of process with the Plantronics SDK REST Service and communicate with the device and the Spokes runtime engine.<br>REST client applications are language independent, as long as HTTP protocol is supported by the language. |
| REST Service Host | Plug-in that acts as a host for REST service instances. |
| Session Manager | API that allows the REST Service Host to access Device Services and Call Services.<br>See Section 8.4: Using Session Manager Services. |
| Device Manager | Lower-level API support for Device Services and Call Services.<br>See Section 8.2:Using Device Services and Section 8.5: Using Call Services. |
| PLT HID Abstraction Layer | Plantronics-built wrapper around the Windows HID. Internal use only. |
| OS Services (Windows, Linux, Mac) HID and API | Low-level USB hardware management via the OS's Human Interface Driver (HID) and API. |
| PlantronicsDevices.xml | Configuration file for Plantronics devices and their attributes. |

### *8.1.3: REST Services*

REST Services expose the Device Services, Session Manager Services, Call Services, and Configuration Services with well-known URLs so that clients can make HTTP POST and GET requests, and can access services in a RESTful way. The service instance registers for device events and Session Manager-level events, such as:

- headset button presses
- headset state changes
- base/hub/handset button presses
- base/hub/handset state changes
- ATD state changes (Mobile and PSTN call indications)
- call state changes

*Device Services* expose device commands and device events. The Device Service exposes URIs for getting a device list, getting metadata about a given device (such as PID, product name, version #, serial #, and so on), sending commands to a given device, and querying for events from a given device. Device Services are discussed in Section 8.2: Using Device Services.

*Session Manager Services* expose session commands and session events. The Session Manager Service exposes URIs for registering and unregistering a plug-in, defining the active state for a plug-in, and querying for events from all plug-ins. Session Manager Services are discussed in Section 8.4: Using Session Manager Services.

*Call Services* expose call commands and call events. The Call Service exposes URIs for querying active plug-ins, sending call commands and querying call states per registered client, querying make call requests, and querying Session Manager-level call states across plug-ins. Call Services are discussed in Section 8.5: Using Call Services.

*Configuration Services* expose commands for managing event queues. The Configuration Service exposes URIs for retrieving the list of event queues on which the application is currently subscribed, adding or removing event queues, and setting maximum count and time to live for event queues. Configuration Services are discussed in Section 8.6: Using Configuration Services.

## 8.2: Using Device Services

This section describes the device-related REST endpoints implemented inside the server.

- Anything found inside a {} is a variable name that is passed in the request.
- All requests are made through GET due to the Same Origin Policy (SOP). SOP permits scripts running on pages originating from the same site – a combination of scheme, hostname, and port number – to access each other's methods and properties with no specific restrictions, but prevents access to most methods and properties across pages on different sites.
- All devices have a Unique Identifer (UID). This UID is used when addressing a single device.
- To attach to a device and receive events, the user must create a session. The Session ID is then used to communicate with the device.

Some URLs have changed for the following reasons:

- Supports only one active device.
- Supports multiple user sessions.
- Support of proximity events.

For a listing of changes to URLs from Spokes 2.7 to Plantronics SDK v3.6, refer to 12.2: Listing of URI Changes.

All URLs start with `http://localhost:port/Spokes`. Default value for "`localhost`" is 127.0.0.1. Default for "`port`" is `32017`.

### *8.2.1: Get Single DeviceInfo Object for a Single Object*

```
http://localhost:port/Spokes/DeviceServices/Info
```

- Returns: A *DeviceInfo* data structure for the device with the requested UID.

### *8.2.2: Attach to a Device*

```
http://localhost:port/Spokes/DeviceServices/Attach?uid={uid}
```

A successful attach request returns a session ID that is used to track events and other actions.

- Parameters: *{uid}* is the unique identifier of a device.
- Returns: A session hash string (or an error string).

### *8.2.3: Delete a Session ID*

```
http://localhost:port/Spokes/DeviceServices/Release?sess={sess}
```

The system deletes a Session ID automatically if a session has not had any activity for 30 seconds.

- Parameters: *{sess}* is the unique hash of a session.
- Returns: True or false (success or failure).

### *8.2.4: Get all Events*

```
http://localhost:port/Spokes/DeviceServices/Events?sess={sess}&queue={queue}
```

Returns an array of events. The client can optionally specify the queue to receive. This is an ordered integer of queue types that increase by a power of 2.

- Parameters:
    - {sess} is the unique hash of a session.
    - {queue} is the number of the queue or queues in which you are interested:
      0 means all queues, else select the appropriate queue number (Unknown = 0, DeviceStateChange = 1, HeadsetStateChange = 2, HeadsetButtonPressed = 4, BaseStateChange = 8, BaseButtonPressed = 16, CallStateChange = 32, ATDStateChange = 4); may be a binary OR of any set of these values.
- Returns: A list of *DeviceEvent* data structures for the requested queue.

### *8.2.5: Get Headset Events*

```
http://localhost:port/Spokes/DeviceServices/HeadsetEvents?sess={sess}
```

This is a convenience method that looks at two queues: *HeadsetStateChange* OR *HeadsetButtonPressed*.

- Parameters: *{sess}* is the unique hash of a session.
- Returns: An array of *DeviceEvent* data structures for the requested queues.

### *8.2.6: Get Base Events*

```
http://localhost:port/Spokes/DeviceServices/BaseEvents?sess={sess}
```

Returns an array of base events. This is a convenience method that looks at two queues: *BaseStateChange* OR *BaseButtonPressed.*

- Parameters: *{sess}* is the unique hash of a session.
- Returns: An array of *DeviceEvent* data structures for the requested queues.

## 8.3: Getting ATD Events

```
http://localhost:port/Spokes/DeviceServices/ATDEvents?sess={sess}
```

Returns an array of ATD events. This is a convenience method that looks at the *ATDStateChange* queue.

- Parameters: *{sess}* is the unique hash of a session.
- Returns: An array of *DeviceEvent* data structures for the requested queue.

### *8.3.1: Enable or Disable the Ringer*

```
http://localhost:port/Spokes/DeviceServices/Ring?enabled={enabled}
```

To enable the ringer, set to true. To disable the ringer, set to false.

- Parameters:
    - *{enabled}* when set to true turns the ringer on, false turns it off.
- Returns: True or an error.

### *8.3.2: Control the Audio State*

```
http://localhost:port/Spokes/DeviceServices/AudioState?state={state}
```

Defines the audio state for the device.

- Parameters:
    - {state} can be one of: Unknown = 0, MonoOn = 1, MonoOff = 2, StereoOn = 3, StereoOff = 4, MonoOnWait = 5, StereoOnWait = 6.
- Returns: True or an error.

### *8.3.3: Get Proximity Events*

```
http://localhost:port/Spokes/DeviceServices/Proximity?
sess={sess}&enabled={enabled}
```

To enable/disable proximity reporting on a device. Proximity events are reported as *HeadsetStateChanged* events (if device supports Proximity!).

- Parameters:
    - {sess} is the unique hash of a session.
    - {enabled} when set to true enables proximity events, false disables collecting events.
- Returns: True or an error.

### *8.3.4: Get ATD Mobile CallerID*

```
http://localhost:port/Spokes/DeviceServices/ATDMobileCallerId?sess={sess}
```

Returns ATD mobile caller ID attribute of a caller with ATD events.

- Parameters:
    - {sess} is the unique hash of a session.
- Returns: a string of caller ID.

## 8.4: Using Session Manager Services

The Session Manager is the interface with which an application communicates to register its plug-in. The Session Manager provides the basic communication interface that Spokes provides when starting and stopping a plug-in.

### *8.4.1: Register a Plug-in*

```
http://localhost:port/Spokes/SessionManager/Register?name={name}
```

Registers a plug-in with the system. Upon success, notifies the client that the plug-in is registered. Use the name of the plug-in for all REST calls involving the plug-in. Also, after a plug-in is registered, the name (instead of a session ID) can be used with *DeviceServices* to communicate with a headset.

- Parameters: *{name}* is the registered name of the plug-in.
- Returns: True or false (indicating success or failure).

### *8.4.2: Unregister a Plug-In with the System*

```
http://localhost:port/Spokes/SessionManager/UnRegister?name={name}
```

- Parameters: *{name}* is the plug-in name.
- Returns: True or false (indicating success or failure).

### *8.4.3: Define the Active State for a Plug-In*

```
http://localhost:port/Spokes/SessionManager/IsActive?name={name}&active={active}
```

The *{active}* parameter allows you to check whether a plug-in is active or whether it is inactive.

- Parameters:
    - *{name}* is the registered plug-in name.
    - *{active}* allows you to test for active (true) or inactive (false).
- Returns: True or false (indicating success or failure).

### *8.4.4: Get all Known Plug-Ins*

```
http://localhost:port/Spokes/SessionManager/PluginList
```

- Parameters: None.
- Returns: List of plug-in names or a REST error.

## 8.5: Using Call Services

Call Services provide access to *ICallServices*. The client specifies the plug-in to request a call service name when required.

### *8.5.1: Get the State of the Call Manager Object*

```
http://localhost:port/Spokes/CallServices/CallManagerState
```

- Parameters: None.
- Returns: A representation of the call manager state.

### *8.5.2: Get the Session Manager Events that have been Queued*

```
http://localhost:port/Spokes/CallServices/Events
```

- Parameters: None.

- Returns: List of *CallState* objects (this is the *CallState* enum from the Plantronics.UC.Common assembly: Unknown = 0, AcceptCall = 1, TerminateCall = 2, HoldCall = 3, Resumecall = 4).

### *8.5.3: Get all Session Manager Call Events for the Specified Plug-In*

```
http://localhost:port/Spokes/CallServices/SessionManagerCallEvent?name={name}
```

- Parameters: *{name}* is the registered name of a plug-in.
- Returns: List of *CallState* objects registered to a named plug-in.

### *8.5.4: Get List of all Call Events for the Specified Plug-In*

```
http://localhost:port/Spokes/CallServices/CallEvents?name={name}
```

- Parameters: *{name}* is the registered name of a plug-in.
- Returns: A list of calls.

### *8.5.5: Get a List of All Call Requests for the Specified Plug-In*

```
http://localhost:port/Spokes/CallServices/CallRequests?name={name}
```

- Parameters: *{name}* is the registered name of a plug-in.
- Returns: A list of contacts (name, friendly name, and so on).

### *8.5.6: Define an Incoming Call*

```
http://localhost:port/Spokes/CallServices/IncomingCall?name={name}&
callID={callID}&contact={contact}&tones={tones}&route={route}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
    - *{name}* is the registered name of a plug-in.
    - *{callID}* is the identifier of a call.
    - *{contact}* is the name of a contact.
    - *{tones}* is the ring tone to use.
    - *{route}* is the audio route: ToHeadset = 0, ToSpeaker = 1.
- Returns: True or false.

### *8.5.7: Define an Outgoing Call*

```
http://localhost:port/Spokes/CallServices/OutgoingCall?name={name}&callID={callID}&contac
t={contact}&route={route}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
    - *{name}* is the registered name of a plug-in.
    - *{callID}* is the identifier of a call.
    - *{contact}* is the name of a contact.
    - *{route}* is the audio route: ToHeadset = 0, ToSpeaker = 1.
- Returns: True or false.

### *8.5.8: Terminate the Call with the Specified CallID*

```
http://localhost:port/Spokes/CallServices/TerminateCall?name={name}&callID={callID}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{callID}* is the identifier of a call.
- Returns: True or false.

### *8.5.9: Answer the Call with the Specified CallID*

```
http://localhost:port/Spokes/CallServices/AnswerCall?name={name}&callID={callID}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{callID}* is the identifier of a call.
- Returns: True or false.

### *8.5.10: Put the Call with the Specified CallID on Hold*

```
http://localhost:port/Spokes/CallServices/HoldCall?name={name}&callID={callID}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{callID}* is the identifier of a call.
- Returns: True or false.

### *8.5.11: Resume a Call with the Specified CallID*

```
http://localhost:port/Spokes/CallServices/ResumeCall?name={name}&callID={callID}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{callID}* is the identifier of a call.
- Returns: True or false.

### *8.5.12: Define the Mute State for the Call with the Specified CallID*

```
http://localhost:port/Spokes/CallServices/MuteCall?name={name}&muted={muted}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{mute}* indicates whether to mute (true) or unmute (false) the call.
- Returns: True or false.

### *8.5.13: Insert the Specified Contact into the Call with the Specified CallID*

```
http://localhost:port/Spokes/CallServices/InsertCall?name={name}&callID={callID}&contact=
{contact}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{callID}* is the identifier of a call.
  - *{contact}* is the name of a contact.
- Returns: True or false.

### *8.5.14: Define an Audio Route for the Specified CallID*

```
http://localhost:port/Spokes/CallServices/SetAudioRoute?name={name}&
callID={callID}&route={route}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{callID}* is the identifier of a call.
  - *{route}* is the audio route: ToHeadset = 0, ToSpeaker = 1.
- Returns: True or false.

### *8.5.15: Define a Call that is in a Conference*

```
http://localhost:port/Spokes/CallServices/SetConferenceId?name={name}&
callID={callID}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{callID}* is the identifier of a call.
- Returns: True or false.

### *8.5.16: Make a Call to the Specified Contact*

```
http://localhost:port/Spokes/CallServices/MakeCall?name={name}&contact={contact}
```

All complex variables must be based inline on the URI as JSON objects.

- Parameters:
  - *{name}* is the registered name of a plug-in.
  - *{contact}* is the name of a contact.
- Returns: True or false.

## 8.6: Using Configuration Services

This section describes configuration service REST endpoints implemented inside the server.

### 8.6.1: Return the Queue's Registry

```
http://localhost:port/Spokes/EventManager/GetRegistry?sess={sess}
```

*GetRegistry* returns the queues with which you are registered. Each queue is a power of two from the previous one, and all the ones you register for are OR'ed together.

- Parameters: *{sess}* is the unique hash of the session.
- Returns: The queues with which you are registered.

### 8.6.2: Return the Registry Queue's List

```
http://localhost:port/Spokes/EventManager/SetRegistry?sess={sess}&queue={queue}
```

*SetRegistry* returns the queues with which you are registered. The parameter is the list of queues with which you would like to register.

- Parameters:
    - *{sess}* is the unique hash of the session.
    - *{queue}* is the list of queues to be registered with.
- Returns: The queues you are registered with.

### 8.6.3: Add Queues to the Registry

```
http://localhost:port/Spokes/EventManager/AddRegistry?sess={sess}&queue={queue}
```

*AddRegistry* adds the queue(s) in the second parameter to the list of queues with which a session is registered.

- Parameters:
    - *{sess}* is the unique hash of the session.
    - *{queue}* is the list of queues to be added to current queue list.
- Returns: The list of queues you are registered with.

### 8.6.4: Remove Queues from the Registry

```
http://localhost:port/Spokes/EventManager/RemoveRegistry?sess={sess}
&queue={queue}
```

*RemoveRegistry* removes the named queue(s) from the set with which a session is registered.

- Parameters:
    - *{sess}* is the unique hash of the session.
    - *{queue}* is the list of queues to be removed from current queue list.
- Returns: The list of queues you are registered with.

### 8.6.5: Maximum Time to Live (TTL) for all Queues

```
http://localhost:port/Spokes/EventManager/GlobalTTL?sess=
{sess}&ttl={ttl}
```

*GlobalTTL* sets the max time to live of all queues.

- Parameters:
    - *{sess}* is the unique hash of the session.

- {ttl} is the maximum time to live of all queues.
- Returns: The current max time to live.

### *8.6.6: Maximum Count for Queues*

```
http://localhost:port/Spokes/EventManager/GlobalMaxCount?sess={sess}&max={max}
```

*GlobalMaxCount* sets the max count for all queues.

- Parameters:
  - *{sess}* is the unique hash of the session.
  - *{max}* is the maximum count for all queues.
- Returns: The current max count.

### *8.6.7: Maximum Time to Live (TTL) for Queue(s)*

```
http://localhost:port/Spokes/EventManager/TTL?sess={sess}&queue=
{queue}&ttl={ttl}
```

*TTL* sets the max time to live for a queue or set of queues.

- Parameters:
  - *{queue}* is the list of queues for which to set the TTL.
  - *{ttl}* is the maximum time to live for selected queues.
- Returns: The max time to live for that queue or queues.

### *8.6.8: Maximum Count for Queue(s)*

```
http://localhost:port/Spokes/EventManager/MaxCount?sess={sess}&queue={queue}
&max={max}
```

*MaxCount* sets the max count for a queue or set of queues.

- Parameters:
  - *{queue}* is the list of queues for which to set the max count.
  - *{max}* is the max count to set for selected queues.
- Returns: The max count for that queue or set of queues.

## 8.7: Handling Incoming Data: the REST Response Format

All responses are returned in the *RestResponse<T>* object. This provides a consistent way for applications to deal with incoming data. The *RestResponse<T>* object consists of:

```
//The  type of response the user has given
[DataMember]
public RestResponseType Type = RestResponseType.Unknown;

//The type of response the user has given
[DataMember]
public string Type_Name = RestResponseType.Unknown.ToString();
```

```
//Defines the user object
[DataMember]
public string Description = "";

//True if there was an error and the value should be ignored
[DataMember]
public bool isError = false;

[DataMember]
public RestError Err = null;

//Defines the user object
[DataMember]
public T Result = default(T);
```

## 8.8: Code Snippets

The following sections contain bits of code to help further explain the APIs.

### *8.8.1: Compress Queue List to a Number*

This code compresses an array of queue items down to a single number.

```
//Compresses an array of queue items down to a single number
function compressQueueList( queue )
{
  var val = 0;

    //If we aren't an array, just return queue
  if ( queue.constructor.toString().indexOf("Array") == -1 )
    return queue;

    //We have an array, go through it and or everthing together
  for ( var i = 0; i < queue.length; i++ )
    if ( typeof( queue[i] ) === "number" && queue[i] > 0 )
      val |= queue[i];

  return val;
}
```

### *8.8.2: Get device Info*

This code gets device information.

```
//Get Device Information
local = this;
$.getJSON(this.Path+"/DeviceServices/Info?callback=?",
    function(data)
```

```
    {
       //Create a nice object, and ensure its of the right type
      var resp = new SpokesResponse( data );
      if ( resp.Type != SpokesResponseType.DeviceInfo )
        resp.isValid = false;

        //Store my device uid
      if ( resp.isValid && !resp.isError )
      {
       local.Device_Id = resp.Result.Uid;
      }

        //Pass my result back to the user
      action.isValid = false;
      if (action.Callback != null && action.Callback != undefined)
          action.Callback(resp);
    } );
```

### *8.8.3: Attach a Device and Register a Session*

This code attaches a device and registers a session.

```
  //Register the callback
var action = new SpokesAction( callback );
SpokesAction.Action_List.unshift( action );

  //Register a session
local = this;
$.getJSON(this.Path + "/DeviceServices/Attach?uid="+ uid +"&callback=?",
    function(data)
    {
        //Create a nice object, and ensure its of the right type
      var resp = new SpokesResponse( data );
      if ( resp.Type != SpokesResponseType.SessionHash )
        resp.isValid = false;

        //Store my session and set that we are attached
      if ( resp.isValid && !resp.isError )
      {
        local.Sess_Id = resp.Result;
        local.isAttached = true;
      }

        //Pass my result back to the user
      action.isValid = false;
      if ( typeof( action.Callback ) === 'function' )
        action.Callback( resp );
    } );
```

### *8.8.4: Register for Events*

This code returns all the valid events that have happened since the last call.

```
Device.prototype.events = function( queue, callback )
{
    //Check if I was only given one argument
  if ( callback == undefined && typeof( queue ) === 'function' )
  {
    callback = queue;
    queue = 0;
  }

    //If they gave me an array of queue items, compress them down to an int
  queue = compressQueueList( queue );

    //Can't release, we aren't attached
  if ( callback == null || callback == undefined )
    return false;

    //Register the callback
  var action = new SpokesAction( callback );
  SpokesAction.Action_List.unshift( action );

    //Register a session
  $.getJSON(this.Path+"/DeviceServices/Events?sess="+ this.Sess_Id +"&queue="+ queue
+"&callback=?",
      function(data)
      {
          //Create a nice object, and ensure its of the right type
        var resp = new SpokesResponse( data );
        if ( resp.Type != SpokesResponseType.DeviceEventArray )
          resp.isValid = false;

          //Pass my result back to the user
        action.isValid = false;
        if (action.Callback != null && action.Callback != undefined)
           //resp.Result
           action.Callback(resp);
            });

  return true;
}
```

# Chapter 9: Appendix A: Plantronics SDK v3.6 Feature Parity

On the Windows platform, Plantronics SDK v3.6 is compatible with the 32-bit Windows XP operating system and 32-bit and 64-bit Vista, Windows 7 and Windows 8 operating systems. On the Mac operating system, Plantronics SDK v3.6 is compatible with 10.7 or later.

This section shows functions and several interfaces, enabling developers to select the level of integration desired with the Plantronics platform. Depending on which interface is used, developers have different application deployment options.

## 9.1: ISessionManager Interface

For descriptions of these functions, read section 0 See Also: *SMResult* enum values are listed in section 2.8.1.1: SMResult enum.

ISessionManager Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `SMResult registerSession(const wchar_t* pPluginName, ppSession ppSess)` | ✓ | ✓ | ✓ |
| `SMResult unregisterSession(pSession pSess)` | ✓ | ✓ | ✓ |
| `SMResult setPluginStatus(int32_t pluginId, ePluginState pluginState)` | ✓ | ✗ | ✓ |
| `SMResult getActive(int32_t pluginId, bool& bActive)` | ✓ | ✗ | ✗ |
| `SMResult setOption(eSessionMgrOption eDOption, int32_t lValue)` | ✓ | ✗ | ✗ |
| `SMResult getOption(eSessionMgrOption eDOption, int32_t* plValue, int16_t iMaxLength, int16_t* piLength)` | ✓ | ✗ | ✗ |
| `SMResult registerCallback(pSessionManagerEvents callback)` | ✓ | ✓ | ✓ |

| | | | |
|---|---|---|---|
| `SMResult unregisterCallback(pSessionManagerEvents callback)` | ✓ | ✓ | ✓ |
| `SMResult getUserPreference(IUserPreference** userPref)` | ✓ | ✓ | ✗ |
| `SMResult getCallManagerState(ICallManagerState ** callManagerState)` | ✓ | ✓ | ✓ |
| `SMResult getActiveDevice(ppDevice ppDev)` | ✓ | ✗ | ✗ |
| `SMResult getDeviceForPath(const wchar_t* devicePath, ppDevice ppDev)` | ✓ | ✗ | ✗ |

## 9.2: ISessionManagerEvents Interface

For descriptions of these functions, read section 2.10: ISessionManagerEvents Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `bool OnCallStateChanged(CallStateEventArgs const& cseArgs)` | ✓ | ✓ | ✓ (not complete) |
| `bool OnDeviceStateChanged(DeviceStateEventArgs const& devArgs)` | ✓ | ✓ | ✗ |

## 9.3: ICallCommand Interface

For descriptions of these functions, read section 2.2: ICallCommand Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|

| | | | |
|---|---|---|---|
| `CMResult incomingCall(pCall callID, pContact contact, eRingTone tones, eAudioRoute route)` | ✓ | ✓ | ✓ |
| `CMResult outgoingCall(pCall callID, pContact contact, eAudioRoute route)` | ✓ | ✓ | ✓ |
| `CMResult terminateCall(pCall callID)` | ✓ | ✓ | ✓ |
| `CMResult answeredCall(pCall callID)` | ✓ | ✓ | ✓ |
| `CMResult holdCall(pCall callID)` | ✓ | ✓ | ✓ |
| `CMResult resumeCall(pCall callID)` | ✓ | ✓ | ✓ |
| `CMResult insertCall(pCall callID, pContact contact)` | ✓ | ✓ | ✓ |
| `CMResult muteCall(bool bMute)` | ✓ | ✓ | ✓ |
| `CMResult setAudioRoute(pCall callID, eAudioRoute route)` | ✓ | ✓ | ✓ |
| `CMResult makeCall(pContact contact)` | ✓ | ✓ | ✓ |
| `CMResult setConferenceId(pCall callID)` | ✓ | ✓ | ✓ |

## 9.4: ISession Interface

This interface is indirectly supported.

For descriptions of these functions, read 2.8: ISession Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `SMResult getSessionID(wchar_t* id, uint32_t len)` | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| `SMResult getPluginId(int32_t* pluginId)` | ✓ | ✗ | ✗ |
| `SMResult getPluginName(wchar_t* pName, uint32_t len)` | ✓ | ✓ | ✓ |
| `SMResult registerCallback(pCallEvents callback)` | ✓ | ✓ | ✓ |
| `SMResult unregisterCallback(pCallEvents callback)` | ✓ | ✓ | ✓ |
| `SMResult getCallCommand(ppCallCommand ppCommand)` | ✓ | ✓ | ✓ |
| `SMResult getCallEvents(ppCallEvents ppEvents)` | ✓ | ✗ | ✗ |
| `SMResult getActiveDevice(ppDevice ppDev)` | ✓ | ✓ | ✓ (indirectly supported) |
| `SMResult getActiveDeviceName(wchar_t* pDeviceName, int32_t len)` | ✓ | ✓ (indirectly supported) | ✓ |
| `SMResult enableAudio(bool bEnable)` | ✓ | ✗ | ✗ |

## 9.5: ICallEvents Interface

For descriptions of these functions, read section 2.3: ICallEvents Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `bool OnCallStateChanged(CallStateEventArgs const& pcscArgs)` | ✓ | ✓ | ✓ |

| | | | |
|---|---|---|---|
| `bool OnCallRequest(CallRequestEventArgs const& pcscArgs)` | ✓ | ✓ | ✓ |

## 9.6: ICallManagerState Interface

For descriptions of these methods, read section 2.6: ICallManagerState Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `bool HasActiveCall()` | ✓ | ✓ | ✓ |
| `int32_t numberOfCallsOnHold()` | ✓ | ✓ | ✓ |
| `bool getCalls(ICallInfoGroup** group)` | ✓ | ✓ | ✓ |

## 9.7: ICallInfoGroup Interface

For descriptions of these functions, read section 2.5: ICallInfoGroup Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `ICallInfo * getCall(uint32_t index)` | ✓ | ✓ | ✓ |
| `uint32_t numCalls()` | ✓ | ✓ | ✓ |

## 9.8: ICallInfo Interface

For descriptions of these functions, read section 2.4: ICallInfo Interface

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `int32_t getCallId()` | ✓ | ✓ | ✓ |
| `bool getSessionId(char * sessionid, int32_t len)` | ✓ | ✓ | ✓ |

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `bool getSource(wchar_t * source, int32_t len)` | ✓ | ✓ | ✓ |
| `bool isActive()` | ✓ | ✓ | ✓ |

## 9.9: ICall Interface

For descriptions of these functions, read section 2.1: ICall Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `int32_t getID()` | ✓ | ✓ | ✓ |
| `void setID(int32_t id)` | ✓ | ✓ | ✓ |
| `int32_t getConferenceID()` | ✓ | ✓ | ✓ |
| `void setConferenceID(int32_t id)` | ✓ | ✓ | ✓ |
| `bool getInConference()` | ✓ | ✓ | ✓ |
| `void setInConference(bool bConference)` | ✓ | ✓ | ✓ |

## 9.10: IContact Interface

Most of the functions here are not used.

For descriptions of these functions, read section 2.7: IContact Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `const wchar_t* getName()` | ✓ | ✓ | ✓ |
| `void setName(const wchar_t* pName)` | ✓ | ✓ | ✓ |

| | | | |
|---|---|---|---|
| `const wchar_t* getFriendlyName()` | ✓ | ✓ | ✓ |
| `void setFriendlyName(const wchar_t* pFName)` | ✓ | ✓ | ✓ |
| `int32_t getID()` | ✓ | ✓ | ✓ |
| `void setID(int32_t id)` | ✓ | ✓ | ✗ |
| `const wchar_t* getSipUri()` | ✓ | ✓ | ✓ |
| `void setSipUri(const wchar_t* pSip)` | ✓ | ✓ | ✗ |
| `const wchar_t* getPhone()` | ✓ | ✓ | ✓ |
| `void setPhone(const wchar_t* pPhone)` | ✓ | ✓ | ✗ |
| `const wchar_t* getEmail()` | ✓ | ✓ | ✓ |
| `void setEmail(const wchar_t* pEmail)` | ✓ | ✓ | ✗ |
| `const wchar_t* getWorkPhone()` | ✓ | ✓ | ✓ |
| `void setWorkPhone(const wchar_t* pWPhone)` | ✓ | ✓ | ✗ |
| `const wchar_t* getMobilePhone()` | ✓ | ✓ | ✓ |
| `void setMobilePhone(const wchar_t* pMPhone)` | ✓ | ✓ | ✗ |
| `const wchar_t* getHomePhone()` | ✓ | ✓ | ✓ |
| `void setHomePhone(const wchar_t* pHPhone)` | ✓ | ✓ | ✗ |

## 9.11: IDevice Interface

For descriptions of these functions, read 3.2: IDevice Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `int32_t getProductID()` | ✓ | ✓ | ✓ |
| `int32_t getVendorID()` | ✓ | ✓ | ✓ |
| `int32_t getVersionNumber()` | ✓ | ✓ | ✗ |
| `DMResult getDevicePath(wchar_t* path, int32_t len)` | ✓ | ✗ | ✓ |
| `DMResult getInternalName(wchar_t* name, int32_t len)` | ✓ | ✓ | ✓ |
| `DMResult getProductName(wchar_t* name, int32_t len)` | ✓ | ✓ | ✓ |
| `DMResult getManufacturerName(wchar_t* name, int32_t len)` | ✓ | ✓ | ✓ |
| `DMResult getSerialNumber(wchar_t* serialNumber, int32_t len)` | ✓ | ✓ | ✓ |
| `bool isAttached()` | ✓ | ✗ | ✓ |
| `bool attach()` | ✓ | ✗ | ✗ |
| `bool detach()` | ✓ | ✗ | ✗ |
| `bool isSupported(uint16_t usage)` | ✓ | ✗ | ✗ |
| `DMResult getDeviceAttributes(IDeviceAttributes **ppAttributes)` | ✓ | ✗ | ✗ |
| `DMResult getHostCommand(ppHostCommand ppCommand)` | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| `DMResult getDeviceEvents(ppDeviceEvents ppEvents)` | ✓ | ✗ | ✓ |
| `DMResult getDeviceListener(ppDeviceListener ppListener)` | ✓ | ✓ | ✓ (indirectly) |
| `DMResult setOption(eDeviceOption eDOption, int32_t lValue)` | ✓ | ✗ | ✗ |
| `DMResult getOption(eDeviceOption eDOption, int32_t* plValue, int16_t iMaxLength, int16_t* piLength)` | ✓ | ✗ | ✗ |
| `DMResult registerCallback(pDeviceCallback pDevCallbacks)` | ✓ | ✗ | ✗ |
| `DMResult unregisterCallback(pDeviceCallback pDevCallbacks)` | ✓ | ✗ | ✗ |
| `DMResult getDeviceType(uint32_t *io_cntType, DeviceType *o_pType)` | ✓ | ✗ | ✗ |
| `IDeviceManager *getDeviceManager()` | ✓ | ✗ | ✗ |

## 9.12: IDeviceManager Interface

For descriptions of these functions, read **Error! Reference source not found. Error! Reference source not found.**.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult getDevice(eVendorID vendorID, eProductID productID, ppDevice ppDev)` | ✓ | ✗ | ✗ |
| `DMResult getDevice(eVendorID vendorID, ppDevice ppDev)` | ✓ | ✗ | ✗ |

| | | | |
|---|---|---|---|
| DMResult getDevice(const wchar_t* devicePath, ppDevice ppDev) | ✓ | ✗ | ✗ |
| DMResult getDevices(IDeviceGroup** devices) | ✓ | ✗ | ✗ |
| DMResult getHIDDevices(uint16_t i_usagePage, IDeviceGroup** devices) | ✓ | ✗ | ✗ |
| DMResult getDevices(eVendorID vendorID, IDeviceGroup** devices) | ✓ | ✗ | ✗ |
| DMResult getDevices(eVendorID vendorID, eProductID productID, IDeviceGroup** devices) | ✓ | ✗ | ✗ |
| DMResult setOption(eDMOption eDOption, int32_t lValue) | ✓ | ✗ | ✗ |
| DMResult getOption(eDMOption eDOption, int32_t* plValue, int16_t iMaxLength, int16_t* piLength) | ✓ | ✗ | ✗ |
| DMResult registerCallback(pDeviceManagerCallback pDMCallbacks) | ✓ | ✗ | ✗ |
| DMResult unregisterCallback(pDeviceManagerCallback pDMCallbacks) | ✓ | ✗ | ✗ |
| DMResult registerCallback(IGenericDeviceCallback *pDMCallbacks) | ✓ | ✗ | ✗ |
| DMResult unregisterCallback(IGenericDeviceCallback *pDMCallbacks) | ✓ | ✗ | ✗ |
| DMResult setDeviceFilterEnabled(bool) | ✓ | ✗ | ✗ |

## 9.13: IDeviceBaseExt Interface

For descriptions of these functions, read section 3.1: IDeviceBaseExt Interface.IDeviceBaseExt Interface

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `virtual DMResult getAllConnectedDevices()` | ✓ | ✗ | ✗ |

## 9.14: IHostCommand Interface

For descriptions of these functions, read section 5.1: IHostCommand Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `IHostCommandQuery *getQuery()` | ✓ | ✗ | ✗ |
| `DMResult setRing(bool bEnable)` | ✓ | ✓ | ✗ |
| `DMResult getAudioState(eAudioState &state)` | ✓ | ✓ | ✗ |
| `DMResult setAudioState(eAudioState audioType)` | ✓ | ✓ | ✗ |
| `DMResult getMute(bool &mute)` | ✓ | ✓ | ✗ |
| `DMResult registerCommand(eRegistrationType sign)` | ✓ | ✓ | ✗ |
| `DMResult getVersion(eVersionType version, wchar_t* buffer, uint32_t bufferLength)` | ✓ | ✓ | ✗ |
| `bool isSupported(eFeatureType feature)` | ✓ | ✗ | ✗ |

## 9.15: IHostCommandOption Interface

For descriptions of these functions, read section 5.3: IHostCommandOption Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `virtual DMResult setOption(eCommandOption eCOption, int32_t lValue)` | ✓ | ✓ | ✗ |
| `virtual DMResult getOption(eCommandOption eCOption, int32_t* plValue, int16_t iMaxLength, int16_t* piLength)` | ✓ | ✓ | ✗ |

## 9.16: IHostCommandExt Interface

For descriptions of these functions, read section 5.2: IHostCommandExt Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult getBatteryLevel(eBatteryLevel &level)` | ✓ | ✓ | ✗ |
| `DMResult getHeadsetsInConference(int32_t &count)` | ✓ | ✓ | ✗ |
| `DMResult setHeadsetMute(bool bMute)` | ✓ | ✓ | ✗ |
| `DMResult getHeadsetType(eHeadsetType &hs)` | ✓ | ✓ | ✗ |
| `DMResult sendVersionRequest()` | ✓ | ✗ | ✗ |
| `DMResult setActiveLink(eLineType lineType, bool bActive)` | ✓ | ✓ | ✗ |
| `DMResult isLineActive(eLineType i_lineType, bool &o_active)` | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| DMResult isLineConnected(eLineType i_lineType, bool &o_active) | ✓ | ✓ | ✗ |
| DMResult hold(eLineType lineType, bool bHold) | ✓ | ✓ | ✗ |
| DMResult getHoldState(eLineType i_lineType, bool &o_hold) | ✓ | ✓ | ✗ |
| DMResult getAudioLinkState(eAudioLinkState &state) | ✓ | ✓ | ✗ |
| DMResult isHeadsetDocked(bool& isHsDocked) | ✓ | ✓ | ✗ |
| DMResult getHeadsetProductName(wchar_t* buffer, int32_t length) | ✓ | ✓ | ✗ |
| DMResult getAudioLocation(eAudioLocation &location) | ✓ | ✓ | ✗ |
| DMResult getBatteryInfo(eBTBatteryStatus* btBatteryStatus, int32_t* life, eBTChargingStatus* btChargingStatus) | ✓ | ✓ | ✗ |
| DMResult sendSerialNumberReq(Plantronics::DM::eDeviceType devType) | ✓ | ✓ | ✗ |
| DMResult getSerialNumber(Plantronics::DM::eDeviceType devType, uint8_t* serialNumber, int32_t serialNumberBufferSize) | ✓ | ✓ | ✗ |
| bool isConstant(uint16_t usage) | ✓ | ✗ | ✗ |
| DMResult getProximity() | ✓ | ✓ | ✗ |
| DMResult enableProximity(bool bEnable) | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| `DMResult getHeadsetState(eHeadsetState &state)` | ✓ | ✓ | ✗ |
| `DMResult HeadsetPairing(bool on)` | ✓ | ✗ | ✗ |

## 9.17: IATDCommand Interface

For descriptions of these functions, read section 5.6: IATDCommand Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult AnswerMobileCall()` | ✓ | ✓ | ✗ |
| `DMResult EndMobileCall()` | ✓ | ✓ | ✗ |
| `DMResult Redial()` | ✓ | ✓ | ✗ |
| `DMResult MakeMobileCall(wchar_t const *callerId, int32_t length)` | ✓ | ✓ | ✗ |
| `DMResult GetMobileCallStatus()` | ✓ | ✓ | ✗ |
| `wchar_t const *getCallerId()` | ✓ | ✓ | ✗ |
| `DMResult MuteMobileCall(bool mute)` | ✓ | ✓ | ✗ |

## 9.18: IHostCommandQuery Interface

Only for C++. For descriptions of these functions, read section 5.4: IHostCommandQuery Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `bool query(IHostCommand **p)` | ✓ | ✗ | ✗ |
| `bool query(IHostCommandExt **p)` | ✓ | ✗ | ✗ |
| `bool query(IDeviceSettings  **p)` | ✓ | ✗ | ✗ |
| `bool query(IAdvanceSettings **p)` | ✓ | ✗ | ✗ |
| `bool query(IDeviceExtendedInfo **p)` | ✓ | ✗ | ✗ |

## 9.19: IDeviceSettingsExt Interface

For descriptions of these functions, read section 5.8: IDeviceSettingsExt Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult getAcousticsReporting(eAcousticsReportType type, bool& value) const` | ✓ | ✓ | ✗ |
| `DMResult setAcousticsReporting(eAcousticsReportType type, bool value)` | ✓ | ✓ | ✗ |
| `DMResult getAALReportingThreshold(AALReportingThreshold& value) const` | ✓ | ✓ | ✗ |
| `DMResult setAALReportingThreshold( AALReportingThreshold value)` | ✓ | ✓ | ✗ |
| `DMResult getTWAReportingPeriod(uint32_t& value) const` | ✓ | ✓ | ✗ |
| `DMResult setTWAReportingPeriod(uint32_t value)` | ✓ | ✓ | ✗ |
| `DMResult getConversationDynamicsPeriod(uint32_t& value) const` | ✓ | ✓ | ✗ |
| `DMResult setConversationDynamicsPeriod(uint32_t value)` | ✓ | ✓ | ✗ |
| `DMResult getPartitionInfo(uint16_t & partition, uint16_t & position, uint16_t & version, uint16_t & number)` | ✓ | ✓ | ✗ |
| `DMResult removePartition(const uint16_t&)` | ✓ | ✓ | ✗ |
| `DMResult setLanguage(uint16_t lang)` | ✓ | ✓ | ✗ |
| `DMResult getLanguage(uint16_t & lang)` | ✓ | ✓ | ✗ |
| `DMResult getAnswerIgnoreVoiceCommand(bool & on)` | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| DMResult setAnswerIgnoreVoiceCommand(bool on) | ✓ | ✓ | ✗ |
| DMResult getCallerNameAnnouncement(bool & on) | ✓ | ✓ | ✗ |
| DMResult setCallerNameAnnouncement(bool on) | ✓ | ✓ | ✗ |
| DMResult getMuteOffVoicePrompt(bool & on) | ✓ | ✓ | ✗ |
| DMResult setMuteOffVoicePrompt(bool on) | ✓ | ✓ | ✗ |
| DMResult getMutePromptInterval(uint16_t & interval) | ✓ | ✓ | ✗ |
| DMResult setMutePromptInterval(uint16_t interval) | ✓ | ✓ | ✗ |
| DMResult getCallButtonLock(bool & on) | ✓ | ✓ | ✗ |
| DMResult setCallButtonLock(bool on) | ✓ | ✓ | ✗ |
| DMResult getSupportedLanguages(uint16_t *, unsigned int & capacity) | ✓ | ✓ | ✗ |
| DMResult setSignalStrengthEvents(SignalStrengthParams param) | ✓ | ✓ | ✗ |
| DMResult getCurrentSignalStrengthEvents(uint8_t id, uint8_t& conId, uint8_t& strength, SignalStrengthType& type) | ✓ | ✓ | ✗ |
| DMResult setPairingMode(bool enable) | ✓ | ✓ | ✗ |
| DMResult getPairingMode(bool& enable) | ✓ | ✓ | ✗ |
| DMResult getConnectionStatus(std::vector<uint8_t>& downPort, std::vector<uint8_t>& connectedPort, uint8_t& origPort) | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| DMResult getSignalStrengthConfiguration(uint8_t connId, SignalStrengthParams& param) | ✓ | ✓ | ✗ |
| DMResult getAudioStatus(uint8_t& codec, uint8_t& port, uint8_t& speakerGain, uint8_t& micGain) | ✓ | ✓ | ✗ |
| DMResult getSCOTone(bool & tone) | ✓ | ✓ | ✗ |
| DMResult setSCOTone(bool tone) | ✓ | ✓ | ✗ |
| DMResult setFindHeadsetLEDAlert(bool enable) | ✓ | ✓ | ✗ |
| DMResult getFindHeadsetLEDAlert(bool& enabled) | ✓ | ✓ | ✗ |
| DMResult setAudioTransmitGain(uint8_t gain) | ✓ | ✓ | ✗ |
| DMResult setSpeakerVolume(bool action, uint16_t volume) | ✓ | ✓ | ✗ |
| DMResult getWearingSensorEnabled(bool& enabled) | ✓ | ✓ | ✗ |
| DMResult setWearingSensorEnabled(bool enabled) | ✓ | ✓ | ✗ |
| DMResult setVRCallRejectAnswer(bool enable) | ✓ | ✓ | ✗ |
| DMResult getVRCallRejectAnswer(bool& enabled) | ✓ | ✓ | ✗ |
| DMResult getHeadsetConnectedState(bool& enabled) const | ✓ | ✓ | ✗ |
| DMResult getHTopSelector(eHTopSelectorType& value) const | ✓ | ✓ | ✗ |
| DMResult setHTopSelector(eHTopSelectorType value) | ✓ | ✓ | ✗ |
| DMResult setAutoMuteCall(bool enabled) | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| `DMResult getAutoMuteCall(bool& enabled) const` | ✓ | ✓ | ✗ |
| `DMResult setMuteAlert(eMuteAlertValues value)` | ✓ | ✓ | ✗ |
| `DMResult getMuteAlert(eMuteAlertValues &value) const` | ✓ | ✓ | ✗ |
| `DMResult getVoiceSilenceDetectionMode( eVoiceSilenceDetectionMode& mode) const` | ✓ | ✓ | ✗ |
| `DMResult setOLIFeature(bool enabled)` | ✓ | ✓ | ✗ |
| `DMResult getOLIFeature(bool& enabled) const` | ✓ | ✓ | ✗ |

## 9.20: IDeviceListener Interface

For descriptions of these functions, read section 4.1: IDeviceListener Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `IDeviceListenerQuery *getQuery()`<br>*Only for C++* | ✓ | ✗ | ✗ |
| `DMResult ring(bool enable)` | ✓ | ✓ | ✓ |
| `DMResult getAudioState(eAudioState &state)` | ✓ | ✓ | ✓ |
| `DMResult setAudioState(eAudioState value)` | ✓ | ✓ | ✗ |
| `DMResult getMute(bool &mute)` | ✓ | ✓ | ✗ |
| `DMResult setMute(bool value)` | ✓ | ✓ | ✗ |
| `DMResult getIntellistand(bool &enabled)` | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| DMResult setIntellistand(bool value) | ✓ | ✓ | ✗ |
| DMResult getAudioMixerState(int32_t &state) | ✓ | ✓ | ✗ |
| DMResult setAudioMixerState(int32_t value) | ✓ | ✓ | ✗ |
| DMResult getDefaultLine(eLineType &line) | ✓ | ✓ | ✗ |
| DMResult setDefaultLine(eLineType value) | ✓ | ✓ | ✗ |
| DMResult inCall(bool bOn) | ✓ | ✓ | ✗ |
| DMResult suppressDialTone(bool bOn) | ✓ | ✗ | ✗ |
| DMResult setActiveLink(eLineType lineType, bool bActive) | ✓ | ✓ | ✗ |
| DMResult isLineActive(eLineType lineType, bool &active) | ✓ | ✓ | ✗ |
| DMResult hold(eLineType lineType, bool bHold) | ✓ | ✓ | ✗ |
| DMResult getHoldState(eLineType lineType, bool &hold) | ✓ | ✓ | ✗ |
| DMResult getAudioLinkState(eAudioLinkState &state) | ✓ | ✓ | ✗ |
| DMResult getAudioLocation(eAudioLocation &location) | ✓ | ✓ | ✗ |
| DMResult registerCallback(IDeviceListenerCallback*) | ✓ | ✓ | ✗ |
| DMResult unregisterCallback(IDeviceListenerCallback*) | ✓ | ✓ | ✗ |

## 9.21: IDisplayDeviceListener Interface

For descriptions of these functions, read section 0 See Also: Section 4.1.1.1: DMResult enum.

IDisplayDeviceListener Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult setLocale(uint32_t lcid)` | ✓ | ✗ | ✗ |
| `DMResult setPresence(DDSoftphoneID spID, DDPresence presence)` | ✓ | ✗ | ✗ |
| `DMResult setDefaultSoftphone(DDSoftphoneID spID)` | ✓ | ✗ | ✗ |
| `DMResult setDateTime(DM_DateTime const& dt)` | ✓ | ✗ | ✗ |
| `DMResult setDateTimeFormat(uint32_t lcid)` | ✓ | ✗ | ✗ |
| `DMResult setSoftphoneName(DDSoftphoneID spID, wchar_t const *spName)` | ✓ | ✗ | ✗ |
| `DMResult setMultiCallState(int32_t nCount, IDisplayDeviceCall **ddCall)` | ✓ | ✗ | ✗ |
| `DDSoftphoneID getSoftphoneID()` | ✓ | ✗ | ✗ |
| `wchar_t const *getMakeCallName()` | ✓ | ✗ | ✗ |

## 9.22: IDisplayDeviceCall Interface

For descriptions of these functions, read section 4.4: IDisplayDeviceCall Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DM_GUID getSessionId()` | ✓ | ✗ | ✗ |

| | | | |
|---|---|---|---|
| `DMResult setSessionId(DM_GUID guid)` | ✓ | ✗ | ✗ |
| `int32_t getCallId()` | ✓ | ✗ | ✗ |
| `DMResult setCallId(int32_t callId)` | ✓ | ✗ | ✗ |
| `DDCallState getCallState()` | ✓ | ✗ | ✗ |
| `DMResult setCallState(DDCallState callState)` | ✓ | ✗ | ✗ |
| `DDSoftphoneID getSoftphoneID()` | ✓ | ✗ | ✗ |
| `DMResult setSoftphoneID(DDSoftphoneID spID)` | ✓ | ✗ | ✗ |
| `wchar_t const *getCallerId()` | ✓ | ✗ | ✗ |
| `DMResult setCallerId(wchar_t const *callerId)` | ✓ | ✗ | ✗ |
| `wchar_t const *getFriendlyName()` | ✓ | ✗ | ✗ |
| `DMResult setFriendlyName(wchar_t const *name)` | ✓ | ✗ | ✗ |

## 9.23: IDeviceListenerCallback Interface

For descriptions of these functions, read section0See Also: Section 4.1.1.1: DMResult enum.

IDeviceListenerCallback Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `void onHeadsetButtonPressed(DeviceListenerEventArgs const&)` | ✓ | ✓ | ✓ |

| | | | |
|---|---|---|---|
| `void onHeadsetStateChanged(DeviceListenerEventArgs const&)` | ✓ | ✓ | ✓ |
| `void onBaseButtonPressed(DeviceListenerEventArgs const&)` | ✓ | ✓ | ✓ |
| `void onBaseStateChanged(DeviceListenerEventArgs const&)` | ✓ | ✓ | ✓ |
| `void onATDStateChanged(DeviceListenerEventArgs const&)` | ✓ | ✓ | ✓ |

## 9.24: IDeviceEvents Interface

For descriptions of these functions, read section 6.1: IDeviceEvents Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `IDeviceEventsQuery *getQuery()` | ✓ | ✗ | ✗ |
| `DMResult regCbDeviceEvents(IDeviceEventsCallback*)` | ✓ | ✓ | ✗ |
| `DMResult unregCbDeviceEvents(IDeviceEventsCallback*)` | ✓ | ✓ | ✗ |
| `DMResult setInputReport(uint8_t const * reportBuffer, int32_t length)` | ✓ | ✗ | ✗ |

## 9.25: IDeviceEventsQuery Interface

For descriptions of these functions, read section *6.5:* IDeviceEventsQuery Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|

| | | | |
|---|---|---|---|
| `bool query(IDeviceEvents **p)` | ✓ | ✗ | ✗ |
| `bool query(IDeviceEventsExt **p)` | ✓ | ✗ | ✗ |
| `bool query(IBaseEvents **p)` | ✓ | ✗ | ✗ |
| `bool query(IHIDPipeEvents **p)` | ✓ | ✗ | ✗ |
| `bool query(IMobilePresenceEvents **p)` | ✓ | ✗ | ✗ |
| `bool query(IMOCEventsCallback **p)` | ✓ | ✗ | ✗ |
| `bool query(IMOCEvents **p)` | ✓ | ✗ | ✗ |
| `bool query(IMOCEventsExtCallback **p)` | ✓ | ✗ | ✗ |
| `bool query(IMOCEventsExt **p)` | ✓ | ✗ | ✗ |
| `bool query(IAudioSenseEventsCallback **p)` | ✓ | ✗ | ✗ |
| `bool query(IAudioSenseEvents **p)` | ✓ | ✗ | ✗ |

## 9.26: IDeviceEventsExt Interface

For descriptions of these functions, read section 6.3: IDeviceEventsExt Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult regCbDeviceEventsExt(IDeviceEventsExtCallback*)` | ✓ | ✓ | ✗ |
| `DMResult unregCbDeviceEventsExt(IDeviceEventsExtCallback*)` | ✓ | ✓ | ✗ |

## 9.27: IBaseEvents Interface

For descriptions of these functions, read section 6.6*:* IBaseEvents Interface*.*

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult regCbBaseEvents(IBaseEventsCallback*)` | ✓ | ✓ | ✗ |
| `DMResult unregCbBaseEvents(IBaseEventsCallback*)` | ✓ | ✓ | ✗ |

## 9.28: IMobilePresenceEvents Interface

For descriptions of these functions, read section 6.8*:* IMobilePresenceEvents Interface*.*

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult regCbMobilePresenceEvents(IMobilePresenceEventsCallback*)` | ✓ | ✓ | ✗ |
| `DMResult unregCbMobilePresenceEvents(IMobilePresenceEventsCallback*)` | ✓ | ✓ | ✗ |

## 9.29: IDeviceAttributes Interface

This interface provides metadata interface about the USB HID device. For descriptions of these function, read section 2.11: IDeviceAttributes Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DeviceCaps const* GetDeviceCaps()` | ✓ | ✗ | ✗ |
| `ICapabilities const* GetInputCaps()` | ✓ | ✗ | ✗ |
| `ICapabilities const* GetOutputCaps()` | ✓ | ✗ | ✗ |

| | | | |
|---|---|---|---|
| `ICapabilities const* GetFeatureCaps()` | ✓ | ✗ | ✗ |
| `int32_t GetDelayTime()` | ✓ | ✗ | ✗ |
| `void SetDelayTime(int32_t value)` | ✓ | ✗ | ✗ |
| `bool HasUsage(USAGE usage)` | ✓ | ✗ | ✗ |
| `bool HasUsage(USAGE usagePage, USAGE usage)` | ✓ | ✗ | ✗ |
| `bool HasBtnUsage(USAGE usage, ReportType enumReportType)` | ✓ | ✗ | ✗ |
| `bool HasValUsage(USAGE usage, ReportType enumReportType)` | ✓ | ✗ | ✗ |
| `bool HasBtnUsage(USAGE usagePage, USAGE usage, ReportType enumReportType)` | ✓ | ✗ | ✗ |
| `bool HasValUsage(USAGE usagePage, USAGE usage, ReportType enumReportType)` | ✓ | ✗ | ✗ |
| `IDeviceUsage* GetDeviceUsage(ReportType eReportType)` | ✓ | ✗ | ✗ |
| `bool IsConstant(USAGE usage)` | ✓ | ✗ | ✗ |

## 9.30: IDeviceUsage Interface

This interface allows you to set and get HID output and feature usage from a USB HID device. For descriptions of these function, read section 2.12: IDeviceUsage Class.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `void AddUsage(const Usage& usage)` | ✓ | ✗ | ✗ |

| | | | |
|---|---|---|---|
| `void AddUsage(uint16_t usagePage, uint16_t usage)` | ✓ | ✗ | ✗ |
| `void AddUsage(uint16_t usagePage, uint16_t usage, unsigned long value)` | ✓ | ✗ | ✗ |
| `Usage const* FindUsage(const USAGE usage)` | ✓ | ✗ | ✗ |
| `DMResult SetUsage()` | ✓ | ✗ | ✗ |
| `DMResult SetOutputUsage(const Usage& usage)` | ✓ | ✗ | ✗ |
| `DMResult SetOutputUsage(uint16_t usagePage, uint16_t usage)` | ✓ | ✗ | ✗ |
| `DMResult SetOutputUsage(uint16_t usagePage, uint16_t usage, unsigned long value)` | ✓ | ✗ | ✗ |
| `DMResult SetOutputUsage(uint8_t reportID, uint16_t usagePage, uint16_t usage, unsigned long value)` | ✓ | ✗ | ✗ |
| `DMResult SetValue(int32_t val)` | ✓ | ✗ | ✗ |
| `DMResult SetValue(const Usage& usage)` | ✓ | ✗ | ✗ |
| `DMResult SetOutputValue(const Usage& usage)` | ✓ | ✗ | ✗ |
| `DMResult SetOutputValue(uint16_t usagePage, uint16_t usage)` | ✓ | ✗ | ✗ |
| `DMResult SetOutputValue(uint16_t usagePage, uint16_t usage, unsigned long value)` | ✓ | ✗ | ✗ |
| `DMResult GetReportID(uint16_t i_usagePage, uint16_t i_usage, uint8_t &o_reportID)` | ✓ | ✗ | ✗ |

| | | | |
|---|---|---|---|
| `DMResult GetReportID(uint16_t i_usage, uint8_t &o_reportID)` | ✓ | ✗ | ✗ |
| `int32_t GetMaxUsageListLength(USAGE usagePage)` | ✓ | ✗ | ✗ |
| `DMResult GetUsages(USAGE usagePage, USAGE usages[], uint32_t& maxUsageListLen)` | ✓ | ✗ | ✗ |
| `DMResult GetUsages(USAGE usagePage, USAGE usages[], uint32_t& maxUsageListLen, uint8_t const * pReportBuffer, int32_t reportBufferSize)` | ✓ | ✗ | ✗ |
| `DMResult GetUsage()` | ✓ | ✗ | ✗ |
| `uint32_t GetValue()` | ✓ | ✗ | ✗ |
| `DMResult GetValue(USAGE usagePage, USAGE usage, uint32_t &data)` | ✓ | ✗ | ✗ |
| `DMResult GetValue(USAGE usagePage, USAGE usage, uint8_t* buffer, int32_t bufferSize, uint32_t& data)` | ✓ | ✗ | ✗ |
| `DMResult GetValueArray(uint8_t* values, uint16_t len)` | ✓ | ✗ | ✗ |
| `DMResult GetValueArray(uint16_t usagePage, USAGE usage, uint8_t* values, uint16_t len)` | ✓ | ✗ | ✗ |
| `uint32_t GetValueArraySize()` | ✓ | ✗ | ✗ |
| `uint32_t GetValueArraySize(uint16_t usagePage, USAGE usage)` | ✓ | ✗ | ✗ |
| `DMResult GetData(USAGE usagePage, USAGE usage, uint32_t& data)` | ✓ | ✗ | ✗ |

| | | | |
|---|---|---|---|
| DMResult GetData(USAGE usagePage, USAGE usage, uint8_t const * buffer, int32_t bufferSize, uint32_t& data) | ✓ | ✗ | ✗ |
| DMResult GetMakeList(pUSAGE previousUsageList, pUSAGE currentUsageList, pUSAGE makeList, int32_t usageListLen) | ✓ | ✗ | ✗ |
| DMResult GetBreakList(pUSAGE previousUsageList, pUSAGE currentUsageList, pUSAGE breakList, int32_t usageListLen) | ✓ | ✗ | ✗ |
| DMResult GetMakeBreakList(pUSAGE previousUsageList, pUSAGE currentUsageList, pUSAGE makeList, pUSAGE breakList, uint32_t usageListLen) | ✓ | ✗ | ✗ |
| DMResult WriteReport() | ✓ | ✗ | ✗ |
| DMResult WriteReport(uint8_t const *buffer, int32_t len) | ✓ | ✗ | ✗ |
| DMResult WriteReportRaw(uint8_t const *buffer, int32_t len) | ✓ | ✗ | ✗ |
| DMResult WriteReportCT(uint8_t const *buffer, int32_t len) | ✓ | ✗ | ✗ |
| DMResult WriteReportWithDelay() | ✓ | ✗ | ✗ |
| DMResult GetReport(uint8_t** reportBuffer, uint32_t *length) | ✓ | ✗ | ✗ |
| DMResult ReadFeature(uint8_t* buffer, int32_t len) | ✓ | ✗ | ✗ |
| DMResult ReadFeature() | ✓ | ✗ | ✗ |

| | | | |
|---|---|---|---|
| `DMResult ReadInput(uint8_t* buffer, int32_t len)` | ✓ | ✗ | ✗ |
| `DMResult ReadInput()` | ✓ | ✗ | ✗ |

## 9.31: IDeviceSettings Interface

The *IDeviceSettings* interface is used to read and set device configuration parameters. For descriptions of these function, read section 0See Also: Section 4.1.1.1: DMResult enum.

IDeviceSettings Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult getVOIPRing(eDeviceRingTone &tone)` | ✓ | ✓ | ✗ |
| `DMResult setVOIPRing(eDeviceRingTone value)` | ✓ | ✓ | ✗ |
| `DMResult getVOIPBandwidth(eAudioBandwidth &audio)` | ✓ | ✓ | ✗ |
| `DMResult setVOIPBandwidth(eAudioBandwidth value)` | ✓ | ✓ | ✗ |
| `DMResult getVOIPRingVolume(eVolumeLevel &level)` | ✓ | ✓ | ✗ |
| `DMResult setVOIPRingVolume(eVolumeLevel value)` | ✓ | ✓ | ✗ |
| `DMResult getVOIPToneControl(eToneLevel &level)` | ✓ | ✓ | ✗ |
| `DMResult setVOIPToneControl(eToneLevel value)` | ✓ | ✓ | ✗ |
| `DMResult getPSTNRing(eDeviceRingTone& value)` | ✓ | ✓ | ✗ |
| `DMResult setPSTNRing(eDeviceRingTone value)` | ✓ | ✓ | ✗ |
| `DMResult getPSTNBandwidth(eAudioBandwidth& value)` | ✓ | ✓ | ✗ |
| `DMResult setPSTNBandwidth(eAudioBandwidth value)` | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| DMResult getPSTNRingVolume(eVolumeLevel& value) | ✓ | ✓ | ✗ |
| DMResult setPSTNRingVolume(eVolumeLevel value) | ✓ | ✓ | ✗ |
| DMResult getPSTNToneControl(eToneLevel &level) | ✓ | ✓ | ✗ |
| DMResult setPSTNToneControl(eToneLevel value) | ✓ | ✓ | ✗ |
| DMResult getMobileRing(eDeviceRingTone &tone) | ✓ | ✓ | ✗ |
| DMResult setMobileRing(eDeviceRingTone value) | ✓ | ✓ | ✗ |
| DMResult getMobileBandwidth(eAudioBandwidth &audio) | ✓ | ✓ | ✗ |
| DMResult setMobileBandwidth(eAudioBandwidth value) | ✓ | ✓ | ✗ |
| DMResult getMobileRingVolume(eVolumeLevel &level) | ✓ | ✓ | ✗ |
| DMResult setMobileRingVolume(eVolumeLevel value) | ✓ | ✓ | ✗ |
| DMResult getMuteTone(eDeviceRingTone &tone) | ✓ | ✓ | ✗ |
| DMResult setMuteTone(eDeviceRingTone value) | ✓ | ✓ | ✗ |
| DMResult getToneVolume(eVolumeLevel &level) | ✓ | ✓ | ✗ |
| DMResult setToneVolume(eVolumeLevel value) | ✓ | ✓ | ✗ |
| DMResult getMuteToneVolume(eVolumeLevel &level) | ✓ | ✓ | ✗ |
| DMResult setMuteToneVolume(eVolumeLevel value) | ✓ | ✓ | ✗ |
| DMResult getActiveCallRing(eActiveCallRing &ring) | ✓ | ✓ | ✗ |
| DMResult setActiveCallRing(eActiveCallRing value) | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| DMResult getAntiStartleEnabled(bool& value) | ✓ | ✓ | ✗ |
| DMResult setAntiStartleEnabled(bool value) | ✓ | ✓ | ✗ |
| DMResult getAudioLimit(eAudioLimit& value) | ✓ | ✓ | ✗ |
| DMResult setAudioLimit(eAudioLimit value) | ✓ | ✓ | ✗ |
| DMResult getG616Enabled(bool& value) | ✓ | ✓ | ✗ |
| DMResult setG616Enabled(bool value) | ✓ | ✓ | ✗ |
| DMResult getOTAEnabled(bool &enabled) | ✓ | ✓ | ✗ |
| DMResult setOTAEnabled(bool value) | ✓ | ✓ | ✗ |
| DMResult getIntellistandEnabled(bool &enabled) | ✓ | ✓ | ✗ |
| DMResult setIntellistandEnabled(bool value) | ✓ | ✓ | ✗ |
| DMResult getPowerMode(ePowerLevel &level) | ✓ | ✓ | ✗ |
| DMResult setPowerMode(ePowerLevel value) | ✓ | ✓ | ✗ |
| DMResult getData(eCustomData dataType) | ✓ | ✗ | ✗ |
| DMResult setData(eCustomData dataType, uint8_t* buffer, int32_t bufferLength) | ✓ | ✓ | ✗ |
| DMResult isPasswordProtected(bool &protec) | ✓ | ✓ | ✗ |
| DMResult getDECTMode(bool &mode) | ✓ | ✓ | ✗ |
| DMResult getTWAPeriod(eTWAPeriod &period) | ✓ | ✓ | ✗ |
| DMResult setTWAPeriod(eTWAPeriod value) | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| DMResult getPhoneLine(eLineType &type) | ✓ | ✓ | ✗ |
| DMResult setPhoneLine(eLineType value) | ✓ | ✓ | ✗ |
| DMResult getBTInterfaceEnabled(bool &enabled) | ✓ | ✓ | ✗ |
| DMResult setBTInterfaceEnabled(bool value) | ✓ | ✓ | ✗ |
| DMResult getBTAutoConnectEnabled(bool &enabled) | ✓ | ✓ | ✗ |
| DMResult setBTAutoConnectEnabled(bool value) | ✓ | ✓ | ✗ |
| DMResult restoreDefaultSettings() | ✓ | ✓ | ✗ |
| DMResult getACLPollingEnabled(bool &enabled) | ✓ | ✓ | ✗ |
| DMResult setACLPollingEnabled(bool value) | ✓ | ✓ | ✗ |
| DMResult getBTVoiceCommandEnabled(bool &enabled) | ✓ | ✓ | ✗ |
| DMResult setBTVoiceCommandEnabled(bool value) | ✓ | ✓ | ✗ |
| DMResult getVolumeControlOrientation(eVolumeControlOrientation& value) | ✓ | ✗ | ✗ |
| DMResult setVolumeControlOrientation(eVolumeControlOrientation value) | ✓ | ✗ | ✗ |
| DMResult getA2DPEnabled(bool& value) | ✓ | ✗ | ✗ |
| DMResult setA2DPEnabled(bool value) | ✓ | ✗ | ✗ |

## 9.32: IAdvanceSettings Interface

Use the *IAdvanceSettings* interface to discover and specify advanced settings on the host device. For descriptions of these function, read section 5.5: IAdvanceSettings Interface.

| Feature Name | Native | COM | REST |
|---|---|---|---|
| `DMResult getDialTone(bool& value)` | ✓ | ✓ | ✗ |
| `DMResult setDialTone(bool value)` | ✓ | ✓ | ✗ |
| `DMResult getAudioSensing(bool& value)` | ✓ | ✓ | ✗ |
| `DMResult setAudioSensing(bool value)` | ✓ | ✓ | ✗ |
| `DMResult getDialToneActive(bool& value)` | ✓ | ✓ | ✗ |
| `DMResult getAnswerOnDon(bool& bEnable)` | ✓ | ✓ | ✗ |
| `DMResult setAnswerOnDon(bool value)` | ✓ | ✓ | ✗ |
| `DMResult getAutoTransfer(eSensorControl& value)` | ✓ | ✓ | ✗ |
| `DMResult setAutoTransfer(eSensorControl value)` | ✓ | ✓ | ✗ |
| `DMResult getAutoDisconnect(eSensorControl& value)` | ✓ | ✓ | ✗ |
| `DMResult setAutoDisconnect(eSensorControl value)` | ✓ | ✓ | ✗ |
| `DMResult getAutoReject(eSensorControl& value)` | ✓ | ✓ | ✗ |
| `DMResult setAutoReject(eSensorControl value)` | ✓ | ✓ | ✗ |

| | | | |
|---|---|---|---|
| `DMResult getLockHookswitch(eSensorControl& value)` | ✓ | ✓ | ✗ |
| `DMResult setLockHookswitch(eSensorControl value)` | ✓ | ✓ | ✗ |
| `DMResult getAutoPause(eSensorControl& value)` | ✓ | ✓ | ✗ |
| `DMResult setAutoPause(eSensorControl value)` | ✓ | ✓ | ✗ |
| `DMResult getVoicePrompt(eSensorControl& value)` | ✓ | ✓ | ✗ |
| `DMResult setVoicePrompt(eSensorControl value)` | ✓ | ✓ | ✗ |

# Chapter 10: Appendix B: Plantronics SDK Call handling and Softphone interaction

The following sections describe in more detail what the behavior of the Call Manager will be under the circumstances described in the heading.

The following behavior is described:

- A Single Incoming Call on a Softphone
- A Single Incoming Call on a SoftPhone with Auto-Answer Enabled
- Single Outgoing Call on a Softphone
- Device-Initiated Outgoing Call
- Single Softphone Call using Hold and Resume
- Single Softphone with Multiple Incoming Calls
- Multiple Softphones with Multiple Incoming Calls
- Media Player Interaction
- Softphone Interaction with Multi-Channel Device (VoIP/PSTN/Mobile)

## 10.1: A Single Incoming Call on a Softphone

### *10.1.1: Incoming Call Notification from Softphone*

Softphone  module calls Plantronics SDK API to render call after receiving incoming call notification from the Softphone.

```
ICallCommand.incomingCall(callID, contact, tones, route)
```

Call Manager creates a new Call instance, requests the Device Listener to render incoming call, and enters into Ringing state.

- Call answered by Talk button press:

User responds with a Talk button press and the CM brings up the audio link (Wireless only). As the device responds back with Audio link ON, Spokes notifies the SP component to accept the call.

```
ICallCommand.answeredCall(callID)
```

Call Manager at this point knows that a SP call is active.

- Spokes Call State: Active Call
- Call ended by Talk button press:

Device Listener notifies Call Manager about Talk press. Spokes drops the active call and brings down the audio Link (Wireless only). Spokes notifies the SP module with a terminateCall command and the Softphone call terminates.

```
ICallCommand.terminateCall(callID)
```

CSM enters into an idle state waiting for new incoming calls.

- Spokes Call State: Idle

### *10.1.2: Call Answered by SP GUI*

SP module notifies Spokes about an answered call.

```
ICallCommand.answeredCall(callID)
```

Spokes CM brings up the audio Link (Wireless only) and enters an active call state.

- Spokes Call State: Active Call

### *10.1.3: Call ended by SoftPhone GUI*

SP module tells Spokes to drop the call.

```
ICallCommand.terminateCall(callID)
```

Spokes requests Device Listener to bring down the audio Link and terminate the call. Spokes Call Manager enters into an idle state.

- Spokes Call State: Idle

### *10.1.4: Ringing call rejected/ended by SoftPhone GUI*

SP module tells Spokes to drop the call.

```
ICallCommand.terminateCall(callID)
```

Spokes requests Device Listener to bring down the audio Link. Call terminates. Spokes Call Manager enters into an idle state.

- Spokes Call State: Idle

## 10.2: A Single Incoming Call on a SoftPhone with Auto-Answer Enabled

SP module calls Spokes API to render call after getting incoming call notification from the Softphone.

```
ICallCommand. incomingCall(callID, contact, tones, route)
```

Call Manager creates a new Call instance, requests Device Listener to render incoming call, and enters into Ringing state.

- Call answered by headset (HS) undocked:

The Device Listener brings up the audio Link (Wireless only) and reports an Undocked event to the CSM on devices that support a HS undocked event. The call transitions from a Ringing stateto active, and Spokes notifies the SP to accept the call.

```
ICallCommand.answeredCall(callID)
```

Call Manager at this point knows that a SP call is active

- Spokes Call State: Active Call

### *10.2.1: Call ended by HS docked*

The Device Listener notifies Call Manager about an Undocked event on devices that support a HS docked event. The CSM transitions from active call to Idle, Spokes notifies the SP module with a terminateCall command, and the Softphone call terminates.

```
ICallCommand.terminateCall(callID)
```

Spokes Call Manager at this point enters into an idle state.

- Spokes Call State: Idle

## 10.3: Single Outgoing Call on a Softphone

SP module calls Plantronics SDK API to render outgoing call after getting outgoing call notification from the Softphone.

```
ICallCommand. outgoingCall(callID, contact, route)
```

Call Manager creates a new Call instance, requests Device Listener to render outgoing call, and brings up the audio Link (Wireless only).

- Spokes Call State: Active call

### *10.3.1: Call ended by Talk button press*

Device Listener notifies Call Manager about Talk press, Spokes drops the active Call and brings down the audio Link (Wireless only). Spokes notifies the SP module with a terminateCall command and the Softphone call terminates.

```
ICallCommand.terminateCall(callID)
```

Spokes Call instance at this point enters into an idle state waiting for new incoming calls.

- Spokes Call State: Idle

### *10.3.2: Call ended by SP GUI*

SP module tells Spokes to drop the call

```
ICallCommand.terminateCall(callID)
```

CM requests Device Listener bring down the audio Link and terminate the call. Spokes Call instance enters into an idle state waiting for new incoming calls.

- Spokes Call State: Idle

## 10.4: Device-Initiated Outgoing Call

Call initiated from the device. Device must support keypad, contact list or call logs. Device Listener gets a makeCall command from the Device Manager and notifies the Call Manager.

CM notifies the default SP module with a makeCall command and the Softphone module will make the call with the provided number or contact ID.

```
ICallCommand.makeCall(contact)
```

If the SP module is capable of supporting the command it will respond back with an Outgoing call request.

```
ICallComand.outgoingCall(callID)
```

Call Manager creates a new Call instance and requests Device Listener to render outgoing call and bring up the audio Link (Wireless onlyCall Manager at this point transitions into an active call.

- Spokes Call State: Active call

### *10.4.1: Call ended by Talk button press*

Device Listener notifies Call Manager about Talk press. Spokes drops the active Call and brings down the audio Link (Wireless only). Spokes notifies the SP module with a terminateCall command and the Softphone call terminates.

```
ICallCommand.terminateCall(callID)
```

Spokes Call Manager at this point enters into an idle state waiting for new incoming calls.

- Spokes Call State: Idle

### *10.4.2: Call ended by SP GUI*

SP module tells Spokes to drop the call

```
ICallCommand.terminateCall(callID)
```

CM requests Device Listener bring down the audio Link and terminate the call. Spokes Call instance enters into an idle state waiting for new incoming calls.

- Spokes Call State: Idle

## 10.5: Single Softphone Call using Hold and Resume

Spokes has an active Call in SP. The call is held by Flash button press. Device Listener notifies CM about Flash button press. CM puts the current active call on hold. CM also notifies SP module with the following command.

```
ICallCommand.holdCall(callID)
```

- Spokes Call State: Held

### *10.5.1: Call Resumed by Flash button press*

With a subsequent Flash button press, CM resumes the current call on hold and notifies the SP module with the following command.

```
ICallCommand.resumeCall(callID)
```

- Spokes Call State: Active

### *10.5.2: Call Held by SP GUI Hold*

User can also put an active call on hold through the SP GUI. The SP module notifies the CM about a held call. The CM adds the call to the hold queue.

```
ICallCommand.holdCall(callID)
```

- Spokes Call State: Held

### *10.5.3: Call Resumed by SP GUI Resume*

A resume on the SP GUI notifies the CM, which pops the held call from the queue and makes it active.

```
ICallCommand.resumeCall(callID)
```

- Spokes Call State: Active

## 10.6: Single Softphone with Multiple Incoming Calls

Assuming Spokes has an active Call (Call 1) in SP and a second incoming call (Call 2) is rendered.

- Spokes Call State: Active Call – Call 1

```
ICallCommand.incomingCall(callID, contact, tones, route) – Call 2
```

Call Manager requests Device Listener render incoming call and enters into Ringing state.

At this point the user choices are the following:

- Terminate current call by pressing the Talk button and accepting incoming call with a Talk press.
- Hold current call with a Flash button press and accept incoming call with a Talk press.
- Terminate either call in SP GUI.
- Accept incoming call in SP GUI (same as #2).

### *10.6.1: Call ended by Talk button press:*

User presses Talk and DL notifies CM about Talk press. CM drops the active call and notifies the SP module.

```
ICallCommand.terminateCall(callID) - Call 1
```

On the next Talk press, DL notifies CM about Talk press and CM will answer second call and makes that call active.

```
ICallCommand.answeredCall(callID) - Call 2
```

- Spokes Call State: Active Call – Call 2

### *10.6.2: Call Flashed by Flash button press*

DL notifies CM about Flash button press. CM puts the current active Call on hold, accepts the incoming call, and makes it active. CM also notifies SP module with the following commands.

```
ICallCommand.holdCall(callID)  - Call 1
```

On the next Talk press, DL notifies CM about Talk press. CM answers second call and makes that call active.

```
ICallCommand.answeredCall(callID)  - Call 2
```

- Spokes Call State: Active Call – Call 2, Held Call – Call 1

(Note: This flash behavior is applicable only for single-channel devices and multi-channel devices that do not exhibit any pre-defined FW behavior)

### *10.6.3: A subsequent Flash button press*

DL notifies CM about Flash button press. CM puts the current active Call on hold, pops the call on the Hold calls queue, make it active, and notifies SP module.

```
ICallCommand.holdCall(callID)  - Call 2
ICallCommand.resumeCall(callID)  - Call 1
```

- Spokes Call State: Active Call – Call 1, Held Call – Call 2

### *10.6.4: Call ended by SP GUI*

SP module tells Spokes to drop the call.

```
ICallCommand.terminateCall(callID)
```

If the active call is dropped, CM drops the call and second call is in ringing state. If the second incoming call is dropped, the call will be removed from the ringing queue. If the held call is dropped, CM removes the call from the Hold queue. Once both the calls are dropped, CM will send a request to DL to bring down the audio Link. CM then enters idle state.

- Spokes Call State: Idle

## 10.7: Multiple Softphones with Multiple Incoming Calls

Same as above (10.6:) except the notifications go to different SP modules.

## 10.8: Media Player Interaction

Scenario: Assuming Media Player is playing, Spokes knows that music is playing, and an incoming call is rendered.

```
ICallCommand.incomingCall(callID, contact, tones, route)
```

Call Manager notifies Session Manager (SM) about an incoming Call. The SM publishes the command to all plug-in modules that have registered for the same.

The MP modules now take Pause, Stop or some other action based on user preference reported by the SM. Call Manager requests the DL to Ring the device.

### *10.8.1: Call answered by Talk button press*

User responds with a Talk button press. The CM notifies SP module to accept the call.

```
ICallCommand.incomingCall(callID, contact, tones, route) – To MP
ICallCommand.answeredCall(callID) - To SP
```

CM knows at this point that SP call is active and that the MP module acted on the notification (for example; to pause music).

- Spokes Call State: Active Call, Music Paused

### *10.8.2: Call ended by Talk button press*

DL notifies CM about Talk press and the active call is dropped and both SP and MP modules are notified. The MP module acts on the user preference (for example : to resume/play music.)

```
ICallCommand.terminateCall(callID) – To  SP
ICallCommand.terminateCall(callID)– To MP
```

- Spokes Call State: Idle, Music resumed

### *10.8.3: Call Answered by SP GUI*

SP module notifies Spokes about an answered call.

```
ICallCommand.incomingCall(callID, contact, tones, route) – To MP
ICallCommand.answeredCall(callID)
```

CSM enters active call state

- Spokes Call State: Active Call, Music Paused

### *10.8.4: Call ended by SP GUI*

SP module tells Spokes to drop the call

```
ICallCommand.terminateCall(callID)
ICallCommand.terminateCall(callID) – To MP
```

Spokes Call Manager at this point enters into idle state.

- Spokes Call State: Idle

## 10.9: Softphone Interaction with Multi-Channel Device (VoIP/PSTN/Mobile)

### *10.9.1: Without Pre-Defined Behavior*

Same as multiple calls handling except that, in addition to handling multiple VOIP calls, PSTN and mobile calls are also managed by the Call state machine. To support this feature, all host devices should accept, end, hold (or mute Tx and Rx), and resume (unmute Tx, Rx) PSTN and Mobile calls.

### *10.9.2: With Pre-Defined Behavior*

Same as multiple call handling except the Flash behavior will be limited to only between a VOIP call and a PSTN call. Only the VOIP call is managed by the Call state machine as the device currently takes care of the PSTN side.

# Appendix C: Sample Code

This section contains some examples of code for Plantronics SDK native interface.

## 10.10: Sample Code for Plantronics SDK Native Callback (Header file)

```
//NativeCallback.h

#pragma once

#include <memory>
#include "Spokes3G.h"

std::wstring CallStateToString( eCallState state );

class CMDeviceCallback : public IDeviceCallback
{
    virtual void onDataReceived(ReportDataEventArgs const& args);
};

class CMDeviceEventCallback : public IDeviceEventsCallback
{
    virtual void onTalkButtonPressed(DeviceEventArgs const& args);
    virtual void onButtonPressed(DeviceEventArgs const& args);
    virtual void onMuteStateChanged(DeviceEventArgs const& args);
    virtual void onAudioStateChanged(DeviceEventArgs const& args);
    virtual void onFlashButtonPressed(DeviceEventArgs const& args);
    virtual void onSmartButtonPressed(DeviceEventArgs const& args);
};

class CMDeviceListenerCallback : public IDeviceListenerCallback
{
    virtual void onHeadsetButtonPressed(DeviceListenerEventArgs const& args);
    virtual void onHeadsetStateChanged(DeviceListenerEventArgs const& args);
    virtual void onBaseButtonPressed(DeviceListenerEventArgs const& args);
    virtual void onBaseStateChanged(DeviceListenerEventArgs const& args);
    virtual void onATDStateChanged(DeviceListenerEventArgs const& args);
};
```

## 10.11: Sample Code for Plantronics SDK Native Callback (C++)

```
//NativeCallback.cpp
//#include "NativeCallBack.h"

#include <iostream>
#include <string>
```

```
#include <memory>


using namespace std;

void CMDeviceCallback::onDataReceived(ReportDataEventArgs const& args)
{
    wcout << L"Data from device received" << endl;
}
void CMDeviceEventCallback::onTalkButtonPressed(DeviceEventArgs const& args)
{
    wcout << L"Talk button pressed" << endl;
}
void CMDeviceEventCallback::onMuteStateChanged(DeviceEventArgs const& args)
{
    wcout << L"Mute state changed" << endl;
    wcout << L"New mute state: " << (args.mute?"MUTED":"UNMUTED") << endl;

}
void CMDeviceEventCallback::onButtonPressed(DeviceEventArgs const& args)
{
    wcout << L"ButtonPressed: ";

    switch (args.buttonPressed)
    {
    case HEADSET_BUTTON_VOLUME_UP :
        std::wcout << "Volume Up";
        break;
    case HEADSET_BUTTON_VOLUME_DOWN :
        std::wcout << "Volume Down";
        break;
    case HEADSET_BUTTON_MUTE :
        std::wcout << "Mute";
        break;
    default:
        std::wcout << "Button"<< endl;
    }
        }
void CMDeviceEventCallback::onAudioStateChanged(DeviceEventArgs const& args)
{
    wcout << L"Audio state changed " << (int)args.audioState << endl;
}
void CMDeviceEventCallback::onFlashButtonPressed(DeviceEventArgs const& args)
{
    wcout << L"Flash pressed" << endl;
}
void CMDeviceEventCallback::onSmartButtonPressed(DeviceEventArgs const& args)
{
```

```
    wcout << L"Smart button pressed" << endl;
}
void CMDeviceListenerCallback::onBaseButtonPressed(DeviceListenerEventArgs const& args)
{
    cout << "onBaseButtonPressed: " << BaseButtonToString(args.baseButton) << endl;
}
void CMDeviceListenerCallback::onATDStateChanged(DeviceListenerEventArgs const& args)
{
    cout << "onATDStateChanged: " << ATDStateChangeToString(args.ATDStateChange) << endl;
}
void CMDeviceListenerCallback::onBaseStateChanged(DeviceListenerEventArgs const& args)
{
    cout << "onBaseStateChanged: " << BaseStateChangeToString(args.baseStateChange) <<
endl;
}
void CMDeviceListenerCallback::onHeadsetButtonPressed(DeviceListenerEventArgs const&
args)
{
    cout << "onHeadsetButtonPressed: " << HeadsetButtonToString(args.headsetButton) <<
endl;
}
void CMDeviceListenerCallback::onHeadsetStateChanged(DeviceListenerEventArgs const& args)
{
    cout << "onHeadsetStateChanged: " <<
HeadsetStateChangeToString(args.headsetStateChange) << endl;
}

std::wstring CallStateToString( eCallState state )
{
    switch(state)
    {
        case CALL_STATE_UNKNOWN: return L"CALL_STATE_UNKNOWN";
        case CALL_STATE_ACCEPT_CALL: return L"CALL_STATE_ACCEPT_CALL";
        case CALL_STATE_TERMINATE_CALL: return L"CALL_STATE_TERMINATE_CALL";
        case CALL_STATE_HOLD_CALL: return L"CALL_STATE_HOLD_CALL";
        case CALL_STATE_RESUME_CALL: return L"CALL_STATE_RESUME_CALL";
        case CALL_STATE_FLASH: return L"CALL_STATE_FLASH";
        case CALL_STATE_CALL_IN_PROGRESS: return L"CALL_STATE_CALL_IN_PROGRESS";
        case CALL_STATE_CALL_RINGING: return L"CALL_STATE_CALL_RINGING";
        case CALL_STATE_CALL_ENDED: return L"CALL_STATE_CALL_ENDED";
        case CALL_STATE_TRANSFER_TO_HEADSET: return L"CALL_STATE_TRANSFER_TO_HEADSET";
        case CALL_STATE_TRANSFER_TO_SPEAKER: return L"CALL_STATE_TRANSFER_TO_SPEAKER";
        case CALL_STATE_MUTEON: return L"CALL_STATE_MUTEON";
        case CALL_STATE_MUTEOFF: return L"CALL_STATE_MUTEOFF";
        case CALL_STATE_MOBILE_CALL_RINGING: return L"CALL_STATE_MOBILE_CALL_RINGING";
        case CALL_STATE_MOBILE_CALL_IN_PROGRESS: return
             L"CALL_STATE_MOBILE_CALL_IN_PROGRESS";
        case CALL_STATE_MOBILE_CALL_ENDED: return L"CALL_STATE_MOBILE_CALL_ENDED";
```

```
        case CALL_STATE_DON: return L"CALL_STATE_DON";
        case CALL_STATE_DOFF: return L"CALL_STATE_DOFF";
        case CALL_STATE_CALL_IDLE: return L"CALL_STATE_CALL_IDLE";
        case CALL_STATE_PLAY: return L"CALL_STATE_PLAY";
        case CALL_STATE_PAUSE: return L"CALL_STATE_PAUSE";
        case CALL_STATE_STOP: return L"CALL_STATE_STOP";
        case CALL_STATE_DTMF_KEY: return L"CALL_STATE_DTMF_KEY";
        case CALL_STATE_REJECT_CALL: return L"CALL_STATE_REJECT_CALL";
        default: return L"Unknown Call state";
    }
}
```

## 10.12: Sample Code for Plantronics SDK (C++ Snippet)

This code snippet serves to illustrate how to get a native client app started by linking to `Spokes.dll` and calling into the native interface, and registering to receive the interested events as callbacks. It is NOT intended as a complete, ready, compile and go app.

```
//NativeSnippet.cpp
#include "Spokes3G.h"
#include "NativeCallBack.h"
#include <iostream>
#include <string>

    IDeviceEvents* deviceEvents = nullptr;
    ISessionManager* sessionManager = nullptr;
    ISession* session = nullptr;
    IDevice* device;
    IDeviceListener* devListener = nullptr;
    //CMCallEvents callEvents;
    CMDeviceEventCallback deviceEventCallback;
    CMDeviceListenerCallback deviceListenerCallback;
    //CMSessionManagerEvents sessionManagerEvents;
    const int bufferSize = 128;
    wchar_t wcharRes[bufferSize];
    wchar_t wcharRes2[bufferSize];
    wchar_t activeDeviceName[bufferSize];
    int32_t iProductID;

using namespace std;

void ExitSession()
    {
        sessionManager->unregisterSession(session);
        sessionManager->Release();
    }
    void InitSession()
```

```
    {
         if( SM_RESULT_SUCCESS == ::getSessionManager(&sessionManager) )
         {
              std::wcout << L"Get Session Manager succeeded.";
         }
         else {
              std::wcout << L"Failed to get Session Manager.";
             return;

         }

// Get session
        if( SM_RESULT_SUCCESS == sessionManager->registerSession(
           L"Spokes3GNativePlugin", &session ) )
        {
             std::wcout << L"sessionManager->
                  registerSession()=SM_RESULT_SUCCESS\n";

            if(session->getSessionId(wcharRes2,
                  bufferSize)==SM_RESULT_SUCCESS)
            {
                 std::wcout << L"SessionID = " << wcharRes2 ;
            }

            std::string input;

            if (sessionManager->registerCallback(&sessionManagerEvents)==0)
                 std::wcout << L"sessionManager->
                      registerCallback(sessionManagerEvents)=SM_RESULT_SUCCESS";
            else
                 std::wcout << L"sessionManager->
                      registerCallback(sessionManagerEvents)=SM_RESULT_FAIL";

        if( SM_RESULT_SUCCESS == session->getActiveDevice(&device) )
        {
            std::wcout << L"session->getActiveDevice(&device) =
                  SM_RESULT_SUCCESS\n";
            if (SM_RESULT_SUCCESS == sessionManager->
                  getActiveDevice(&device))
            {
            std::wcout << L"sessionManager->
                 getActiveDevice(&device) = SM_RESULT_SUCCESS\n";
            }
            else
            {
            std::wcout << L"sessionManager->getActiveDevice(&device)
                    FAILED!!!";
            }
```

```
            session->getActiveDeviceName(activeDeviceName, bufferSize);

            int32_t prodID = device->getProductID();

            //Checking whether plt device is connected
            if (prodID <= 0)
            {
                    std::wcout << L"Active device not detected";
            }

            session->registerCallback(&callEvents);

            if (device->getDeviceListener(&devListener) != DM_RESULT_SUCCESS)
            {
                    std::wcout << L"Failed to get device listener.";
            }

            if (devListener->registerCallback(&deviceListenerCallback) !=
            DM_RESULT_SUCCESS)
            {
                    std::wcout << L"Failed to register callback.";
            }

            if (device->getDeviceEvents(&deviceEvents) != DM_RESULT_SUCCESS)
            {
                    std::wcout << L"Failed to get device events" ;
            }

            if (deviceEvents->regCbDeviceEvents(&deviceEventCallback)  !=
            DM_RESULT_SUCCESS)
            {
                    std::wcout << L"Failed to register device events" ;
            }
        }
    }
```

## 10.13: Javascript code snippet for REST sample code

```
//Define CallState
function SessionCallState() {}
SessionCallState.Unknown            = 0;
SessionCallState.AcceptCall         = 1;
SessionCallState.TerminateCall      = 2;
SessionCallState.HoldCall           = 3;
SessionCallState.Resumecall         = 4;
SessionCallState.Flash              = 5;
```

```
SessionCallState.CallInProgress     = 6;
SessionCallState.CallRinging        = 7;
SessionCallState.CallEnded          = 8;
SessionCallState.TransferToHeadSet  = 9;
SessionCallState.TransferToSpeaker  = 10;
SessionCallState.MuteON             = 11;
SessionCallState.MuteOFF            = 12;
SessionCallState.MobileCallRinging  = 13;
SessionCallState.MobileCallInProgress = 14;
SessionCallState.MobileCallEnded    = 15;
SessionCallState.Don                = 16;
SessionCallState.Doff               = 17;
SessionCallState.CallIdle           = 18;
SessionCallState.Play               = 19;
SessionCallState.Pause              = 20;
SessionCallState.Stop               = 21;
SessionCallState.DTMFKey            = 22;
SessionCallState.RejectCall         = 23;

SessionCallState.Lookup = Array();
SessionCallState.Lookup[SessionCallState.Unknown] = "Unknown";
SessionCallState.Lookup[SessionCallState.AcceptCall] = "AcceptCall";
SessionCallState.Lookup[SessionCallState.TerminateCall] = "TerminateCall";
SessionCallState.Lookup[SessionCallState.HoldCall] = "HoldCall";
SessionCallState.Lookup[SessionCallState.Resumecall] = "Resumecall";
SessionCallState.Lookup[SessionCallState.Flash] = "Flash";
SessionCallState.Lookup[SessionCallState.CallInProgress] = "CallInProgress";
SessionCallState.Lookup[SessionCallState.CallRinging] = "CallRinging";
SessionCallState.Lookup[SessionCallState.CallEnded] = "CallEnded";
SessionCallState.Lookup[SessionCallState.TransferToHeadSet] = "TransferToHeadSet";
SessionCallState.Lookup[SessionCallState.TransferToSpeaker] = "TransferToSpeaker";
SessionCallState.Lookup[SessionCallState.MuteON] = "MuteON";
SessionCallState.Lookup[SessionCallState.MuteOFF] = "MuteOFF";
SessionCallState.Lookup[SessionCallState.MobileCallRinging] = "MobileCallRinging";
SessionCallState.Lookup[SessionCallState.MobileCallInProgress] = "MobileCallInProgress";
SessionCallState.Lookup[SessionCallState.MobileCallEnded] = "MobileCallEnded";
SessionCallState.Lookup[SessionCallState.Don] = "Don";
SessionCallState.Lookup[SessionCallState.Doff] = "Doff";
SessionCallState.Lookup[SessionCallState.CallIdle] = "CallIdle";
SessionCallState.Lookup[SessionCallState.Play] = "Play";
SessionCallState.Lookup[SessionCallState.Pause] = "Pause";
SessionCallState.Lookup[SessionCallState.Stop] = "Stop";
SessionCallState.Lookup[SessionCallState.DTMFKey] = "DTMFKey";
SessionCallState.Lookup[SessionCallState.RejectCall] = "RejectCall";
  //Define all the audio states that exist
function SpokesAudioType() {}
SpokesAudioType.MonoOn        = 1;
SpokesAudioType.MonoOff       = 2;
```

```
SpokesAudioType.StereoOn      = 3;
SpokesAudioType.StereoOff     = 4;
SpokesAudioType.MonoOnWait    = 5;
SpokesAudioType.StereoOnWait  = 6;

  //Create a lookup for the textual names of the enum
SpokesAudioType.Lookup = Array();
SpokesAudioType.Lookup[SpokesAudioType.MonoOn] = "MonoOn";
SpokesAudioType.Lookup[SpokesAudioType.MonoOff] = "MonoOff";
SpokesAudioType.Lookup[SpokesAudioType.StereoOn] = "StereoOn";
SpokesAudioType.Lookup[SpokesAudioType.StereoOff] = "StereoOff";
SpokesAudioType.Lookup[SpokesAudioType.MonoOnWait] = "MonoOnWait";
SpokesAudioType.Lookup[SpokesAudioType.StereoOnWait] = "StereoOnWait";

  //Define my response types
function SpokesResponseType() {}
SpokesResponseType.Unknown             = 0;
SpokesResponseType.Error               = 1;
SpokesResponseType.Bool                = 2;
SpokesResponseType.Integer             = 3;
SpokesResponseType.DeviceInfo          = 4;
SpokesResponseType.DeviceInfoArray     = 5;
SpokesResponseType.DeviceEventArray    = 6;
SpokesResponseType.SessionHash         = 7;
SpokesResponseType.String              = 8;
SpokesResponseType.CallManagerState    = 9;
SpokesResponseType.CallStateArray      = 10;
SpokesResponseType.ContactArray        = 11;
SpokesResponseType.StringArray         = 12;

  //Define my response class
function SpokesResponse( obj )
{
    //Copy the JSON data into this object
  this.Type         = obj["Type"];
  this.Type_Name    = obj["Type_Name"];
  this.Description  = obj["Description"];
  this.isError      = obj["isError"];
  this.isValid      = true;
  this.Err          = null;
  this.Result       = null;

    //If we have an error, null the result and populate error, else opposite
  if ( this.isError )
  {
    this.Err = new SpokesError( obj["Err"] );
    return;
  }
```

```
  //Register a plugin
Plugin.prototype.register = function (name, callback) {
    //Register the callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    //Register a session
    $.getJSON(this.Path + "/SessionManager/Register?name=" + name +
     "&callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.Bool)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
}

//Unregister a plugin
Plugin.prototype.unRegister = function (name, callback) {
    //Register the callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    //Register a session
    $.getJSON(this.Path + "/SessionManager/UnRegister?name=" + name + "&callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.Bool)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
}
```

```
//Set that the plugin is active
Plugin.prototype.isActive = function (name, active, callback) {
    //Register the callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    //Register a session
    $.getJSON(this.Path + "/SessionManager/IsActive?name=" + name +
                "&active=" + active +
                "&callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.Bool)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
}
//////////////////////////////////////
// Call services

//callManagerState
Plugin.prototype.callManagerState = function (callback) {
    //Register callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    //Register a session
    $.getJSON(this.Path + "/CallServices/CallManagerState?callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.CallManagerState)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
```

```
}
// session events
Plugin.prototype.sessionEvents = function (callback) {
    //Register callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    //Register a session
    $.getJSON(this.Path + "/CallServices/Events?callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.StringArray)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
}
// call call events
Plugin.prototype.sessionCallEvents = function (name, callback) {
    //Register callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    //Register a session
    $.getJSON(this.Path + "/CallServices/SessionManagerCallEvents?name=" + name +
      "&callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.StringArray)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
}

// callEvents
```

```
Plugin.prototype.callEvents = function (name, callback) {
    //Register callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    //Register a session
    $.getJSON(this.Path + "/CallServices/CallEvents?name=" + name + "&callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.CallStateArray)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
}

// callRequests
Plugin.prototype.callRequests = function (name, callback) {
    //Register callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    //Register a session
    $.getJSON(this.Path + "/CallServices/CallRequests?name=" + name + "&callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.ContactArray)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
}
// TODO: RingTone, AudioRoute
// incomingCall
Plugin.prototype.incomingCall = function (name, callID, contact, tones, route, callback)
{
```

```
    //Register callback
    var action = new SpokesAction(callback);
    SpokesAction.Action_List.unshift(action);

    if (callID.getName() != "SpokesCallId" || contact.getName() != "SpokesContact")
        return false;

    callID = JSON.stringify(callID);
    contact = JSON.stringify(contact);
    //Register a session
    $.getJSON(this.Path + "/CallServices/IncomingCall?name=" + name +
                                            "&callID=" + callID +
                                            "&contact=" + contact +
                                            "&tones=" + tones +
                                            "&route=" + route +
                                            "&callback=?",
            function (data) {
                //Create a nice object, and ensure its of the right type
                var resp = new SpokesResponse(data);
                if (resp.Type != SpokesResponseType.Bool)
                    resp.isValid = false;

                //Pass my result back to the user
                action.isValid = false;
                if (action.Callback != null && action.Callback != undefined)
                    action.Callback(resp);
            });

    return true;
}
```

# Chapter 11: Appendix D: Plantronics SDK v3.6 COM Server Migration Guidelines

This appendix addresses the changes between Spokes 2.x and Plantronics SDK v3.0. Plantronics SDK v3.6 includes these same differences.

## 11.1: Architectural Differences

The Plantronics SDK COM server is developed as a C++ ATL project and is deployed as a Spokes plug-in, `PlantronicsCOM.dll`. Based on the COM specification, this Module provides the following entry points: *`DllCanUnloadNow`*, *`DllGetClassObject`*, *`DllRegisterServer`*, *`DllUnregisterServer`*. During installation, the module is registered to the system. The hosting process, `PLTHub.exe`, is pointed as the out-of-process COM server for co-class *`COMSessionManager`*. This is the only class that can be directly instantiated by COM client.

The Spokes 2.x COM server relies on the .NET runtime to create COM-Callable Wrappers (CCW) that expose all managed types marked for COM interop to COM, and marshal calls between COM clients and .NET objects. During installation, all types that are visible to COM are registered to the system. `PlantronicsURE.exe` process is defined as an out-of-process server for co-class `SessionComManager`, and the `Plantronics.Device.Common.dll` module is registered as an in-process server for co-class *DeviceManager*.

The Spokes 2.x COM server has .NET 2.0 as a prerequisite. This is not the case for the Plantronics SDK v3.x COM server.

The Spokes 2.x COM server also exposes the Device Manager to COM clients, which provides access to all devices attached to system. Plantronics SDK v3.x exposes only the Session Manager layer, and only current active devices are accessible.

## 11.2: Type Library Differences

The Spokes SDK provides Type Library files that contain metadata representing COM types. These files are processed by language compilers and runtime environments.

Spokes 2.x SDK provides two type libraries:

- `Plantronics.Device.Common.tlb` – Contains metadata for Device Manager Interfaces and types.
- `Plantronics.UC.Common.tlb` – Contains metadata from Session Manager.

Plantronics SDK provides a single type library:

- `Plantronics.tlb`

When type libraries from the C++ language are used, the COM client should use the `#import` command to introduce proper types and create smart pointers.

For Spokes 2.x, both libraries MUST be included:

```
#import "Plantronics.Device.Common.tlb"
#import "Plantronics.UC.Common.tlb"
```

For Plantronics SDK client, only a single library must be imported:

```
#import "Plantronics.tlb"
```

## 11.3: Changes in Interface Sources

The following tables present differences in the IDL between versions 2.x and v3.0 of Spokes.

### *11.3.1: Session Manager Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **ISessionCOMManager** | **ICOMSessionManager** |
| `HRESULT Register([in] BSTR plgName,[out, retval] IComSession** pRetVal);` | `HRESULT Register(BSTR pluginName, [out] ICOMSession** pSession)` |
| `HRESULT UnRegister([in] IComSession* session);` | `HRESULT Unregister(ICOMSession* pSession)` |
| `HRESULT CallManagerState([out, retval] ICallManagerState** pRetVal);` | `HRESULT get_CallManagerState([out, retval] ICOMCallManagerState** pVal )` |
| Does not exist in Spokes 2.x | `HRESULT get_UserPreference([out, retval] ICOMUserPreference** pVal)` |

### *11.3.2: COM Session Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IComSession** | **ICOMSession** |
| `HRESULT Id([out, retval] GUID* pRetVal);` | `HRESULT get_SessionId([out, retval] BSTR* pVal);` |
| `HRESULT PluginName([out, retval] BSTR* pRetVal);` | `HRESULT get_PluginName([out, retval] BSTR* pVal);` |
| `HRESULT CallCommand([out, retval] ICallCommand** pRetVal);` | `HRESULT GetCallCommand([out,retval] ICOMCallCommand** callCommand)` |
| `HRESULT CallEvents([out, retval] ICallEvents** pRetVal);` | Deprecated. *ICOMSession* is the source of Call events. |
| `HRESULT ActiveDevice([out, retval] IDevice** pRetVal)` | `HRESULT GetActiveDevice([out,retval] ICOMDevice** device);` |

| | | |
|---|---|---|
| `HRESULT EnableAudio([in] VARIANT_BOOL bEnable)` | Does not exist. | |
| | The same as *ICOMSessionManager* unregister | `HRESULT Unregister(void);` |

## *11.3.3: Call Manager State Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **ICallManagerState** | **ICOMCallManagerState** |
| `HRESULT HasActiveCall([out, retval] VARIANT_BOOL* pRetVal)` | `HRESULT get_HasActiveCall([out, retval] VARIANT_BOOL* pVal)` |
| `HRESULT CallsOnHold([out, retval] long* pRetVal)` | `HRESULT get_NumberOfHoldCalls([out, retval] SHORT* pVal)` |
| `HRESULT GetCalls([out, retval] IEnumerable** pRetVal)` | `HRESULT get_Calls([out, retval] SAFEARRAY** ppArray)` |
| `[restricted] void Missing9()` | This was auto-generated by the .NET compiler. |

## *11.3.4: Call Command Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **ICallCommand** | **ICOMCallCommand** |
| `HRESULT IncomingCall([in] ICall* CallId, [in] IContact* Contact,[in] RingTone tones, [in] AudioRoute route);` | `HRESULT IncomingCall(ICOMCall* call, ICOMContact* contact, CallRingTone ringTone, CallAudioRoute audioRoute)` |
| `HRESULT OutgoingCall([in] ICall* CallId,[in] IContact* Contact, [in] AudioRoute route);` | `HRESULT OutgoingCall(ICOMCall* call, ICOMContact* contact, CallAudioRoute audioRoute)` |
| `HRESULT TerminateCall([in] ICall* CallId)` | `HRESULT TerminateCall(ICOMCall* call)` |
| `HRESULT AnsweredCall([in] ICall* CallId)` | `HRESULT AnsweredCall(ICOMCall* call)` |

| | |
|---|---|
| HRESULT HoldCall([in] ICall* CallId) | HRESULT HoldCall(ICOMCall* call); |
| HRESULT ResumeCall([in] ICall* CallId) | HRESULT ResumeCall(ICOMCall* call) |
| HRESULT Mute([in] ICall* CallId,[in] VARIANT_BOOL bMute) | HRESULT MuteCall(ICOMCall* call, VARIANT_BOOL mute) |
| HRESULT InsertCall([in] ICall* CallId, [in] IContact* Contact); | HRESULT InsertCall(ICOMCall* call, ICOMContact* contact) |
| HRESULT SetAudioRoute([in] ICall* CallId, [in] AudioRoute route); | HRESULT SetAudioRoute(ICOMCall* call, CallAudioRoute audioRoute) |
| HRESULT SetConferenceId([in] ICall* CallId); | HRESULT SetConferenceId(ICOMCall* call) |
| HRESULT CallHandOff([in] VARIANT_BOOL bEnable); | Deprecated. |
| HRESULT MakeCall([in] IContact* Contact) | HRESULT MakeCall(ICOMContact* call) |

## *11.3.5: Device Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IDevice** | **ICOMDevice** |
| HRESULT DevicePath([out, retval] BSTR* pRetVal); | Deprecated |
| HRESULT VendorID([out, retval] long* pRetVal); | HRESULT get_VendorId([out, retval] USHORT* pVal); |
| HRESULT ProductID([out, retval] long* pRetVal); | HRESULT get_ProductId([out, retval] USHORT* pVal); |
| HRESULT VersionNumber([out, retval] long* pRetVal); | HRESULT get_VersionNumber([out, retval] USHORT* pVal); |
| HRESULT InternalName([out, retval] BSTR* pRetVal); | HRESULT get_InternalName([out, retval] BSTR* pVal); |
| HRESULT ProductName([out, retval] BSTR* pRetVal); | HRESULT get_ProductName([out, retval] BSTR* pVal); |

| | |
|---|---|
| `HRESULT ManufacturerName([out, retval] BSTR* pRetVal);` | `HRESULT get_ManufacturerName([out, retval] BSTR* pVal);` |
| `HRESULT serialNumber([out, retval] BSTR* pRetVal);` | `HRESULT get_SerialNumber([out, retval] BSTR* pVal);` |
| `HRESULT IsAttached([out, retval] VARIANT_BOOL* pRetVal);` | Deprecated |
| `HRESULT Attribute([out, retval] IDeviceAttribute** pRetVal);` | Deprecated |
| `HRESULT DeviceEvents([out, retval] IDeviceEvents** pRetVal);` | Object that implements ICOMDevice is source of device related events |
| `HRESULT HostCommand([out, retval] IHostCommand** pRetVal);` | `HRESULT get_HostCommand([out, retval] ICOMHostCommand** pVal);` |
| `HRESULT add_DataReceived([in] IUnknown* value);` | Deprecated |
| `HRESULT remove_DataReceived([in] IUnknown* value);` | Deprecated |
| `HRESULT add_DeviceStateChanged([in] IUnknown* value);` | Deprecated |
| `HRESULT remove_DeviceStateChanged([in] IUnknown* value);` | Deprecated |
| `HRESULT Attach();` | Deprecated |
| `HRESULT Detach();` | Deprecated |
| `HRESULT RegisterNotification([out, retval] VARIANT_BOOL* pRetVal);` | Deprecated |
| `HRESULT UnRegisterNotification([out, retval] VARIANT_BOOL* pRetVal);` | Deprecated |
| `HRESULT IsSupported( [in] unsigned short Usage, [out, retval] VARIANT_BOOL* pRetVal);` | Deprecated |

| | |
|---|---|
| `HRESULT Dispose();` | This is a .NET function, so it is deprecated in Plantronics SDK v3.x. |
| `HRESULT DeviceListener([out, retval] IDeviceListener** pRetVal);` | `HRESULT get_DeviceListener([out, retval] ICOMDeviceListener** pVal);` |

## *11.3.6: Host Command Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IHostCommand** | **ICOMHostCommand** |
| `HRESULT Ring([in] VARIANT_BOOL enable);` | `HRESULT put_SetRing([in] VARIANT_BOOL newVal);` |
| `HRESULT AudioState([out, retval] AudioType* pRetVal);` | `HRESULT get_AudioState([out, retval] DeviceAudioState* pVal);` |
| `HRESULT AudioState([in] AudioType pRetVal);` | `HRESULT put_AudioState([in] DeviceAudioState newVal);` |
| `HRESULT Mute([out, retval] VARIANT_BOOL* pRetVal);` | `HRESULT get_Mute([out, retval] VARIANT_BOOL* pVal);` |
| `HRESULT Register( [in] RegistrationType sign,[out, retval] VARIANT_BOOL* pRetVal);` | `HRESULT RegisterCommand(DeviceRegistrationType registrationType);` |
| `HRESULT GetVersion( [in] VersionType version, [out, retval] BSTR* pRetVal);` | `HRESULT GetFirmwareVersion(FirmwareVersionType versionType, [out,retval] BSTR* version);` |
| `HRESULT IsSupported( [in] FeatureType feature, [out, retval] VARIANT_BOOL* pRetVal);` | Deprecated |

## *11.3.7: Host Command Ext Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IHostCommandExt** | **ICOMHostCommandExt** |
| `HRESULT BatteryLevel([out, retval] BatteryLevel* pRetVal);` | `HRESULT get_BatteryLevel([out, retval] DeviceBatteryLevel* pVal);` |

| | |
|---|---|
| HRESULT HeadsetsInConference([out, retval] long* pRetVal); | HRESULT get_NumHeadsetInConf([out, retval] SHORT* pVal); |
| HRESULT SetHeadsetMute([in] VARIANT_BOOL bMute); | HRESULT put_MuteHeadset([in] VARIANT_BOOL newVal); |
| HRESULT HeadsetType([out, retval] HeadsetType* pRetVal); | HRESULT get_HeadsetType([out, retval] DeviceHeadsetType* pVal); |
| HRESULT GetVersion(); | Deprecated |
| HRESULT SetActiveLink( [in] LineType LineType, [in] VARIANT_BOOL bActive,[out, retval] VARIANT_BOOL* pRetVal); | HRESULT SetActiveLink(COMLineType lineType, VARIANT_BOOL bActive); |
| HRESULT IsLineActive( [in] LineType LineType, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT IsLineActive(COMLineType lineType, [out, retval] VARIANT_BOOL* pVal); |
| HRESULT Hold( [in] LineType LineType, [in] VARIANT_BOOL bHold, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT SetLinkHoldState(COMLineType lineType, VARIANT_BOOL bHold); |
| HRESULT GetHoldState( [in] LineType LineType, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT GetLinkHoldState(COMLineType lineType, [out, retval] VARIANT_BOOL* pVal); |
| HRESULT DialedKey([out, retval] long* pRetVal); | Deprecated |
| HRESULT AudioLinkState([out, retval] AudioLinkState* pRetVal); | HRESULT get_AudioLinkState([out, retval] DeviceAudioLinkState* pVal); |
| HRESULT IsHeadsetDocked([out, retval] VARIANT_BOOL* pRetVal); | HRESULT get_HeadsetDocked([out, retval] VARIANT_BOOL* pVal); |
| HRESULT HeadsetProductName([out, retval] BSTR* pRetVal); | HRESULT get_HeadsetProductName([out, retval] BSTR* pVal); |
| HRESULT AudioLocation([out, retval] AudioLocation* pRetVal); | HRESULT get_AudioLocation([out, retval] DeviceAudioLocation* pVal); |

| | |
|---|---|
| HRESULT BatteryInfo([out, retval] IBTBatteryInfo** pRetVal); | HRESULT get_BatteryInfo([out, retval] ICOMBatteryInfo** pVal); |
| HRESULT GetSerialNumber( [in] DeviceType devType, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT RequestSerialNumber(COMDeviceType deviceType); |
| HRESULT IsConstant( [in] unsigned short Usage, [out, retval] VARIANT_BOOL* pRetVal); | Deprecated |
| HRESULT GetProximity([out, retval] VARIANT_BOOL* pRetVal); | HRESULT RequestProximity(void); |
| HRESULT EnableProximity( [in] VARIANT_BOOL bEnable, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT EnableProximity(VARIANT_BOOL bEnable); |
| HRESULT HeadsetState([out, retval] HeadsetState* pRetVal); | HRESULT get_HeadsetState([out, retval] DeviceHeadsetState* pVal); |
| HRESULT GetSerialNumber_2( [in] DeviceType devType, [in] SAFEARRAY(unsigned char) serialNumber, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT GetSerialNumber([in] COMDeviceType deviceType, [out, retval] VARIANT* pVal); |

### *11.3.8: ATD Command Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IATDCommand** | **ICOMATDCommand** |
| HRESULT AnswerMobileCall([out, retval] VARIANT_BOOL* pRetVal); | HRESULT AnswerMobileCall(void); |
| HRESULT EndMobileCall([out, retval] VARIANT_BOOL* pRetVal); | HRESULT EndMobileCall(void); |
| HRESULT Redial([out, retval] VARIANT_BOOL* pRetVal); | HRESULT Redial(void); |
| HRESULT MakeMobileCall( [in] BSTR CallerID, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT MakeMobileCall(BSTR callerId); |

| | |
|---|---|
| HRESULT GetMobileCallStatus([out, retval] VARIANT_BOOL* pRetVal); | HRESULT RequestMobileCallStatus(void); |
| HRESULT MuteMobileCall( [in] VARIANT_BOOL bMute, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT MuteMobileCall(VARIANT_BOOL bMute); |
| HRESULT CallerID([out, retval] BSTR* pRetVal); | HRESULT CallerId([out, retval] BSTR* pVal); |

### *11.3.9: Device Listener Interface*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IDeviceListener** | **ICOMDeviceListener** |
| HRESULT Ring([in] VARIANT_BOOL enable); | HRESULT Ring(VARIANT_BOOL enable); |
| HRESULT AudioState([out, retval] AudioType* pRetVal); | HRESULT get_AudioState([out, retval] DeviceAudioState* pVal); |
| HRESULT AudioState([in] AudioType pRetVal); | HRESULT put_AudioState([in] DeviceAudioState newVal); |
| HRESULT Mute([out, retval] VARIANT_BOOL* pRetVal); | HRESULT get_Mute([out, retval] VARIANT_BOOL* pVal); |
| HRESULT Mute([in] VARIANT_BOOL pRetVal); | HRESULT put_Mute([in] VARIANT_BOOL newVal); |
| HRESULT EnableIntellistand([out, retval] VARIANT_BOOL* pRetVal); | HRESULT get_Intellistand([out, retval] VARIANT_BOOL* pVal); |
| HRESULT EnableIntellistand([in] VARIANT_BOOL pRetVal); | HRESULT put_Intellistand([in] VARIANT_BOOL newVal); |
| HRESULT AudioMixerState([out, retval] MuteState* pRetVal); | HRESULT get_AudioMixerState([out, retval] DeviceMuteState* pVal); |
| HRESULT AudioMixerState([in] MuteState pRetVal); | HRESULT put_AudioMixerState([in] DeviceMuteState newVal); |
| HRESULT DefaultLine([out, retval] LineType* pRetVal); | HRESULT get_DefaultLine([out, retval] COMLineType* pVal); |

| | |
|---|---|
| HRESULT DefaultLine([in] LineType pRetVal); | HRESULT put_DefaultLine([in] COMLineType newVal); |
| HRESULT SuppressDialTone([in] VARIANT_BOOL bOn); | Deprecated |
| HRESULT InCall([in] VARIANT_BOOL bOn); | HRESULT InCall(VARIANT_BOOL bOn); |
| HRESULT DialedKey([out, retval] long* pRetVal); | Deprecated |
| HRESULT SetActiveLink( [in] LineType LineType, [in] VARIANT_BOOL bActive, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT SetActiveLink(COMLineType lineType, VARIANT_BOOL bActive); |
| HRESULT IsLineActive( [in] LineType LineType, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT IsLineActive(COMLineType lineType, [out, retval] VARIANT_BOOL* pVal); |
| HRESULT Hold( [in] LineType LineType,[in] VARIANT_BOOL bHold,[out, retval] VARIANT_BOOL* pRetVal); | HRESULT Hold(COMLineType lineType, VARIANT_BOOL bHold); |
| HRESULT GetHoldState( [in] LineType LineType, [out, retval] VARIANT_BOOL* pRetVal); | HRESULT GetHoldState(COMLineType lineType, [out, retval] VARIANT_BOOL* pVal); |
| HRESULT AudioLinkState([out, retval] AudioLinkState* pRetVal); | HRESULT get_AudioLinkState([out, retval] DeviceAudioLinkState* pVal); |
| HRESULT AudioLocation([out, retval] AudioLocation* pRetVal); | HRESULT get_AudioLocation([out, retval] DeviceAudioLocation* pVal); |

## *11.3.10: Battery Info Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IBTBatteryInfo** | **ICOMBatteryInfo** |
| HRESULT Life([out, retval] long* pRetVal); | HRESULT get_Lifetime([out, retval] SHORT* pVal); |
| HRESULT Life([in] long pRetVal); | Deprecated |

| | |
|---|---|
| HRESULT Status([out, retval] BTBatteryStatus* pRetVal); | HRESULT get_BatteryStatus([out, retval] DeviceBatteryStatus* pVal); |
| HRESULT Status([in] BTBatteryStatus pRetVal); | Deprecated |
| HRESULT Charge([out, retval] BTChargingStatus* pRetVal); | HRESULT get_ChargingStatus([out, retval] DeviceChargingStatus* pVal); |
| HRESULT Charge([in] BTChargingStatus pRetVal); | Deprecated |

### *11.3.11: Call Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **ICall** | **ICOMCall** |
| HRESULT Id([out, retval] long* pRetVal); | HRESULT get_Id([out, retval] LONG* pVal); |
| HRESULT Id([in] long pRetVal); | HRESULT put_Id([in] LONG newVal); |
| HRESULT InConference([out, retval] VARIANT_BOOL* pRetVal); | HRESULT get_ConferenceId([out, retval] LONG* pVal); |
| HRESULT InConference([in] VARIANT_BOOL pRetVal); | HRESULT put_ConferenceId([in] LONG newVal); |
| HRESULT ConferenceId([out, retval] long* pRetVal); | HRESULT get_InConference([out, retval] VARIANT_BOOL* pVal); |
| HRESULT ConferenceId([in] long pRetVal); | HRESULT put_InConference([in] VARIANT_BOOL newVal); |

### *11.3.12: Contact Interfaces*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IContact** | **ICOMContact** |
| HRESULT Name([out, retval] BSTR* pRetVal); | HRESULT get_Name([out, retval] BSTR* pVal); |
| HRESULT Name([in] BSTR pRetVal); | HRESULT put_Name([in] BSTR newVal); |

| | |
|---|---|
| HRESULT FriendlyName([out, retval] BSTR* pRetVal); | HRESULT get_FriendlyName([out, retval] BSTR* pVal); |
| HRESULT FriendlyName([in] BSTR pRetVal); | HRESULT put_FriendlyName([in] BSTR newVal); |
| HRESULT Id([out, retval] long* pRetVal); | HRESULT get_ID([out, retval] LONG* pVal); |
| HRESULT Id([in] long pRetVal); | HRESULT put_ID([in] LONG newVal); |
| HRESULT SipUri([out, retval] BSTR* pRetVal); | HRESULT get_SipUri([out, retval] BSTR* pVal); |
| HRESULT SipUri([in] BSTR pRetVal); | HRESULT put_SipUri([in] BSTR newVal); |
| HRESULT Phone([out, retval] BSTR* pRetVal); | HRESULT get_Phone([out, retval] BSTR* pVal); |
| HRESULT Phone([in] BSTR pRetVal); | HRESULT put_Phone([in] BSTR newVal); |
| HRESULT Email([out, retval] BSTR* pRetVal); | HRESULT get_Email([out, retval] BSTR* pVal); |
| HRESULT Email([in] BSTR pRetVal); | HRESULT put_Email([in] BSTR newVal); |
| HRESULT WorkPhone([out, retval] BSTR* pRetVal); | HRESULT get_WorkPhone([out, retval] BSTR* pVal); |
| HRESULT WorkPhone([in] BSTR pRetVal); | HRESULT put_WorkPhone([in] BSTR newVal); |
| HRESULT MobilePhone([out, retval] BSTR* pRetVal); | HRESULT get_MobilePhone([out, retval] BSTR* pVal); |
| HRESULT MobilePhone([in] BSTR pRetVal); | HRESULT put_MobilePhone([in] BSTR newVal); |
| HRESULT HomePhone([out, retval] BSTR* pRetVal); | HRESULT get_HomePhone([out, retval] BSTR* pVal); |
| HRESULT HomePhone([in] BSTR pRetVal); | HRESULT put_HomePhone([in] BSTR newVal); |

## 11.4: Changes in Event Sources

### *11.4.1: Session Manager Events*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **ISessionCOMManagerEvents** | **ICOMSessionManagerEvents** |
| `HRESULT DeviceStateChanged( [in] VARIANT sender,[in] _DeviceStateEventArgs* e);` | `HRESULT onDeviceStateChanged([in] ICOMStateDeviceEventArgs* deviceEventArgs);` |
| `HRESULT CallStateChanged( [in] VARIANT sender, [in] _CallStateEventArgs* e);` | `HRESULT onCallStateChanged([in]ICOMCallEventArgs* callEventArgs);` |

### *11.4.2: Call Events*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **ICOMCallEvents** | **ICOMCallEvents** |
| `HRESULT CallRequested( [in] VARIANT sender,[in] _CallRequestEventArgs* e);` | `HRESULT onCallRequested([in] ICOMCallRequestEventArgs* callRequestedEventArgs);` |
| `HRESULT CallStateChanged([in] VARIANT sender,[in] _CallStateEventArgs* e);` | `HRESULT onCallStateChanged([in]ICOMCallEventArgs* callEventArgs);` |

### *11.4.3: Device Events*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IDeviceCOMEvents** | **ICOMDeviceEvents** |
| `HRESULT TalkPressed( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | `HRESULT onTalkButtonPressed(ICOMDeviceEventArgs* args);` |
| `HRESULT ButtonPressed( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | `HRESULT onButtonPressed(ICOMDeviceEventArgs* args);` |

| | |
|---|---|
| `HRESULT MuteStateChanged( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | `HRESULT onMuteStateChanged(ICOMDeviceEventArgs* args);` |
| `HRESULT AudioStateChanged( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | `HRESULT onAudioStateChanged(ICOMDeviceEventArgs* args);` |
| `HRESULT FlashPressed( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | `HRESULT onFlashButtonPressed(ICOMDeviceEventArgs* args);` |
| `HRESULT SmartPressed([in] VARIANT sender, [in] _DeviceEventArgs* e);` | `HRESULT onSmartButtonPressed(ICOMDeviceEventArgs* args);` |

### *11.4.4: COM Device Events Ext*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IDeviceCOMEventsExt** | **ICOMDeviceEventsExt** |
| `HRESULT TalkPressed( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | Use *ICOMDeviceEvents* interface |
| `HRESULT ButtonPressed( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | Use *ICOMDeviceEvents* interface |
| `HRESULT MuteStateChanged( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | Use *ICOMDeviceEvents* interface |
| `HRESULT AudioStateChanged( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | Use *ICOMDeviceEvents* interface |
| `HRESULT FlashPressed( [in] VARIANT sender, [in] _DeviceEventArgs* e);` | Use *ICOMDeviceEvents* interface |
| `HRESULT SmartPressed([in] VARIANT sender, [in] _DeviceEventArgs* e);` | Use *ICOMDeviceEvents* interface |

| | |
|---|---|
| `HRESULT BatteryLevelChanged([in] VARIANT sender,[in] _BatteryLevelEventArgs* e);` | `HRESULT onBatteryLevelChanged (ICOMBatteryLevelEventArgs* args);` |
| `HRESULT HeadsetStateChanged( [in] VARIANT sender,[in] _HeadsetStateEventArgs* e);` | `HRESULT onHeadsetStateChanged (ICOMHeadsetStateEventArgs* args);` |
| | **ICOMMobilePresenceEvents** |
| Not supported in 2.x | `HRESULT onPresenceChanged (ICOMMobilePresenceEventArgs* args);` |
| | **ICOMBaseEvents** |
| *IDeviceCOMEvents*, *IDeviceCOMEventsExt* | Use *ICOMDeviceEvents* and *ICOMDeviceEventsExt* interfaces instead. |
| `HRESULT BaseEventReceived( [in] VARIANT sender, [in] _BaseEventArgs* e);` | `HRESULT onBaseEventReceived(ICOMBaseEventArgs* args);` |

## *11.4.5: COM Device Listener Events*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **IDeviceListenerCOMEvents** | **ICOMDeviceListenerEvents** |
| `HRESULT HeadsetButtonPressed( [in] VARIANT sender, [in] _DeviceListenerEventArgs* e);` | `HRESULT onHeadsetButtonPressed (ICOMDeviceListenerEventArgs* args);` |
| `HRESULT HeadsetStateChanged( [in] VARIANT sender, [in] _DeviceListenerEventArgs* e);` | `HRESULT onHeadsetStateChanged (ICOMDeviceListenerEventArgs* args);` |
| `HRESULT BaseButtonPressed([in] VARIANT sender, [in] _DeviceListenerEventArgs* e);` | `HRESULT onBaseButtonPressed (ICOMDeviceListenerEventArgs* args);` |
| `HRESULT BaseStateChanged( [in] VARIANT sender, [in] _DeviceListenerEventArgs* e);` | `HRESULT onBaseStateChanged (ICOMDeviceListenerEventArgs* args);` |

| | |
|---|---|
| HRESULT ATDStateChanged( [in] VARIANT sender, [in] _DeviceListenerEventArgs* e); | HRESULT onATDStateChanged (ICOMDeviceListenerEventArgs* args); |

## *11.4.6: Call Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **_CallStateEventArgs** | **ICOMCallEventArgs** |
| ToString, Equals, GetHashCode, GetType | Not supported, this is .NET legacy |
| HRESULT Action([out, retval] CallState* pRetVal); | HRESULT get_CallState([out, retval] CallState* pVal); |
| HRESULT DeviceEvent([out, retval] DeviceEventKind* pRetVal); | HRESULT get_DeviceEventKind([out, retval] DeviceEventKind* pVal); |
| | GetOptions(BSTR option, [out] BSTR* value); |
| HRESULT CallId([out, retval] ICall** pRetVal); | HRESULT get_call([out, retval] ICOMCall** pVal); |
| HRESULT CallSource([out, retval] BSTR* pRetVal); | HRESULT get_CallSource([out, retval] BSTR* pVal); |
| HRESULT UserPreference([out, retval] IUnknown** pRetVal); | Deprecated. |
| | HRESULT get_CallerIdentity([out,retval] BSTR* pVal); |

## *11.4.7: Device Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **_DeviceEventArgs** | **ICOMDeviceEventArgs** |
| ToString, Equals, GetHashCode, GetType | Not supported, this is .NET legacy |
| HRESULT ButtonPressed([out, retval] HeadsetButton* pRetVal); | HRESULT get_ButtonPressed([out, retval] DeviceHeadsetButton* pVal); |

| | |
|---|---|
| `HRESULT Usage([out, retval] _Usage** pRetVal);` | Deprecated |
| `HRESULT Mute([out, retval] VARIANT_BOOL* pRetVal);` | `HRESULT get_mute([out, retval] VARIANT_BOOL* pVal);` |
| `HRESULT AudioState([out, retval] AudioType* pRetVal);` | `HRESULT get_AudioState([out, retval] DeviceAudioState* pVal);` |

## *11.4.8: Call Request Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **_CallRequestEventArgs** | **ICOMCallRequestEventArgs** |
| `HRESULT ToString([out, retval] BSTR* pRetVal);` | Not supported, this is .NET legacy |
| `HRESULT Equals( [in] VARIANT obj, [out, retval] VARIANT_BOOL* pRetVal);` | Not supported, this is .NET legacy |
| `HRESULT GetHashCode([out, retval] long* pRetVal);` | Not supported, this is .NET legacy |
| `HRESULT GetType([out, retval] _Type** pRetVal);` | Not supported, this is .NET legacy |
| `HRESULT Contact([out, retval] IContact** pRetVal);` | `HRESULT get_contact([out, retval] ICOMContact** pVal)` |

## *11.4.9: State Device Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **_DeviceStateEventArgs** | **ICOMStateDeviceEventArgs** |
| `ToString, Equals, GetHashCode, GetType` | Not supported, this is .NET legacy |
| `HRESULT State([out, retval] DeviceState* pRetVal);` | `HRESULT get_DeviceState([out, retval] DeviceChangeState* pVal);` |

| | |
|---|---|
| `HRESULT DevicePath([out, retval] BSTR* pRetVal);` | Deprecated |

## *11.4.10: Battery Level Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **_BatteryLevelEventArgs** | **ICOMBatteryLevelEventArgs** |
| `ToString, Equals, GetHashCode, GetType` | Not supported, this is .NET legacy |
| `HRESULT Level([out, retval] BatteryLevel* pRetVal);` | `HRESULT get_BatteryLevel([out, retval] DeviceBatteryLevel* pVal)` |
| `HRESULT Level([in] BatteryLevel pRetVal);` | Setter is deprecated |

## *11.4.11: Headset State Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **_HeadsetStateEventArgs** | **ICOMHeadsetStateEventArgs** |
| `ToString, Equals, GetHashCode, GetType` | Not supported, this is .NET legacy |
| `HRESULT State([out, retval] HeadsetState* pRetVal);` | `HRESULT get_HeadsetState([out, retval] DeviceHeadsetState* pVal);` |
| `HRESULT HeadsetsInConference([out, retval] long* pRetVal);` | `HRESULT get_NumHeadsetsInConference([out, retval] SHORT* pVal);` |
| `HRESULT serialNumber([out, retval] SAFEARRAY(unsigned char)* pRetVal);` | `HRESULT get_SerialNumber([out,retval] VARIANT* pVal);` |
| `HRESULT Proximity([out, retval] Proximity* pRetVal);` | `HRESULT get_Proximity([out, retval] DeviceProximity* pVal);` |
| `HRESULT Name([out, retval] BSTR* pRetVal);` | `HRESULT get_HeadsetName([out, retval] BSTR* pVal);` |
| `HRESULT BatteryInfo([out, retval] IBTBatteryInfo** pRetVal);` | `HRESULT get_BatteryInfo([out, retval] ICOMBatteryInfo** pVal);` |

### *11.4.12: Mobile Presence Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **_MobilePresenceEventArgs** | **ICOMMobilePresenceEventArgs** |
| `ToString, Equals, GetHashCode, GetType` | Not supported, this is .NET legacy |
| `HRESULT CallerID([out, retval] BSTR* pRetVal);` | `HRESULT get_CallerId([out, retval] BSTR* pVal);` |
| `HRESULT State([out, retval] MobileCallState* pRetVal);` | `HRESULT get_MobileCallState([out, retval] DeviceMobileCallState* pVal)` |

### *11.4.13: Base Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|
| **_BaseEventArgs** | **ICOMBaseEventArgs** |
| `ToString, Equals, GetHashCode, GetType` | Not supported, this is .NET legacy |
| `HRESULT BaseEvent([out, retval] BaseEventTypeExt* pRetVal);` | `HRESULT get_EventType([out, retval] BaseEventTypeExt* pVal);` |
| `HRESULT Report([out, retval] SAFEARRAY(unsigned char)* pRetVal);` | Not supported |
| `HRESULT FeatureLock([out, retval] FeatureLock* pRetVal);` | `HRESULT get_FeatureLock([out, retval] DeviceFeatureLock* pVal);` |
| `HRESULT Password([out, retval] SAFEARRAY(unsigned char)* pRetVal);` | `HRESULT get_Password([out, retval] VARIANT* pVal);` |
| `HRESULT serialNumber([out, retval] SAFEARRAY(unsigned char)* pRetVal);` | `HRESULT get_SerialNumber ([out, retval] VARIANT* pVal);` |
| `HRESULT Count([out, retval] unsigned long* pRetVal);` | Not supported |

### *11.4.14: Device Listener Event Args*

| 2.x COM Interfaces | v3.6 COM Interfaces |
|---|---|

| | ICOMDeviceListenerEventArgs |
|---|---|
| ToString, Equals, GetHashCode, GetType | Not supported, this is .NET legacy |
| HRESULT DeviceEventType([out, retval] DeviceEventType* pRetVal); | HRESULT get_DeviceEventType([out, retval] COMDeviceEventType* pVal); |
| HRESULT HeadsetButtonPressed([out, retval] HeadsetButton* pRetVal); | HRESULT get_HeadsetButton([out, retval] DeviceHeadsetButton* pVal); |
| HRESULT HeadsetStateChange([out, retval] HeadsetStateChange* pRetVal); | HRESULT get_HeadsetStateChange([out, retval] DeviceHeadsetStateChange* pVal); |
| HRESULT BaseButtonPressed([out, retval] BaseButton* pRetVal); | HRESULT get_BaseButton([out, retval] DeviceBaseButton* pVal); |
| HRESULT BaseStateChange([out, retval] BaseStateChange* pRetVal); | HRESULT get_BaseStateChange([out, retval] DeviceBaseStateChange* pVal); |
| HRESULT ATDStateChange([out, retval] ATDStateChange* pRetVal); | HRESULT get_ATDStateChange([out, retval] DeviceATDStateChange* pVal); |
| HRESULT DialedKey([out, retval] long* pRetVal); | HRESULT get_DialedKey([out, retval] SHORT* pVal); |

## 11.5: Interfaces Added since Plantronics SDK v3.0

The interfaces in the following sections are new beginning with Plantronics SDK v3.0.

### *11.5.1: ICOMUserPreference*

| |
|---|
| HRESULT get_MediaPlayerActionIncomingCall([out,retval] BSTR* action); |
| HRESULT put_MediaPlayerActionIncomingCall([in] BSTR action); |
| HRESULT get_MediaPlayerActionEndedCall([out,retval] BSTR* action); |
| HRESULT put_MediaPlayerActionEndedCall([in] BSTR action); |
| HRESULT get_RingPCAndHS([out,retval] VARIANT_BOOL* value); |
| HRESULT put_RingPCAndHS([in] VARIANT_BOOL value); |
| HRESULT get_DefaultSoftphone([out,retval] BSTR* value); |

```
HRESULT put_DefaultSoftphone([in] BSTR value);
HRESULT get_EscalateToVoiceSoftPhone([out,retval] BSTR* value);
HRESULT put_EscalateToVoiceSoftPhone([in] BSTR value);
HRESULT get_KeepLinkUp([out,retval] VARIANT_BOOL* value);
HRESULT put_KeepLinkUp([in] VARIANT_BOOL value);
HRESULT get_AutoPresence([out,retval] VARIANT_BOOL* value);
HRESULT put_AutoPresence(VARIANT_BOOL value);
HRESULT get_DonAction([out,retval] BSTR* value);
HRESULT put_DonAction([in] BSTR value);
HRESULT get_DoffAction([out,retval] BSTR* action);
HRESULT put_DoffAction([in] BSTR action);
```

### *11.5.2: ICOMCallInfo*

```
HRESULT get_CallId([out, retval] LONG* pVal);
HRESULT get_SessionId([out, retval] BSTR* pVal);
HRESULT get_CallSource([out, retval] BSTR* pVal);
HRESULT get_CallActive([out, retval] VARIANT_BOOL* pVal);
```

### *11.5.3: ICOMDeviceSettings*

```
HRESULT get_VOIPRing([out, retval] COMRingTone* pVal);
HRESULT put_VOIPRing([in] COMRingTone newVal);
HRESULT get_VOIPBandwidth([out, retval] COMAudioBandwidth* pVal);
HRESULT put_VOIPBandwidth([in] COMAudioBandwidth newVal);
HRESULT get_VOIPRingVolume([out, retval] COMVolumeLevel* pVal);
HRESULT put_VOIPRingVolume([in] COMVolumeLevel newVal);
```

| |
|---|
| HRESULT get_VOIPToneLevel([out, retval] COMToneLevel* pVal); |
| HRESULT put_VOIPToneLevel([in] COMToneLevel newVal); |
| HRESULT get_PSTNRing([out, retval] COMRingTone* pVal); |
| HRESULT put_PSTNRing([in] COMRingTone newVal); |
| HRESULT get_PSTNBandwidth([out, retval] COMAudioBandwidth* pVal); |
| HRESULT put_PSTNBandwidth([in] COMAudioBandwidth newVal); |
| HRESULT get_PSTNRingVolume([out, retval] COMVolumeLevel* pVal); |
| HRESULT put_PSTNRingVolume([in] COMVolumeLevel newVal); |
| HRESULT get_PSTNToneLevel([out, retval] COMToneLevel* pVal); |
| HRESULT put_PSTNToneLevel([in] COMToneLevel newVal); |
| HRESULT get_MobileRing([out, retval] COMRingTone* pVal); |
| HRESULT put_MobileRing([in] COMRingTone newVal); |
| HRESULT get_MobileBandwidth([out, retval] COMAudioBandwidth* pVal); |
| HRESULT put_MobileBandwidth([in] COMAudioBandwidth newVal); |
| HRESULT get_MobileRingVolume([out, retval] COMVolumeLevel* pVal); |
| HRESULT put_MobileRingVolume([in] COMVolumeLevel newVal); |
| HRESULT get_MuteTone([out, retval] COMRingTone* pVal); |
| HRESULT put_MuteTone([in] COMRingTone newVal); |
| HRESULT get_ToneVolume([out, retval] COMVolumeLevel* pVal); |
| HRESULT put_ToneVolume([in] COMVolumeLevel newVal); |
| HRESULT get_MuteToneVolume([out, retval] COMVolumeLevel* pVal); |
| HRESULT put_MuteToneVolume([in] COMVolumeLevel newVal); |
| HRESULT get_ActiveCallRing([out, retval] COMActiveCallRing* pVal); |
| HRESULT put_ActiveCallRing([in] COMActiveCallRing newVal); |

| |
|---|
| HRESULT get_AntiStartleEnabled([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_AntiStartleEnabled([in] VARIANT_BOOL newVal); |
| HRESULT get_AudioLimit([out, retval] COMAudioLimit* pVal); |
| HRESULT put_AudioLimit([in] COMAudioLimit newVal); |
| HRESULT get_G616Enabled([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_G616Enabled([in] VARIANT_BOOL newVal); |
| HRESULT get_OTAEnabled([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_OTAEnabled([in] VARIANT_BOOL newVal); |
| HRESULT get_IntellistandEnabled([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_IntellistandEnabled([in] VARIANT_BOOL newVal); |
| HRESULT get_PowerMode([out, retval] COMPowerLevel* pVal); |
| HRESULT put_PowerMode([in] COMPowerLevel newVal); |
| HRESULT get_TWAPeriod([out, retval] DSPTWAPeriod* pVal); |
| HRESULT put_TWAPeriod([in] DSPTWAPeriod newVal); |
| HRESULT get_PhoneLine([out, retval] COMLineType* pVal); |
| HRESULT put_PhoneLine([in] COMLineType newVal); |
| HRESULT get_BTInterfaceEnabled([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_BTInterfaceEnabled([in] VARIANT_BOOL newVal); |
| HRESULT get_BTAutoConnectEnabled([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_BTAutoConnectEnabled([in] VARIANT_BOOL newVal); |
| HRESULT get_ACLPollingEnabled([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_ACLPollingEnabled([in] VARIANT_BOOL newVal); |
| HRESULT get_BTVoiceCommandEnabled([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_BTVoiceCommandEnabled([in] VARIANT_BOOL newVal); |

| |
|---|
| HRESULT get_PasswordProtected([out, retval] VARIANT_BOOL* pVal); |
| HRESULT get_DECTMode([out, retval] VARIANT_BOOL* pVal); |
| HRESULT RestoreDefaultSettings(void); |
| HRESULT SetData([in] COMCustomData dataType, [in] SAFEARRAY* buffer); |

### *11.5.4: ICOMAdvanceSettings*

| |
|---|
| HRESULT get_DialTone([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_DialTone([in] VARIANT_BOOL newVal); |
| HRESULT get_AudioSensing([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_AudioSensing([in] VARIANT_BOOL newVal); |
| HRESULT get_AnswerOnDon([out, retval] VARIANT_BOOL* pVal); |
| HRESULT put_AnswerOnDon([in] VARIANT_BOOL newVal); |
| HRESULT get_AutoTransfer([out, retval] DeviceSensorControl* pVal); |
| HRESULT put_AutoTransfer([in] DeviceSensorControl newVal); |
| HRESULT get_AutoDisconnect([out, retval] DeviceSensorControl* pVal); |
| HRESULT put_AutoDisconnect([in] DeviceSensorControl newVal); |
| HRESULT get_AutoReject([out, retval] DeviceSensorControl* pVal); |
| HRESULT put_AutoReject([in] DeviceSensorControl newVal); |
| HRESULT get_LockHookSwitch([out, retval] DeviceSensorControl* pVal); |
| HRESULT put_LockHookSwitch([in] DeviceSensorControl newVal); |
| HRESULT get_AutoPause([out, retval] DeviceSensorControl* pVal); |
| HRESULT put_AutoPause([in] DeviceSensorControl newVal); |
| HRESULT get_VoicePrompt([out, retval] DeviceSensorControl* pVal); |
| HRESULT put_VoicePrompt([in] DeviceSensorControl newVal); |
| HRESULT get_DialToneActive([out, retval] VARIANT_BOOL* pVal); |

## 11.6: Interfaces Added in Plantronics SDK v3.4

These interfaces were added in support of DA80/DA90 products.

### *11.6.1: IDeviceSettingExt*

For more description of this interface, read section 5.8: IDeviceSettingsExt Interface.

### *11.6.2: Structs*

#### 11.6.2.1: AALAcousticIncidentReportingEventArgs

For more information on this struct, read section **Error! Reference source not found. Error! Reference source not found.**

#### 11.6.2.2: AALTWAReportEventArgs

For more information about this struct, read section **Error! Reference source not found. Error! Reference source not und.**

#### 11.6.2.3: ConversationDynamiceReportEventArgs

For more information about this struct, read section **Error! Reference source not found. Error! Reference source not und.**

## 11.7: Interfaces Added in Plantronics SDK v3.6

No new interfaces are added in Plantronics SDK v3.6.

## 11.8: Changes to iPlugin Support

If you have an app based on the *iPlugin* API, you will need to port it to "COM Service .NET API" because *iPlugin* support is deprecated starting from Plantronics SDK v3.0.

Please read the following article for the full details, including the steps needed to accomplish this porting. A free Plantronics Developer Connection membership is required to view this article.

Porting iPlugin app to Standalone app with Spokes SDK

In summary, these are the key steps for porting from an *iPlugin* app to a standalone app:

Key steps to port from iPlugin app to standalone app:

- Remove the following references: `Plantronics.Device.Common, Plantronics.UC.Common`
- Change the project type from DLL to Console application (or WinForms or WPF as you require)
- Change the target .NET Framework from 3.5 to 4.0 or higher (not client profile)
- In the code: remove *iPlugin*, add a main function as entry point (refer to the code in the linked article above)
- Set the “Startup Object” in the application properties to the "Program" class, so it can find the *main* function
- In the *main* function, grab Spokes singleton (see article), register for some events, call Connect (this wires up Spokes event to the code), then wait for Enter key press to quit.

# Chapter 12: Appendix E: REST in Plantronics SDK v3.x: Migration Guide

This chapter provides guidelines for migrating HTTP applications from Spokes 2.x REST to Plantronics SDK v3.x REST. The migration consists of two steps:

1. Migrate to REST in Plantronics SDK JavaScript Web client. Replace/add the following files from REST in Plantronics SDK v2.x JavaScript SDK:
   - `jquery-1.7.2.js`
   - `spokes.js`
   - `json2.js`
2. Remove references to deprecated JavaScript methods from your application. The following features are deprecated:
   - Device list
   - Plug-in list
   - Call hand off

## 12.1: Multiple Sessions Handling

The REST plug-in supports multiple device sessions. The following methods are provided to work with device sessions:

- `/DeviceServices/Attach?uid={uid}`
- `/DeviceServices/Release?sess={sess}`

## 12.2: Listing of URI Changes

The following table illustrates the differences between the old URI scheme and a new one. As before all URLs starts with `http://localhost:port/Spokes`:

| Services | Old URIs | New URIs |
|---|---|---|
| /DeviceServices | /DeviceList | deprecated |
| | /{uid}/Attach | /Attach?uid={uid} |
| | /{sess}/Release | /Release?sess={sess} |
| | /{sess}/Info | /Info |
| | /{sess}/Events?queue={queue} | /Events?sess={sess}&queue={queue} |
| | Not implemented | /Proximity?sess={sess}&enabled={enabled} |
| | Not implemented | /ATDMobileCallerId?sess={sess} |
| | /{sess}/Ring?enabled={enabled} | /Ring?enabled={enabled} |
| | /{sess}/AudioState?state={state} | /AudioState?state={state} |
| /CallServices | /CallManagerState | /CallManagerState |
| | /Events | /Events |
| | /{name}/SessionManagerCallEvents | /SessionManagerCallEvents?name={name} |
| | /{name}/CallEvents | /CallEvents?name={name} |
| | /{name}/CallRequests | /CallRequests?name={name} |
| | /{name}/IncomingCall?callID={callID}&contact={contact}&tones={tones}&route={route} | /IncomingCall?name={name}&callID={callID}&contact={contact}&tones={tones}&route={route} |
| | /{name}/OutgoingCall?callID={callID}&contact={contact}&route={route} | /OutgoingCall?name={name}&callID={callID}&contact={contact}&route={route} |
| | /{name}/TerminateCall?callID={callID} | /TerminateCall?name={name}&callID={callID} |

| | | |
|---|---|---|
| | /{name}/AnswerCall?callID={callID} | /AnswerCall?name={name}&callID={callID} |
| | /{name}/HoldCall?callID={callID} | /HoldCall?name={name}&callID={callID} |
| | /{name}/ResumeCall?callID={callID} | /ResumeCall?name={name}&callID={callID} |
| | /{name}/MuteCall?callID={callID}&muted={muted} | /MuteCall?name={name}&callID={callID}&muted={muted} |
| | /{name}/InsertCall?callID={callID}&contact={contact} | /InsertCall?name={name}&callID={callID}&contact={contact} |
| | /{name}/SetAudioRoute?callID={callID}&route={route} | /SetAudioRoute?name={name}&callID={callID}&route={route} |
| | /{name}/SetConferenceId?callID={callID} | /SetConferenceId?name={name}&callID={callID} |
| | /{name}/CallHandOff?enable={enable} | Deprecated |
| | /{name}/MakeCall?contact={contact} | /MakeCall?name={name}&contact={contact} |
| /SessionManager | /PluginList | /PluginList |
| | /Register/{name} | /Register?name={name} |
| | /UnRegister/{name} | /UnRegister?name={name} |
| | /IsActive/{name}?active={active} | /IsActive?name={name}&active={active} |
| /EventManager | /GetRegistry?sess={sess} | /GetRegistry?sess={sess} |
| | /SetRegistry?sess={sess}&queue={queue} | /SetRegistry?sess={sess}&queue={queue} |
| | /AddRegistry?sess={sess}&queue={queue} | /AddRegistry?sess={sess}&queue={queue} |
| | /RemoveRegistry?sess={sess}&queue={queue} | /RemoveRegistry?sess={sess}&queue={queue} |

| | | |
|---|---|---|
| | /GlobalTTL?sess={sess}&ttl={ttl} | /GlobalTTL?sess={sess}&ttl={ttl} |
| | /GlobalMaxCount?sess={sess}&max={max} | /GlobalMaxCount?sess={sess}&max={max} |
| | /TTL?sess={sess}&queue={queue}&ttl={ttl} | /TTL?sess={sess}&queue={queue}&ttl={ttl} |
| | /MaxCount?sess={sess}&queue={queue}&max={max} | /MaxCount?sess={sess}&queue={queue}&max={max} |

Note: Because only one device is supported by Plantronics SDK v3.6, the *DeviceManager* interface and */DeviceServices/DeviceList* are deprecated.

Note: As opposed to Spokes 2.x, */DeviceServices/Info* does not need a session to be created. It always works, just as *DeviceList* did in Spokes 2.x.

The following methods also do not need a session to be created because only one device is supported:

- `/DeviceServices/Ring?enabled={enabled}`
- `/DeviceServices/AudioState?state={state}`

The following methods accept *sessionID* as their first parameter:

- `/DeviceServices/Events?sess={sess}&queue={queue}`
- `/DeviceServices/Proximity?sess={sess}&enabled={enabled}`
- `/DeviceServices/ATDMobileCallerId?sess={sess}`
- `/DeviceServices/Release?sess={sess}`
- `/EventManager/GetRegistry?sess={sess}`
- `/EventManager/SetRegistry?sess={sess}&queue={queue}`
- `/EventManager/AddRegistry?sess={sess}&queue={queue}`
- `/EventManager/RemoveRegistry?sess={sess}&queue={queue}`
- `/EventManager/GlobalTTL?sess={sess}&ttl={ttl}`
- `/EventManager/GlobalMaxCount?sess={sess}&max={max}`
- `/EventManager/TTL?sess={sess}&queue={queue}&ttl={ttl}`
- `/EventManager/MaxCount?sess={sess}&queue={queue}&max={max}`

## 12.3: New features for the REST Service starting in Plantronics SDK v3.0:

Proximity events can be received over REST in Plantronics SDK v3.6. Proximity shows whether Bluetooth device is close to or far from the computer. If device supports Proximity, it is possible to monitor proximity events over REST in Plantronics SDK. Use `Device.events` / `Device.headsetEvents` methods. Proximity events are reported as `HeadsetStateChanged` events. It is also possible to enable or disable proximity reporting on a device.

Call Services are available through IE browser (`json2.js` script contains the fix).

ATD event mobile caller ID can be received over the REST. It will be collected with `Device.events/Device.atdEvents`.

All REST URLs that formerly were formatted:

```
localhost:32001/Spokes/…/{sess}/method?parameter={parameter}
```

now have the format:

```
localhost:32017/Spokes/…/method?sess={sess}&parameter={parameter}
```

# Chapter 13: Appendix F: Delivering the Plantronics SDK Runtime

This is how you will deliver the Plantronics SDK runtime for use with a third-party application. When you run the Plantronics SDK Bootstrapper it will do the following:

1. Install Spokes, the Plantronics SDK runtime engine.
2. Deliver the files needed for application development.
3. Deliver the Plantronics SDK native runtime installer file (`SpokesSDKNativeRuntime.msi`).
4. Deliver the Plantronics SDK runtime installer file (`SpokesSDKRuntime.msi`).
5. Deliver the Plantronics SDK startup installer file (`SpokesSDKStartup.msi`).

When you deliver your application you will need to create an installer for your pieces (`MyApplication.msi` for example) and then run your installer and one or more of our installers sequentially.

## 13.1: SpokesSDKRuntime.msi

This installer database delivers the core Spokes functionality to a target computer. Among other things, this includes the Spokes client executable, the COM and REST plug-ins, and `Spokes.dll`. The COM plug-in is registered. Any third party application that creates a Plantronics SDK COM object will cause the Spokes client to run.

## 13.2: SpokesSDKNativeRuntime.msi

TBD

## 13.3: SpokesSDKStartup.msi

This database will make changes on the target computer to start the Spokes client and have the client start when the computer starts. This is useful for applications that communicate with the Spokes client via the REST interface since the client needs to be running to service REST API calls. If your application uses the Plantronics SDK COM interface then there is no need to install this MSI file.

## 13.4: Chaining MSI files

At Plantronics we use WIX 3.7 to build our installer databases and we use WIX's Burn tool to chain those MSI files into a monolithic bootstrapper executable. If we assume that the MSI file delivering a third party application is called `MyApplication.msi` then the complete application can be delivered by installing `Application.msi`, then installing `SpokesSDKRuntime.msi` and (optionally) installing `SpokesSDKStartup.msi`. A simple Burn script to perform these actions would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Bundle Name="Bootstrapper1" Version="1.0.0.0" UpgradeCode="Put GUID here">
    <BootstrapperApplicationRef Id="WixStandardBootstrapperApplication.RtfLicense" />
    <Chain>
      <MsiPackage
```

```
          SourceFile="Application.msi"
          Compressed="yes" />
        <MsiPackage
          SourceFile="c:\Program Files\Plantronics\Spokes SDK\SpokesSDKRuntime.msi"
          Compressed="yes" />
        <MsiPackage
          SourceFile="c:\Program Files\Plantronics\Spokes SDK\SpokesSDKStartup.msi"
          Compressed="yes" />
      </Chain>
    </Bundle>
  </Wix>
```

Note that the “compress” option causes the MSI to be stored as a resource inside the bootstrapper executable.

If you use Visual Studio 2010 or greater, it is pretty easy to create a bootstrapper project and add files to it. When this project is built the end result is an executable with all the MSI files stored as resources inside it that can be distributed to the end user.

Since the full Spokes installer also delivers `SpokesSDKNativeRuntime.msi`, `SpokesSDKRuntime.msi` and `SpokesSDKStartup.msi`, it can be installed on the same computer as the third party application without causing any conflict. If the bootstrapper determines that a package it is to deliver is already there, it just skips that package and moves on to the next one.

# Chapter 14: Glossary

**Table 2: Glossary**

| Term | Description |
|---|---|
| ATD | Abbreviation for Alternate Telephony Device, which is a device that transports communications services by means of an Internet Protocol (IP) network rather than a public switched telephone network. Also known as a VoIP (Voice over Internet Protocol) device. The device may be a desk phone, a smartphone, or an Internet device. |
| Call Manager | Manages call control functionality (multiple calls, hold, redial, switch channels, and so on). |
| COM Interface | Plantronics SDK provides a COM interface. COM is a binary-interface standard for software components developed by Microsoft. This interface represents a single, unified API and makes the same level of functionality available to both managed and unmanaged COM applications. |
| Contextual Intelligence | Leverage the contextual information available from Plantronics devices, including proximity, location and identity, and use this information to add value to software applications. |
| Device Listener | Provides a consistent interface across all Plantronics devices so that applications need not handle device-specific events and commands. |
| Device Manager | API to provide device level interfaces to events, commands, and properties. |
| HID | Abbreviation for Human Interface Driver. Provides support for devices such as mice, joysticks, and keyboards, as well as sometimes providing support for simple buttons and indicators on other types of devices. It is designed to provide a low latency link, with low power requirements. |
| Native interface | The Plantronics SDK Native interface is a programming framework that enables Plantronics SDK API code to call and be called by programs specific to a hardware and operating system platform and libraries written in other languages. |
| Interop interface | The Plantronics SDK Interop interface is a programming framework that enables Plantronics SDK API code to call and be called by programs regardless of their hardware and operating system platform. |

| | |
|---|---|
| PDC | Plantronics Developer Connection. Visit http://developer.plantronics.com/community/sdk for links to resources and downloads that will help you quickly and easily get up and running with the Plantronics SDK. |
| PlantronicsDevices.xml | Configuration file for all custom device handlers. |
| Plug-ins | Plug-ins are a set of software components that add specific abilities to a larger software application. For Plantronics SDK, these are client applications that implement the *IPlugin* interface and follow the Plug-in usage model. Plug-ins should only be used in very select situations, such as in support of each of the device types in an organization or in support of specific media players. |
| Proximity | The device user's proximity to their computer system. |
| PSTN | Public Switched Telephone Network (PSTN) interface. An option to ATD, PSTN is the network of the world's public circuit-switched telephone networks. It consists of telephone lines, fiber optic cables, microwave transmission links, cellular networks, communications satellites, and undersea telephone cables, all inter-connected by switching centers, thus allowing any telephone in the world to communicate with any other. Originally a network of fixed-line analog telephone systems, the PSTN is now almost entirely digital in its core and includes mobile as well as fixed telephones. |
| Spokes.config | Configuration file for Plantronics SDK plug-in applications. |
| REST | REpresentational State Transfer (REST) is an HTTP-based architecture style for designing distributed applications. |
| SDK | Plantronics Software Development Kit v3.6 |
| Session Manager | Helps applications manage incoming calls by handling client requests and maintaining a list of client sessions. |
| Spokes | Runtime engine for Plantronics SDK |

| | |
|---|---|
| UC | Unified communications (UC) is the integration of real-time communication services such as instant messaging (chat), presence information, telephony (including IP telephony), video conferencing, data sharing (including web connected electronic whiteboards), call control and speech recognition with non-real-time communication services such as unified messaging (integrated voicemail, e-mail, SMS and fax). UC is not necessarily a single product, but a set of products that provides a consistent unified user interface and user experience across multiple devices and media types. |
| USB HID | USB stands for "Universal Serial Bus", and refers to a common port on computers. Some mobile phones also use USB or "mini-usb" connections to transfer data.<br><br>USB human interface device class (USB HID class) is a part of the USB specification for computer peripherals: it specifies a device class (a type of computer hardware) for human interface devices such as keyboards, mice, game controllers and alphanumeric display devices.<br><br>See Also: HID. |
| VoIP | Voice Over Internet Protocol (VoIP) interface. VoIP commonly refers to the communication protocols, technologies, methodologies, and transmission techniques involved in the delivery of voice communications and multimedia sessions over Internet Protocol (IP) networks, such as the Internet. |
| Windows HID | Windows Human Interface Driver (HID), or HID over I2C, which is used for embedded devices in Microsoft Windows 8. |