TAPI Developers Guide for Cisco Unified Communications Manager Release 7.1(2)

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

Text Part Number: OL-18532-01

CONTENTS

Preface

This chapter describes the purpose, intended audience, and organization of this document and describes the conventions that convey instructions and other information. It contains the following topics:

Purpose

This document describes the Cisco Unified TAPI implementation by detailing the functions that comprise the implementation software and illustrating how to use these functions to create applications that support the Cisco Unified Communications hardware, software, and processes. You should use this document with the Cisco Unified Communications Manager manuals to develop applications.

Audience

Cisco intends this document to be for use by telephony software engineers who are developing Cisco telephony applications that require TAPI. This document assumes that the engineer is familiar with both the C or C++ languages and the Microsoft TAPI specification.

This document assumes that you have knowledge of C or C++ languages and the Microsoft TAPI specification.You must also have knowledge or experience in the following areas:

- Extensible Markup Language (XML)
- Hypertext Markup Language (HTML)
- Hypertext Transport Protocol (HTTP)
- Socket programming

- TCP/IP Protocol
- Web Service Definition Language (WSDL) 1.1
- Secure Sockets Layer (SSL)

In addition, as a user of the Cisco Unified Communications Manager APIs, you must have a firm understanding of XML Schema. For more information about XML Schema, refer to http://www.w3.org/TR/xmlschema-0/.

You must have an understanding of Cisco Unified Communications Manager and its applications. See the "Related Documentation" section on page xvi for Cisco Unified Communications Manager documents and other related technologies.

Organization

Chapter	Description
Chapter 1, "Overview"	Outlines key concepts for Cisco Unified TAPI and lists all functions that are available in the implementation.
Chapter 2, "New and Changed Information"	Provides a list new and changed features release–by–release of Cisco Unified Communications Manager.
Chapter 4, "Cisco Unified TAPI Installation"	Provides installation procedures for Cisco Unified TAPI and Cisco Unified TSP.
Chapter 5, "Basic TAPI Implementation"	Describes the supported functions in the Cisco implementation of standard Microsoft TAPI v2.1.
Chapter 6, "Cisco Device-Specific Extensions"	Describes the functions that comprise the Cisco hardware-specific implementation classes.
Chapter 7, "Cisco Unified TAPI Examples"	Provides examples that illustrate the use of the Cisco Unified TAPI implementation.
Appendix A, "Message Sequence Charts"	Lists possible call scenarios and use cases.
Appendix B, "Cisco Unified TAPI Interfaces"	Lists APIs that are supported or not supported.
Appendix C, "Troubleshooting Cisco Unified TAPI"	Describes troubleshooting techniques.
Appendix D, "Cisco Unified TAPI Operations-by-Release"	Lists features, line functions, messages, and structures; phone functions, messages, and structures that have been added or modified by Cisco Unified Communications Manager release.
Appendix E, "CTI Supported Devices"	Lists CTI supported devices.

Related Documentation

This section lists documents and URLs that provide information on Cisco Unified Communications Manager, Cisco Unified IP Phones, TAPI specifications, and the technologies that are required to develop applications.

- Cisco Unified Communications Manager Release 7.1(2)—A suite of documents that relate to the installation and configuration of Cisco Unified Communications Manager. Refer to the *Cisco Unified Communications Manager Documentation Guide for Release* 7.1(2) for a list of documents on installing and configuring Cisco Unified Communications Manager 7.1(2), including:
  - *Cisco Unified Communications Manager Administration Guide, Release* 7.1(2).
  - *Cisco Unified Communications Manager System Guide, Release* 7.1(2).
  - *Cisco Unified Communications Manager Features and Services Guide, Release* 7.1(2).
  - *Cisco Unified Communications Manager Release Notes, Release* 7.1(2).
- *Cisco Unified IP Phones and Services*—A suite of documents that relate to the installation and configuration of Cisco Unified IP Phones.
- *Cisco Distributed Director*—A suite of documents that relate to the installation and configuration of Cisco Distributed Director.

For more information about TAPI specifications, creating an application to use TAPI, or TAPI administration, see the following documents:

- Microsoft TAPI 2.1 Features:
  http://www.microsoft.com/ntserver/techresources/commnet/tele/tapi21.asp
- Getting Started with Windows Telephony
  http://www.microsoft.com/NTServer/commserv/deployment/planguides/getstartedtele.asp
- Windows Telephony API (TAPI)
  http://www.microsoft.com/NTServer/commserv/exec/overview/tapiabout.asp
- Creating Next Generation Telephony Applications:
  http://www.microsoft.com/NTServer/commserv/techdetails/prodarch/tapi21wp.asp
- The Microsoft Telephony Application Programming Interface (TAPI) Programmer's Reference
- "For the Telephony API, Press 1; For Unimodem, Press 2; or Stay on the Line"—A paper on TAPI by Hiroo Umeno, a COMM and TAPI specialist at Microsoft.

  http://www.microsoft.com/msj/0498/tapi.aspx
- "TAPI 2.1 Microsoft TAPI Client Management"
- "TAPI 2.1 Administration Tool"

Developer Support

The Developer Support Program provides formalized support for Cisco Systems interfaces to enable developers, customers, and partners in the Cisco Service Provider solutions Ecosystem and Cisco AVVID Partner programs to accelerate their delivery of compatible solutions.

The Developer Support Engineers are an extension of the product technology engineering teams. They have direct access to the resources necessary to provide expert support in a timely manner.

For additional information on this program, refer to the Developer Support Program web site at http://developer.cisco.com.

Conventions

This document uses the following conventions:

<table>
<tr><th>Convention</th><th>Description</th></tr>
<tr><td>boldface font</td><td>Commands and keywords are in boldface.</td></tr>
<tr><td>italic font</td><td>Arguments for which you supply values are in italics.</td></tr>
<tr><td>[ ]</td><td>Elements in square brackets are optional.</td></tr>
<tr><td>{ x | y | z }</td><td>Alternative keywords are grouped in braces and separated by vertical bars.</td></tr>
<tr><td>[x | y | z]</td><td>Optional alternative keywords are grouped in brackets and separated by vertical bars.</td></tr>
<tr><td>string</td><td>An unquoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.</td></tr>
<tr><td><code>screen</code> font</td><td>Terminal sessions and information that the system displays are in <code>screen</code> font.</td></tr>
<tr><td><code>boldface screen</code> font</td><td>Information you must enter is in <code>boldface screen</code> font.</td></tr>
<tr><td>italic screen font</td><td>Arguments for which you supply values are in italic screen font.</td></tr>
<tr><td>→</td><td>This pointer highlights an important line of text in an example.</td></tr>
<tr><td>^</td><td>The symbol ^ represents the key labeled Control—for example, the key combination ^D in a screen display means hold down the Control key while you press the D key.</td></tr>
<tr><td>< ></td><td>Nonprinting characters, such as passwords are in angle brackets.</td></tr>
</table>

Notes use the following conventions:

Note Means *reader take note*. Notes contain helpful suggestions or references to material not covered in the publication.

Tip Means *the following information might help you solve a proble*m.

Timesaver Means the *described action saves tim*e. You can save time by performing the action described in the paragraph.

Obtaining Documentation and Submitting a Service Request

For information on obtaining documentation, submitting a service request, and gathering additional information, see the monthly What's New in Cisco Product Documentation, which also lists all new and revised Cisco technical documentation, at:

http://www.cisco.com/en/US/docs/general/whatsnew/whatsnew.html

Subscribe to the What's New in Cisco Product Documentation as a Really Simple Syndication (RSS) feed and set content to be delivered directly to your desktop using a reader application. The RSS feeds are a free service and Cisco currently supports RSS Version 2.0.

Cisco Product Security Overview

This product contains cryptographic features and is subject to United States and local country laws governing import, export, transfer and use. Delivery of Cisco cryptographic products does not imply third-party authority to import, export, distribute or use encryption. Importers, exporters, distributors and users are responsible for compliance with U.S. and local country laws. By using this product you agree to comply with applicable laws and regulations. If you are unable to comply with U.S. and local laws, return this product immediately.

A summary of U.S. laws governing Cisco cryptographic products may be found at: http://www.cisco.com/wwl/export/crypto/tool/stqrg.html.

If you require further assistance please contact us by sending email to export@cisco.com.

OpenSSL/Open SSL Project

The following link provides information about the OpenSSL notice:

http://www.cisco.com/en/US/products/hw/phones/ps379/products_licensing_information_listing.html

CHAPTER 1

Overview

This chapter describes the major concepts of Cisco Unified TAPI service provider (Cisco Unified TSP) implementation. It contains the following sections:

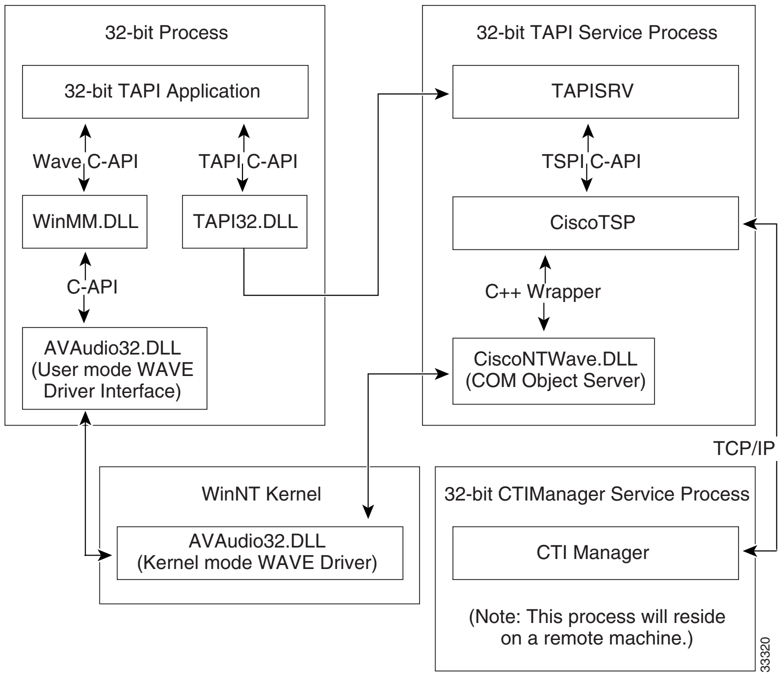
Cisco Unified TSP Overview

The standard TAPI provides an unchanging programming interface for different implementations. The goal of Cisco in implementing TAPI for the Cisco Unified Communications Manager platform remains to conform as closely as possible to the TAPI specification, while providing extensions that enhance TAPI and expose the advanced features of Cisco Unified Communications Manager to applications.

As versions of Cisco Unified Communications Manager and Cisco Unified TSP are released, variances in the API should be minor and should tend in the direction of compliance. Cisco stays committed to maintaining its API extensions with the same stability and reliability, though additional extensions may be provided as new Cisco Unified Communications Manager features become available.

Figure 1-1 shows the architecture of TAPI.

Figure 1-1 Architecture of TAPI Service Process



Note The Cisco TSP is a TAPI 2.1 service provider.

Cisco Unified TSP Concepts

The following are described in this section:

- Basic TAPI Applications, page 1-2
- Cisco TSP Components, page 1-3
- Cisco Wave Drivers, page 1-3
- TAPI Debugging, page 1-4
- Cisco TSP Components, page 1-3

See "Basic TAPI Implementation" section on page 5-1 and "Cisco Device-Specific Extensions" section on page 6-1 for lists and descriptions of interfaces and extensions.

Basic TAPI Applications

Microsoft has defined some basic APIs which can be invoked/supported from application code. All Microsoft defined APIs that can be used from the TAPI applications are declared in TAPI.H file. TAPI.H file is a standard library file that is with the VC++/VS2005 Installation. For example, C:\Program Files\Microsoft Visual Studio\VC98\Include\TAPI.H.

To use any specific API which is added or provided by Cisco TSP, the application needs to invoke that API by using the LineDevSpecific API.

Simple Application

```
#include <tapi.h>
#include <string>
#include "StdAfx.h"
class TapiLineApp {
LINEINITIALIZEEXPARAMS mLineInitializeExParams;//was declared in TAPI.h files
     HLINEAPP    mhLineApp;
     DWORD       mdwNumDevs;
     DWORD dwAPIVersion = 0x20005

public:
    // App Initialization
    // Note hInstance can be NULL
    // appstr - value can be given the app name "test program"
    bool TapiLineApp::LineInitializeEx(HINSTANCE hInstance, std::string appStr)
{
    unsigned long lReturn = 0;
    mLineInitializeExParams.dwTotalSize = sizeof(mLineInitializeExParams);
    mLineInitializeExParams.dwOptions = LINEINITIALIZEEXOPTION_USEEVENT;
    lReturn = lineInitializeEx (&mhLineApp, hInstance, NULL, appStr.c_str),
&mdwNumDevs,&dwAPIVersion,&LineInitializeExParams);
    if ( lReturn == 0 ) {
        return true;
    }
    else {
        return false;
    }
}
//App shutdown
bool TapiLineApp::LineShutdown()
{
    return! (lineShutdown (mhLineApp));
}
};
```

Cisco TSP Components

The following are Cisco TSP components:

- CiscoTsp001.tsp – TAPI service implementation provided by Cisco TSP
- CTIQBE over TCP/IP – Cisco protocol used to monitor and control devices and lines
- CTI Manager Service – Manages CTI resources and connections to devices. Exposed to 3rd-party applications via Cisco TSP and/or JTAPI API

Cisco Wave Drivers

Cisco TSP can be configured to provide either first or third-party call control. In First-Party Call Control, the audio stream is terminated by the application. Ordinarily, this is done using the Cisco Wave Driver. AVAudio32.dll implements the wave interfaces for the Cisco wave drivers. In Third-Party Call control, the audio stream termination is done by the actual physical device like an IP phone or a group of IP phones for which your application is responsible.

For information about the installation of the wave drivers, see Installing the Wave Driver, page 4-13.

TAPI Debugging

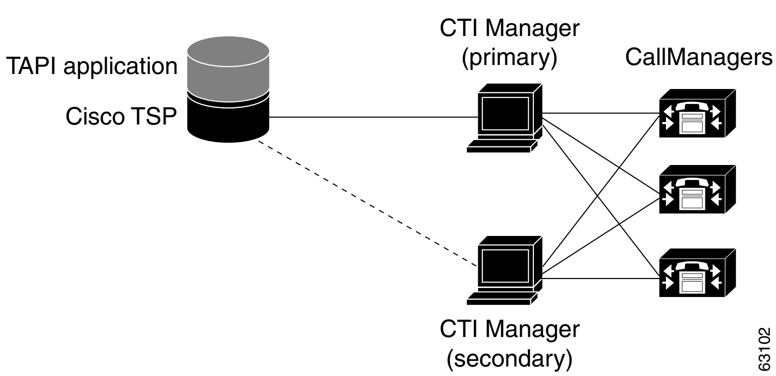
The TAPI browser is a TAPI debugging application. It can be downloaded from the Microsoft MSDN Web site at ftp://ftp.microsoft.com/developr/TAPI/tb20.zip. The TAPI browser can be used to initialize TAPI, for use by TAPI developers to test a TAPI implementation and to verify that the TSP is operational.

CTI Manager (Cluster Support)

The CTI Manager, along with the Cisco Unified TSP, provide an abstraction of the Cisco Unified Communications Manager cluster that allows TAPI applications to access Cisco Unified Communications Manager resources and functionality without being aware of any specific Cisco Unified Communications Manager. The Cisco Unified Communications Manager cluster abstraction also enhances the failover capability of CTI Manager resources. A failover condition occurs when a node fails, a CTI Manager fails, or a TAPI application fails, as illustrated in Figure 1-2.

Note Cisco does not support CTI device monitoring or call control with 3rd-party devices.

Figure 1-2 Cluster Support Architecture



Cisco Unified Communications Manager Failure

When a Cisco Unified Communications Manager node in a cluster fails, the CTI Manager recovers the affected CTI ports and route points by reopening these devices on another Cisco Unified Communications Manager node. When the failure is first detected, Cisco Unified TSP sends a PHONE_STATE (PHONESTATE_SUSPEND) message to the TAPI application.

When the CTI port/route point is successfully reopened on another Cisco Unified Communications Manager, Cisco Unified TSP sends a phone PHONE_STATE (PHONESTATE_RESUME) message to the TAPI application. If no Cisco Unified Communications Manager is available, the CTI Manager waits until an appropriate Cisco Unified Communications Manager comes back in service and tries to open the device again. The lines on the affected device also go out of service and in service with the corresponding LINE_LINEDEVSTATE (LINEDEVSTATE_OUTOFSERVICE) and LINE_LINEDEVSTATE (LINEDEVSTATE_INSERVICE) events Cisco Unified TSP sends to the TAPI application. If for some reason the device or lines cannot be opened, even when all Cisco Unified Communications Managers come back in service, the system closes the devices or lines, and Cisco Unified TSP will send PHONE_CLOSE or LINE_CLOSE messages to the TAPI application.

When a failed Cisco Unified Communications Manager node comes back in service, CTI Manager “re-homes” the affected CTI ports or route points to their original Cisco Unified Communications Manager. The graceful re-homing process ensures that the re-homing only starts when calls are no longer being processed or are active on the affected device. For this reason, the re-homing process may not finish for a long time, especially for route points, which can handle many simultaneous calls.

When a Cisco Unified Communications Manager node fails, phones currently re-home to another node in the same cluster. If a TAPI application has a phone device opened and the phone goes through the re-homing process, CTI Manager automatically recovers that device, and Cisco Unified TSP sends a PHONE_STATE (PHONESTATE_SUSPEND) message to the TAPI application. When the phone successfully re-homes to another Cisco Unified Communications Manager node, Cisco Unified TSP sends a PHONE_STATE (PHONESTATE_RESUME) message to the TAPI application.

The lines on the affected device also go out of service and in service, and Cisco Unified TSP sends LINE_LINEDEVSTATE (LINEDEVSTATE_OUTOFSERVICE) and LINE_LINEDEVSTATE (LINEDEVSTATE_INSERVICE) messages to the TAPI application.

Call Survivability

When a device or Cisco Unified Communications Manager failure occurs, no call survivability exists; however, media streams that are already connected between devices will survive. Calls in the process of being set up or modified (transfer, conference, redirect) simply get dropped.

CTI Manager Failure

When a primary CTI Manager fails, Cisco Unified TSP sends a PHONE_STATE (PHONESTATE_SUSPEND) message and a LINE_LINEDEVSTATE (LINEDEVSTATE_OUTOFSERVICE) message for every phone and line device that the application opened. Cisco Unified TSP then connects to a backup CTIManager. When a connection to a backup CTI Manager is established and the device or line successfully reopens, the Cisco Unified TSP sends a PHONE_STATE (PHONESTATE_RESUME) or LINE_LINEDEVSTATE (LINEDEVSTATE_INSERVICE) message to the TAPI application. If the Cisco Unified TSP is unsuccessful in opening the device or line for a CTI port or route point, the Cisco Unified TSP closes the device or line by sending the appropriate PHONE_CLOSE or LINE_CLOSE message to the TAPI application.

After Cisco Unified TSP is connected to the backup CTIManager, Cisco Unified TSP will not reconnect to the primary CTIManager until the connection is lost between Cisco Unified TSP and the backup CTIManager.

If devices are added to or removed from the user while the CTI Manager is down, Cisco Unified TSP generates PHONE_CREATE/LINE_CREATE or PHONE_REMOVE/LINE_REMOVE events, respectively, when connection to a backup CTI Manager is established.

Cisco Unified TAPI Application Failure

When a Cisco TAPI application fails (the CTI Manager closes the provider), calls at CTI ports and route points that have not yet been terminated get redirected to the Call Forward On Failure (CFF) number that has been configured for them. The system routes new calls into CTI Ports and Route Points that are not opened by an application to their CFNA number.

QoS Support

Cisco Unified TSP supports the Cisco baseline for baselining of Quality of Service (QoS). Cisco Unified TSP marks the IP DSCP (Differentiated Services Code Point) for QBE control signals that flow from TSP to CTI with the value of the Service parameter “DSCP IP for CTI Applications” that CTI sends in the ProviderOpenCompletedEvent. The Cisco TAPI Wave driver marks the RTP packets with the value that CTI sends in the StartTransmissionEvent. The system stores the DSCP value received in the StartTransmissionEvent in the DevSpecific portion of the LINECALLINFO structure, and fires the LINECALLINFOSTATE_DEVSPECIFIC event with the QoS indicator.

Note QoS information is not available if you begin monitoring in the middle of a call because existing calls do not have an RTP event.

Presentation Indication (PI)

There is a need to separate the presentability aspects of a number (calling, called, and so on) from the actual number itself. For example, when the number is not to be displayed on the IP phone, the information might still be needed by another system, such as Unity VM. Hence, each number/name of the display name needs to be associated with a Presentation Indication (PI) flag, which will indicate whether the information should be displayed to the user or not.

You can set up this feature as follows:

On a Per-Call Basis

You can use Route Patterns and Translation Patterns to set or reset PI flags for various partyDNs/Names on a per-call basis. If the pattern matches the digits, the PI settings that are associated with the pattern will be applied to the call information.

On a Permanent Basis

You can configure a trunk device with “Allow” or “Restrict” options for parties. This will set the PI flags for the corresponding party information for all calls from this trunk.

Cisco Unified TSP supports this feature. If calls are made via Translation patterns with all of the flags set to Restricted, the system sends the CallerID/Name, ConnectedID/Name, and RedirectionID/Name to applications as Blank. The system also sets the LINECALLPARTYID flags to Blocked if both the Name and Party number are set to Restricted.

When developing an application, be sure only to use functions that the Cisco TAPI Service Provider supports. For example, the Cisco TAPI Service Provider supports transfer, but not fax detection. If an application requires an unsupported media or bearer mode, the application will not work as expected.

Cisco Unified TSP does not support TAPI 3.0 applications.

Call Control

You can configure Cisco Unified TSP to provide first- or third-party call control.

First-Party Call Control

In first-party call control, the application terminates the audio stream. Ordinarily, this occurs by using the Cisco wave driver. However, if you want the application to control the audio stream instead of the wave driver, use the Cisco device-specific extensions.

Third-Party Call Control

In third-party call control, the control of an audio stream terminating device is not “local” to the Cisco Unified Communications Manager. In such cases, the controller might be the physical IP phone on your desk or a group of IP phones for which your application is responsible.

Note Cisco does not support CTI device monitoring or call control with 3rd-party devices.

CTI Port

For first-party call control, a CTI port device must exist in the Cisco Unified Communications Manager. Because each port can only have one active audio stream at a time, most configurations only need one line per port.

A CTI port device does not actually exist in the system until you run a TAPI application and a line on the port device is opened requesting LINEMEDIAMODE_AUTOMATEDVOICE and LINEMEDIAMODE_INTERACTIVEVOICE. Until the port is opened, anyone who calls the directory number that is associated with that CTI port device receives a busy or reorder tone.

The IP address and UDP port number is either specified statically (the same IP address and UDP port number is used for every call) or dynamically. By default, CTI ports use static registration.

Dynamic Port Registration

Dynamic Port Registration enables applications to specify the IP address and UDP port number on a call-by-call basis. Currently, the IP address and UDP port number are specified when a CTI port registers and is static through the life of the registration of the CTI port. When media is requested to be established to the CTI port, the system uses the same static IP address and UDP port number for every call.

An application that wants to use Dynamic Port Registration must specify the IP address and UDP port number on a call before invoking any features on the call. If the feature is invoked before the IP address and UDP port number are set, the feature will fail, and the call state will be set depending on when the media time-out occurs.

CTI Route Point

You can use Cisco Unified TAPI to control CTI route points. CTI route points allow Cisco Unified TAPI applications to redirect incoming calls with an infinite queue depth. This allows incoming calls to avoid busy signals.

CTI route point devices have an address capability flag of LINEADDRCAPFLAGS_ROUTEPOINT. When your application opens a line of this type, it can handle any incoming call by disconnecting, accepting, or redirecting the call to some other directory number. The basis for redirection decisions can be caller ID information, time of day, or other information that is available to the program.

Media Termination at Route Point

The Media Termination at Route Point feature lets applications terminate media at route points. This feature enables applications to pass the IP address and port number where they want the call at the route point to have media established.

The system supports the following features at route points:

- Answer
- Multiple Active Calls
- Redirect
- Hold
- UnHold
- Blind Transfer
- DTMF Digits
- Tones

Monitoring Call Park Directory Numbers

The Cisco Unified TSP supports monitoring calls on lines that represent Call Park Directory Numbers (Call Park DNs). The Cisco Unified TSP uses a device-specific extension in the LINEDEVCAPS structure that allows TAPI applications to differentiate Call Park DN lines from other lines. If an application opens a Call Park DN line, all calls that are parked to the Call Park DN get reported to the application. The application cannot perform any call control functions on any calls at a Call Park DN.

To open Call Park DN lines, you must check the **Monitor Call Park DNs** check box in Cisco Unified Communications Manager User Administration for the Cisco Unified TSP user. Otherwise, the application will not perceive any of the Call Park DN lines upon initialization.

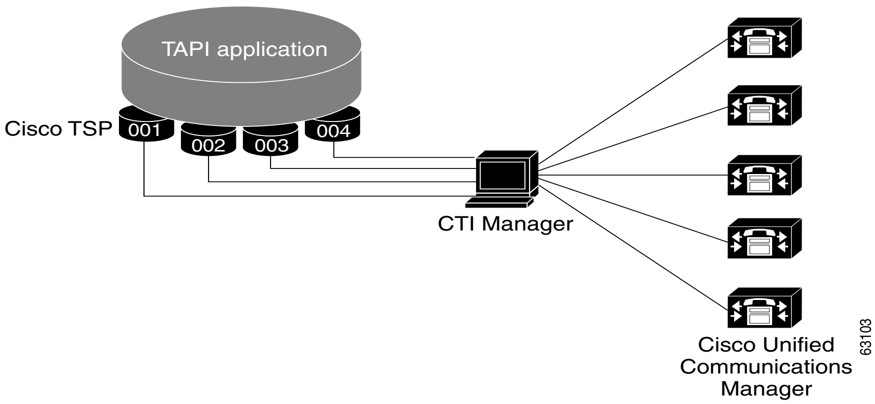
Multiple Cisco Unified TSPs

In the Cisco Unified TAPI solution, the TAPI application and Cisco Unified TSP get installed on the same machine. The Cisco Unified TAPI application and Cisco Unified TSP do not directly interface with each other. A layer written by Microsoft sits between the TAPI application and Cisco Unified TSP. This layer, known as TAPISRV, allows the installation of multiple TSPs on the same machine, and it hides that fact from the Cisco Unified TAPI application. The only difference to the TAPI application is that it is now informed that there are more lines that it can control.

Consider an example—assume that Cisco Unified TSP1 exposes 100 lines, and Cisco Unified TSP2 exposes 100 lines. In the single Cisco Unified TSP architecture where Cisco Unified TSP1 is the only Cisco Unified TSP that is installed, Cisco Unified TSP1 would tell TAPISRV that it supports 100 lines, and TAPISRV would tell the application that it can control 100 lines. In the multiple Cisco Unified TSP architecture, where both Cisco Unified TSPs are installed, this means that Cisco Unified TSP1 would tell TAPISRV that it supports 100 lines, and Cisco Unified TSP2 would tell TAPISRV that it supports 100 lines. TAPISRV would add the lines and inform the application that it now supports 200 lines. The application communicates with TAPISRV, and TAPISRV takes care of communicating with the correct Cisco Unified TSP.

Ensure that each Cisco Unified TSP is configured with a different username and password that you administer in the Cisco Unified Communications Manager Directory. Configure each user in the Directory, so devices that are associated with each user do not overlap. Each Cisco Unified TSP in the multiple Cisco Unified TSP system does not communicate with the others. Each Cisco Unified TSP in the multiple Cisco Unified TSP system creates a separate CTI connection to the CTI Manager as shown in Figure 1-3. Multiple Cisco Unified TSPs help in scalability and higher performance.

Figure 1-3 Multiple Cisco Unified TSPs Connect to CTI Manager



CTI Device/Line Restriction

With CTI Device/Line restriction implementation, a CTIRestricted flag is be placed on device or line basis. When a device is restricted, it assumes that all its configured lines are restricted.

Cisco Unified TSP does not report any restricted devices and lines back to application. When a CTIRestricted flag is changed from Cisco Unified Communications Manager Administration, Cisco Unified TSP treats it as normal device/line add or removal.

CHAPTER 2

New and Changed Information

This chapter describes new and changed Cisco Unified TAPI Service Provider (TSP) information for Cisco Unified Communications Manager release 7.1.(2) and feature supported in the previous releases. This chapter contains the following sections:

- Cisco Unified Communications Manager Release 7.1(2), page 2-1
- Features Supported in Previous Releases, page 2-2

Refer to the programming guides Web site for prior Cisco JTAPI Developer Guides at http://www.cisco.com/en/US/products/sw/voicesw/ps556/products_programming_reference_guides_list.html.

Cisco Unified Communications Manager Release 7.1(2)

This section describes new and changed features that are supported in Cisco Unified Communications Manager Release 7.1(2) and contains the following topics:

- IPv6 Support—Enables IPv6 capabilities in a Cisco Unified Communications Manager (Unified CM) network. For more information, see IPv6 Support, page 3-2.
- Direct Transfer Across Lines Support—Allows the application to directly transfer calls across the lines that are configured on the device. For more information, see Direct Transfer Across Lines Support, page 3-2.
- Message Waiting Indicator Enhancement—The Message Waiting Indicator (MWI) feature enchancement enables the application to display the following information on the supported phones. For more information, see Message Waiting Indicator Enhancement, page 3-3.
- Swap and Cancel Softkey Support—Swap and Control softkey support has been provided the Cisco Unfied IP Phone 7900 Series. For more information, see Swap and Cancel Softkey Support, page 3-4.
- Drop-Any-Party Support—Enables the application to drop any call from the ad-hoc conference. For more information, see Drop-Any-Party Support, page 3-5.
- Park Monitoring Support—Allows you to monitor the status of parked calls. For more information, see Park Monitoring Support, page 3-5.
- Logical Partitioning Support—Restricts VoIP to PSTN calls and vice versa, based on the logical partitioning policy. For more information, see Logical Partitioning Support, page 3-7.

- Support for Cisco Unfied IP Phone 6900 Series—a new user group called Standard allow CTI control devices with roll-over mode is created. When a user is added to the new user group, the TSP applications monitor and control the Cisco Unfied IP Phone 6900 series and Cisco Unfied IP Phone 7931 with roll-over mode. For more information, see Support for Cisco Unfied IP Phone 6900 Series, page 3-7.
- The following Attendant Console versions are supported in Cisco TSP 7.1(2):
  - Cisco Unified Department Attendant Console release 2.0.x, 3.1.x, 8.0.x
  - Cisco Unified Business Attendant Console 2.0.x, 3.1.x, 8.0.x
  - Cisco Unified Enterprise Attendant Console 3.1.x, 8.0.x
  - Arc Enterprise 4.x, 5.x
  - Arc Enterprise Premium 4.x, 5.x
  - Arc Call Connect 4.x, 5.x

Features Supported in Previous Releases

This section describes the features supported in the releases prior to 7.1(2) and contains the following sections:

Cisco Unified Communications Manager Release 7.0(1)

This section describes new and changed features supported in Cisco Unified Communications Manager Release 7.0(1) and contains the following:

Note For the features, Join Across Lines, Do Not Disturb-Reject, and Calling Party Normalization, each TAPI application must be upgraded to a version that is compatible with these features. Additionally, if you are upgrading from Release 5.1 and you use Join Across Lines, the Conference Chaining feature must not be enabled or used until all applications are either upgraded to a version compatible with the new CUCM version. Also, you should verify that the applications are not impacted by the Conference Chaining feature.

Cisco Unified Communications Manager Release 6.1(x)

This section describes new and changed features that Cisco Unified Communications Manager Release 6.1(x) supports and contains the following topic:

• Join Across Lines (SCCP), page 3-11

Cisco Unified Communications Manager Release 6.0(1)

This section describes new and changed features that are supported in Cisco Unified Communications Manager Release 6.0(1), and contains the following topics:

• Intercom Support, page 3-11
• Secure Conferencing Support, page 3-12
• Do Not Disturb, page 3-13
• Conference Enhancements, page 3-14
• Arabic and Hebrew Language Support, page 3-15
• Additional Features Supported on SIP Phones, page 3-16
• Silent Monitoring, page 3-16
• Silent Recording, page 3-17
• Calling Party IP Address, page 3-18

Backward Compatibility

No backward compatibility issues exist for any features that are introduced in Cisco Unified Communications Manager Release 6.0(1).

Cisco Unified Communications Manager Release 5.1

This section describes new and changed features supported in Cisco Unified Communications Manager, Release 5.1 and contains the following topics:

• Partition Support, page 3-18
• Alternate Script, page 3-19
• Secure RTP, page 3-19
• SuperProvider, page 3-20
• Refer and Replaces for Phones that are Running SIP, page 3-21
• SIP URL Address, page 3-22
• 3XX, page 3-22
• Secure TLS Support, page 3-22

Cisco Unified Communications Manager Release 5.0

This section describes new and changed features that are supported in Cisco Unified Communications Manager, Release 5.0, and contains the following topics:

Cisco Unified Communications Manager Release 4.x

This section describes new and changed features that are supported in Cisco Unified Communications Manager, Release 4.x, and contains the following topics:

Release 4.0

Release 4.1

Cisco Unified Communications Manager Releases Prior to 4.x

The chapter includes the following list of all features that are available in the Cisco Unified TSP implementation of Cisco Unified Communications Manager, prior to Release 4.x:

- Silent Install Support, page 3-33

CHAPTER 3

Features Supported by TSP

This chapter describes the features that Cisco Unified TAPI Service Provider (TSP) supports. See Chapter 2, "New and Changed Information," for described features, which are listed by Cisco Unified Communications Manager release.

IPv6 Support

The IPv6 support feature enables IPv6 capabilities in a Cisco Unified Communications Manager (Unified CM) network. IPv6 increases the number of addresses available for network devices. TAPI can connect to Unified CM with IPv6 support if the IPv6 Support feature is enabled on Unified CM. IPv6 enhancements include the following:

- Provides the IPv6 address of the calling party to the called partyin theDevspecific part of LINECALLINFO.
- Support to register a CTI port or a route point with an IPv6 address. The RTP destination address also contains IPv6 addresses if the same is involved in media establishment.

The TSP user interface includes the primary and backup CTI Manager address and a flag that indicates the preference of user while connecting to the CTI Manager. CTI ports and route points can be registered with IPv4, IPv6, or both.

The following new CiscoLineDevSpecific functions allow the application to specify IP address mode and IPv6 address before opening CTI port and route point:

- CciscoLineDevSpecificSetIPv6AddressAndMode
- CciscoLineDevSpecificSetRTPParamsForCallIPv6

For dynamic port registration, on receiving the SLDSMT_OPEN_LOGICAL_CHANNEL event, the CciscoLineDevSpecificSetRTPParamsForCallIPv6 allows the application to provide IPv6 information for the call.

Interface Changes

See Set IP Address Mode, page 6-45 and Set IPv6 Address, page 6-46.

Message Sequences

See IPv6 Use Cases, page A-88.

Backward Compatibility

This feature is backward compatible. The 0x00090000 extension must be negotiated to use this feature.

Direct Transfer Across Lines Support

The Direct Transfer Across Lines feature allows the application to directly transfer calls across the lines that are configured on the device. The application monitors both the lines when directly transferring the calls across the lines.

A new LineDevSpecific extension, CciscoLineDevSpecificDirectTransfer, is added to direct transfer calls across the lines or on the same line. The 0x00090000 extension must be negotiated to use CciscoLineDevSpecificDirectTransfer.

Interface Changes

See Direct Transfer, page 6-48.

Message Sequences

See Direct Transfer Across Lines, page A-96.

Backward Compatibility

This feature is backward compatible.

Message Waiting Indicator Enhancement

The Message Waiting Indicator (MWI) feature enchancement enables the application to display the following information on the supported phones:

- Total number of new voice messages (normal and high priority messages)
- Total number of old voice messages (normal and high priority messages)
- Number of new high priority voice messages
- Number of old high priority voice messages
- Total number of new fax messages (normal and high priority messages)
- Total number of old fax messages (normal and high priority messages)
- Number of new high priority fax messages
- Number of old high priority fax messages

MWI also includes two CCiscoLineDevSpecific subclasses are added to enhance the MWI functionality. Similar to the existing setMessageWaiting operation, one MWI operation sets the summary information for the controlled line, while the another MWI operation sets the message summary information on any line that is reachable by the controlled line, as defined by the configured calling search space of the controlled line.

Interface Changes

See Message Summary, page 6-20 and Message Summary Dirn, page 6-22.

Message Sequences

There are no message sequences for this feature.

Backward Compatibility

This feature is backward compatible.

Swap and Cancel Softkey Support

The following softkeys have been added to the Cisco Unfied IP Phone 7900 Series:

- Swap
- Cancel

Swap

The Swap softkey can be only be used when you use the Transfer or Conference feature. When you press Swap, the phone puts the consultative call on hold and resumes the primary call. Swap operation breaks the internal linkage between the primary and consultative calls, but you can still complete the transfer or conference operation.

Cancel

When you press Cancel before completing the transfer operation, the TSP receives an event notification from CTI and cancels any pending transfer or conference requests.

The Swap and Cancel features operate as follows:

- For swap operation, the primary call state is changed to CONNECTED, and the consult call state is changed to ONHOLD.
- For cancel operation, the primary call state is changed to ONHOLD, and consult call state remains as CONNECTED.
- To complete the transfer operation after swap or cancel, the application invokes LineCompleteTransfer or CciscoLineDevSpecificDirectTransfer.
- To complete the conference operation after swap or cancel, the application invokes Cisco Join API – CCiscoLineDevSpecificJoin.

When using the Swap and Cancel features, the Cisco Unified IP Phones maintain a consulting relationship between the primary and the consulting calls, on invoking consult transfer or consult conference:

- The Swap operation puts the active call on hold and retrieves the held call.
- The Cancel operation breaks the consulting relationship between the primary and the consulting calls.

When users perform the swap operation, the behavior remains the same while resuming calls and all pending transfer or conference operation are cancelled. Users can swap or toggle during consultative transfer or conference transactions, and also swap or toggle between call sessions during the transaction to check the status of each party.

Interface Changes

There are no interface changes for this feature.

Message Sequences

See Swap or Cancel Support, page A-103.

Backward Compatibility

This feature is backward compatible.

Drop-Any-Party Support

The Drop-Any-Party feature enables the application to drop any call from the ad-hoc conference. This feature is currently supported from the phone interface. The application uses the LineRemoveFromConference function to drop the call from a conference. When the call is dropped from a conference, TSP receives CtiDropConferee as the call state change cause, and this is sent to TAPI as the default cause.

Interface Changes

See lineRemoveFromConference, page 5-44.

Message Sequences

See Drop Any Party, page A-125.

Backward Compatibility

This feature is backward compatible. The 0x00090000 extension is added to maintain backward compatibility.

Park Monitoring Support

The Park Monitoring feature allows you to monitor the status of parked calls. This feature improves the user experience of retrieving the parked calls. When TAPI receives a parked call notification, a call representing the parked call is generated, and the call is set to CONNECTED INACTIVE state. The parked call is set to IDLE when it is retrieved or forwarded to Park Monitoring Forward No Retrieve Destination.

DEVSPECIFIC_PARK_STATUS event is sent when call is parked, reminded, retrieved, and aborted. LineDevSpecific SLDST_SET_STATUS_MESSAGES are enhanced to allow the application to enable/disable DEVSPECIFIC_PARK_STATUS event.

When Cisco TSP receives the LINE_PARK_STATUS event for the newly parked call, Cisco TSP simulates a call for each of the newly parked call using the SubID received from the LINE_PARK_STATUS event, and notifies the application about the new parked call using the LINE_NEWCALL event.

Cisco TSP uses LINE_CALLSTATE event to notify changes in the park status to the application. The park status in the LINE_CALLSTATE event can be one of the following:

- Parked - indicates a call is parked by the TSP monitored Cisco Unified IP phone.
- Retrieved - indicates a previously parked call is retrieved.
- Abandoned - indicates a previously parked call is disconnected while waiting to be retrieved.
- Reminder - indicates the park monitoring reversion timer for the parked call has expired.
- Forwarded - indicates the parked call has been forwarded to the configured Park Monitoring Forward No Retrieve destination, or if the FNR destination is not configured, the call is forwarded back to the parker.

When Cisco TSP receives the LINE_PARK_STATUS event, it maps the existing CALLINFO structure with the fields received from LINE_PARK_STATUS event. The application then retrieves the updated structure by invoking lineGetCallInfo.

The mapping of the fields in the LINE_PARK_STATUS event to the LINECALLINFO structure is as follows:

LINE_PARK_STATUS	LINECALLINFO--	Description
LineHandle	hline	Identifies the line handle to which this message applies
GCID	dwcallid	Identifies the global call handle to which this message applies.
TransactionID	dwRedirectingName	A unique ID that identifies a particular parked call
CallReason	dwReason	Identifies the call reason.
Park Status	dwBearerMode	Parked, retrieved, abandoned, reminder, forwarded -indicates the status of the parked call.
ParkSlotDN	dwCallerID	The park slot DN.
ParkSlotPartition	dwCallerIDName	The partition of the park slot DN.
ParkedPartyDN	dwCalledID	The parked party DN.
ParkedPartyPartition	dwCalledIDName	The partition of the parked party DN.

To maintain the existing behavior of the Park feature for the Cisco Unfied IP Phones running SIP, you can set the value of the Park Monitoring Forward No Retrieve Destination timer equal to the existing Call Park Duration timer and leave the Park Monitoring Forward No Retrieve Destination blank.

To override the Park Monitoring feature for the Cisco Unfied IP Phones running SIP, turn off the DEVSPECIFIC_PARK_STATUS message flag by using the lineDevSpecific SLDST_SET_STATUS_MESSAGES request.

Interface Changes

See Set Status Messages, page 6-26.

Message Sequences

See Park Monitoring, page A-138.

Backward Compatibility

This feature is backward compatible.

Logical Partitioning Support

The Logical Partitioning feature restricts VoIP to PSTN calls and vice versa, based on the logical partitioning policy. Any request that interconnects a VOIP call to a PSTN call or vice versa in two different geographical locations fails and the error code is sent back to the applications.

The device, device pool, trunk, and gateway pages now provide configuration to select geo-location values and construction rules for geo-location strings.

A new enterprise parameter has been added for this feature with the following values:

- Name: Logical partitioning enabled
- Values: True or False
- Default: False

A new error code has been added for this feature:
LINEERR_INVALID_CALL_PARTITIONING_POLICY 0xC000000C

Interface Changes

There are no interface changes for this feature.

Message Sequences

See Logical Partitioning Support, page A-149.

Backward Compatibility

This feature is backward compatible. To maintain earlier behavior, set the logical partitioning enabled parameter to **False**.

Support for Cisco Unfied IP Phone 6900 Series

Cisco Unfied IP Phone 6900 Series phones behave similar to theCisco Unfied IP Phone 7931 with roll-over mode. Both the Cisco Unfied IP Phone 7931 with roll-over mode and the Cisco Unfied IP Phone6900 Series are currently restricted and you cannot control them from TSP applications.

For the Cisco Unfied IP Phone 6900 series, a new user group called Standard allow CTI control devices with roll-over mode is created. When a user is added to the new user group, the TSP applications monitor and control the Cisco Unfied IP Phone 6900 series and Cisco Unfied IP Phone 7931 with roll-over mode. For the transfer and conference features, the new consult is created on the second line depending on the roll-over type and maximum calls on the line. On setup transfer or setup conference, the call goes to ONHOLD state instead of OnholdPendingTransfer or OnholdPendingConference, when the consult call is rolled over to the second line. The application uses Direct Transfer Across Lines (DTAL) or Join Across Lines (JAL) to complete the transfer or conference operations.

Interface Changes

There are no interface changes for this feature.

Message Sequences

See Support for Cisco IP Phone 6900 Series, page A-153.

Backward Compatibility

This feature is backward compatible.

To maintain existing behavior, remove the user from the user group, standard allow CTI control devices with roll-over mode.

Join Across Lines (SIP)

This feature allows two or more calls on different lines of the same device to be joined by using the join operation. Applications can use the existing join API to perform the task. When the join across line happens, the consultation call on the different line on which the survival call does not reside will get cleared, and a CONFERENCED call that represents the consultation call will get created on the primary line where conference parent is created. This feature should have no impact when multiple calls are joined on the same line.

This feature is supported both on SCCP and SIP devices that can be controlled by CTI. In addition, this feature also supports chaining of conference calls on different lines on the same device. Also, a join across line can be performed on a non-controller (the original conference controller was on a different device then where the join is being performed) line.

This feature returns an error if one of the lines involved in the Join Across Lines is an intercom line.

Interface Changes

None.

Message Sequences

See Join Across Lines, page A-74.

Backwards Compatibility

This feature is backward compatible.

Localization Infrastructure Changes

Beginning with Release 7.0(1), the TSP localization is automated. The TSP UI can download the new and updated locale files directly from a configured TFTP server location. As a result of the download functionality, Cisco TSP install supports only the English language during the installation.

During installation, the user inputs the TFTP server IP address. When the user opens the TSP interface for the **first time**, the TSP interface automatically downloads the locale files from the configured TFTP server and extracts those files to the resources directory. The languages tab in the TSP preferences UI also provides functionality to update the locale files.

Note To upgrade from Cisco Unified Communications Manager, Release 6.0(1) TSP to Cisco Unified Communications Manager, Release 7.0(1) TSP, you must ensure that Release 6.0(1) TSP was installed by using English. If Release 6.0(1) TSP is installed using any language other than English and if the user upgrades to Release 7.0(1) TSP, then the user must manually uninstall Release 6.0(1) TSP from Add/Remove programs in control panel and then perform a fresh install of Release 7.0(1) TSP.

Interface Changes

None.

Message Sequences

None.

Backward Compatibility

Only English locale is packaged in Cisco TSP installer. The TSP UI downloads the locale files from the configured TFTP server. The end user can select the required and supported locale after the installation.

Do Not Disturb–Reject

Do Not Disturb (DND) enhancements support the rejection of a call. The enhancement Do Not Disturb–Reject (DND–R) enables the user to reject any calls when necessary. Prior to the Cisco Unified Communications Manager Release 7.0(1), DND was available only with the Ringer Off option. If DND was set, the call would still get presented but without ringing the phone.

To enable DND–R, access the Cisco Unified Communications Manager Administration phone page or the user can enable it on the phone.

However, if the call has an emergency priority set, the incoming call is presented on the phone even if the DND–R option is selected. This will make sure that emergency calls are not missed.

Feature priority is introduced and defined in the "enum type" for making calls or redirecting existing calls. The priority is defined as:

```
enum CiscoDoNotDisturbFeaturePriority {
    CallPriority_NORMAL=1
    CallPriority_URGENT=2
    CallPriority_EMERGENCY=3
};
```

Feature priority introduces LineMakeCall as part of DevSpecific data. Currently the following structure is supported in DevSpecific data for LineMakeCall:

```
typedef struct LineParams {
    DWORD FeaturePriority;
} LINE_PARAMS;
```

The new Cisco LineDevSpecific extension, CciscoLineRedirectWithFeaturePriority with type SLDST_REDIRECT_WITH_FEATURE_PRIORITY, supports redirected calls with feature priority.

Also in a shared line scenario, if one of the lines is DND–R enabled and if the Remote In Use is true, then it will be marked as connected inactive.

Interface Changes

See lineMakeCall, page 5-34 and Redirect with Feature Priority, page 6-41.

Message Sequences

See Do Not Disturb–Reject, page A-71.

Backward Compatibility

This feature is backward compatible.

Calling Party Normalization

Prior to the Cisco Unified Communication Manager Release 7.0(1), the "+" symbol was not supported. Also, no support existed for displaying the localized or global number of the caller to the called party on its alerting display and the entry into its call directories for supporting a callback without the need of an EditDial.

Cisco Unified Communication Manager Release 7.0(1) adds support for "+" symbol and also the calling number is globalized and passed to the application. This enables the end user to dial back without using EditDial. Along with the globalized calling party, the user would also get the number type of the calling party. This would help the user to know where the call originated, that is, whether it is a SUBSCRIBER, NATIONAL or INTERNATIONAL number.

Interface Changes

See LINECALLINFO, page 6-6.

Message Sequences

See Calling Party Normalization, page A-67.

Backward Compatibility

This feature is backward compatible.

Click to Conference

Click to Conference capability enables users to create conferences from an Instant Messaging (IM) application without creating a consult call first. The Cisco TSP treats the feature as an existing conference model; however, when the conference is created or dropped, the CtiExtendedReason may come as Click2Conference.

Interface Changes

None.

Message Sequences

See Click to Conference, page A-58.

Backward Compatibility

This feature is backward compatible.

Microsoft Windows Vista

Microsoft Windows Vista operating system supports Cisco TSP client with the following work around:

- Always perform the initial installation of the Cisco TSP and Cisco Unified Communications Manager TSP Wave Driver as a fresh install.
- If a secure connection to Cisco Unified Communications Manager is used, turn off/disable the Windows Firewall.
- If Cisco Unified Communications Manager TSP Wave Driver is used for inbound audio streaming, turn off/disable the Windows Firewall.

- If Cisco Unified Communications Manager TSP Wave Driver is used for audio streaming, you must disable all other devices in the Sound, Video, and Game Controllers group.

Join Across Lines (SCCP)

This feature allows two or more calls on different lines of the same device to be joined through the join operation. Applications can use the existing join API to perform the task. When the join across line happens, the consultation call on the different line on which the survival call does not reside will get cleared, and a CONFERENCED call that represents the consultation call will be created on the primary line where conference parent is created. This feature should have no impact when multiple calls are joined on the same line.

This feature is supported on SCCP devices that can be controlled by CTI. In addition, this feature also supports chaining of conference calls on different lines on the same device. Also, a join across line can be performed on a non-controller line; that is, the original conference controller was on a different device then where the join is being performed.

Note This feature returns an error if one of the lines that are involved in the Join Across Lines is an intercom line.

Backwards Compatibility

This feature is backward compatible.

Intercom Support

The Intercom feature allows one user to call another user and have the call automatically answered with one-way media from the caller to the called party, regardless of whether the called party is busy or idle.

To ensure that no accidental voice of the called party is sent back to the caller, Cisco Unified Communications Manager implements a 'whisper' intercom, which means that only one-way audio from the caller is connected, but not audio from the called party. The called party must manually press a key to talk back to the caller. A zip-zip (auto-answer) tone will play to the called party before the party can hear the voice of the caller. For intercom users to know whether the intercom is using one-way or two-way audio, the lamp for both intercom buttons appears colored amber for a one-way whisper Intercom and green for two-way audio. For TSP applications, only one RTP event occurs for the monitored party: either the intercom originator or the intercom destination, with the call state as whisper, in the case of a one-way whisper intercom.

TSP exposes the Intercom line indication and Intercom Speeddial information in DevSpecific of LineDevCap. The application can retrieve the information by issuing LineGetDevCaps. In the DevSpecific portion, TSP provides information that indicates (DevSpecificFlag = LINEDEVCAPSDEVSPECIFIC_INTERCOMDN) whether this is the Intercom line. You can retrieve the Intercom speeddial information in the DevSpecific portion after the partition field.

If a CTI port is used for the Intercom, the application should open the CTI port with dynamic media termination. TSP returns LINEERR_OPERATIONUNAVAIL if the Intercom line is opened with static media termination.

Note You cannot use CTI Route Point for the Intercom feature.

The administrator can configure the speed dial and label options from Cisco Unified Communications Manager Administration. However, the application can issue CciscoLineSetIntercomSpeeddial with SLDST_LINE_SET_INTERCOM_SPEEDDIAL to set or reset SpeedDial and Label for the intercom line. The Application setting will overwrite the administrator setting that is configured in the database. Cisco Unified Communications Manager uses the application setting to make the intercom call until the line is closed or until the application resets it. In the case of a Communications Manager or CTIManager failover, CTIManager or Cisco TSP resets the speed dial setting of the previous application. If the application restarts, the application must reset the speed dial setting; otherwise, Cisco Unified Communications Manager will use the database setting to make the intercom call. In any case, if resetting of the speed dial or label fails, the system sends a LINE_DEVSPECIFIC event to indicate the failure. When the application wants to release the application setting and have the speed dial setting revert to the database setting, the application can call CCiscoLineDevSpecificSetIntercomSpeedDial with a NULL value for SpeedDial and Label.

If the speed dial setting is changed, whether due to a change in the database or because the application overwrites the setting, the system generates a LineDevSpecific event to indicate the change. However, the application needs to call CCiscoLineDevSpecificSetStatusMsgs with DEVSPECIFIC_SPEEDDIAL_CHANGED to receive this notification. After receiving the notification, the application can call LineGetDevCaps to find out the current settings of speed dial and label.

Users can initiate an intercom call by pressing the Intercom button at the originator or by issuing a LineMakeCall with a NULL destination if Speedial/Label is configured on the intercom line. Otherwise, LineMakeCall should have a valid Intercom DN.

For an intercom call, a CallAttribute field in LINECALLINFODEVSPECIFIC indicates whether the call is for the intercom originator or the intercom target.

After the intercom call is established, the system sends a zip-zip tone event to the application as a tone-changed event.

Users can invoke a TalkBack at the destination in two ways:

- By pressing the intercom button
- By issuing CciscoLineIntercomTalkback with SLDST_LINE _INTERCOM_TALKBACK

TSP reports the Whisper call state in the extended call state bit as CLDSMT_CALL_WHISPER_STATE. If the call is being put on hold because the destination is answering an intercom call by using talk back, the system reports the call reason CtiReasonTalkBack in the extended call reason field for the held call.

The application cannot set line features, such as set call forwarding and set message waiting, other than to initiate the intercom call, drop the intercom call, or talk back. After the intercom call is established, the system limits call features for the intercom call. For the originator, only LINECALLFEATURE_DROP is allowed. For the destination, the system supports only the LINECALLFEATURE_DROP and TalkBack features for the whisper intercom call. After the intercom call becomes two-audio after the destination initiates talk back, the system allows only LINECALLFEATURE_DROP at the destination.

Speed dial labels support unicode.

Secure Conferencing Support

Prior to release 6.0(1), the security status of each call matched the status for the call between the phone and its immediately connected party, which is a conference bridge in the case of a conference call. No secured conference resource existed, so secure conference calls were not possible.

Release 7.0(1) supports a secured conference resource to enable secure conferencing. The secure conferencing feature lets the administrator configure the Conference bridge resources with either a non-secure mode or an encrypted signaling and media mode. If the bridge is configured in encrypted signaling and media mode, the Conference Bridge will register to Cisco Unified Communications Manager as a secure media resource. This enables the user to create a secure conference session. When the media streams of all participants who are involved in the conference are encrypted, the conference exists in encrypted mode. Otherwise, the conference exists in non-secure mode.

The overall conference security status depends on the least-secure call in the conference. For example, suppose parties A (secure), B (secure), and C (non-secure) are in a conference. Even though the conference bridge can support encrypted sRTP and is using that protocol to communicate with A and B, C remains a non-secure phone. Thus, the overall conference security status is non-secure. Even though the overall conference security status is non-secure, because a secure conference bridge was allocated, all secure phones (in this case, A and B) will receive sRTP keys. TSP informs each participant about the overall call security status. The system provides the overall call security level of the conference to the application in the DEVSPECIFIC portion of LINECALLINFO to indicate whether the conference call is encrypted or non-secure.

The Secure Conferencing feature uses four fields to present the call security status:

```
DWORD CallSecurityStatusOffset;
DWORD CallSecurityStatusSize;
DWORD CallSecurityStatusElementCount;
DWORD CallSecurityStatusElementFixedSize;
```

The offset will point to following structure:

```
typedef struct CallSecurityStatusInfo
{
    DWORD CallSecurityStatus;
} CallSecurityStatusInfo;
```

The system updates LINECALLINFO whenever the overall call security status changes during the call because a secure or non-secure party joins or leaves the conference.

A conference resource gets allocated to the conference creator based on the creator security capability. If no conference resource with the same security capability is available, the system allocates a less-secure conference resource to preserve existing functionality.

When a shared line is involved in the secure conference, the phone that has its line in RIU (remote in use) mode will not show a security status for the call. However, TSP exposes the overall security status to the application along with other call information for the inactive call. This means that TSP also reports the OverallSecurityStatus to all RIU lines. The status will match what is reported to the active line. Applications can decide whether to expose the information to the end user.

Do Not Disturb

The Do Not Disturb (DND) feature lets phone users go into a Do Not Disturb state on the phone when they are away from their phone or simply do not want to answer incoming calls. The phone softkey DND enables and disables this feature.

From theCisco Unified Communications Manager user windows, users can select the DND option DNR (Do Not Ring).

Cisco TSP makes the following phone device settings available for DND functionality:

- DND Option: None/Ringer off
- DND Incoming Call Alert: Beep only/flash only/disable

- DND Timer: a value between 0-120 mins. It specifies a period in minutes to remind the user that DND is active.
- DND enable and disable

Cisco TSP includes DND feature support for TAPI applications that negotiate at least extension version 8.0 (0x00080000).

Applications can only enable or disable the DND feature on a device. Cisco TSP allows TAPI applications to enable or disable the DND feature via the lineDevSpecificFeature API.

Cisco TSP notifies applications via the LINE_DEVSPECIFICFEATURE message about changes in the DND configuration or status. To receive change notifications, an application must enable the DEVSPECIFIC_DONOTDISTURB_CHANGED message flag with a lineDevSpecific SLDST_SET_STATUS_MESSAGES request.

This feature applies to phones and CTI ports. It does not apply to route points.

Conference Enhancements

The Conference feature of Cisco Unified Communication Manager has been enhanced with the following functions:

- Allowing a noncontroller to add another party into an ad hoc conference.

  Applications can issue the lineGetCallStatus against a CONNECTED call of a noncontroller conference participant and check the dwCallFeatures before adding another party into the conference. The application should have the PREPAREADDCONF feature in the dwCallFeatures list if the participant is allowed to add another party.
- Allowing multiple conferences to be chained.

Be aware that these features are only available if the 'Advanced Ad-hoc Conference' service parameter is enabled on the Cisco Unified Communications Manager.

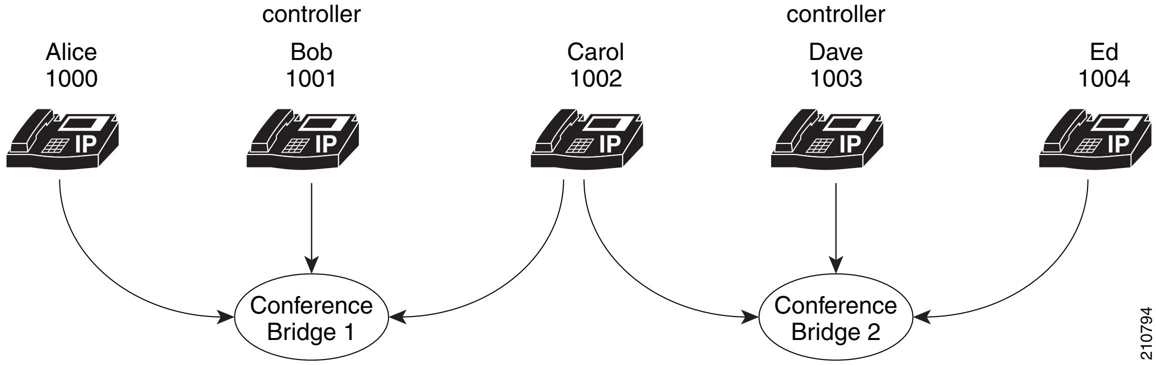
When this service parameter is changed from enabled to disabled, the system no longer allows new chaining between ad hoc conferences. However, existing chained conferences will stay intact. Any participant who is brought into the ad hoc conference by a noncontroller before this change will remain in the conference, but they can no longer add a new participant or remove an existing participant.

To avoid ad hoc conference resources remaining connected together after all real participants have left, Cisco Unified Communications Manager will disallow having more than two conference resources connected to the same ad hoc conference. However, using a star topology to connect multiple conferences could yield better voice quality than a linear topology. A new advanced service parameter, 'Non-linear Ad Hoc Conference Linking Enabled', lets an administrator select the star topology.

A participant can use the conference, transfer, or join commands to chain two conferences together. When two conferences are chained together, each participant only sees the participants from their own conference, and the chained conference appears as a participant with a unique conference bridge name. In other words, participants do not have a full view of the chained conference. The system treats the conferences as two separate conferences, even though all the participants are talking to each other.

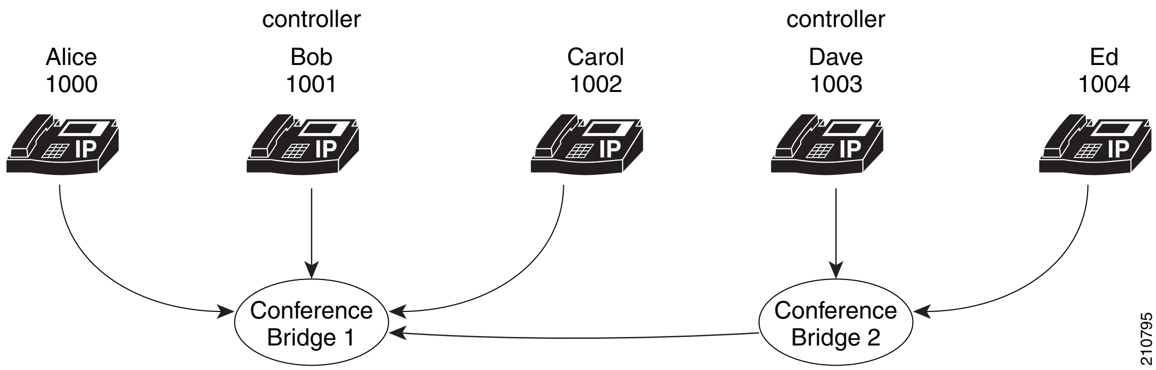
Figure 3-1 shows how TSP presents a conference model in the case of conference chaining. A, B, and C are in conference-1, and C, D, and E are in conference-2. C has an ONHOLD call on conference-1 and an active call on conference-2.

Figure 3-1 Conference Before Join



C then does a join with the primary call from conference-1. For A, B, and C, the conference participants comprise A, B, C, and conference-2. For D and E, the conference participants comprise D, E, and conference-1.

Figure 3-2 Conference After Join



When a user removes a CONFERENCE from its conference list on the phone, the operation actually drops the chained conference bridge. In the previous example, the two chained conferences have been unchained. Conference-1 will remain active and has A, B, and C as participants. However, conference-2 will become a direct call between Dave and Ed because they are the only two parties left in the conference.

Applications can achieve conference chaining by issuing a JOIN or TRANSFER on two separated conference calls. However, a LineCompleteTransfer with a conference option will fail due to a Microsoft TAPI limitation on this standard API. The application can use the Cisco LineDevSpecific extension to issue the join request to chain multiple conferences together.

Arabic and Hebrew Language Support

Users can select Arabic and Hebrew languages during installation and also in the Cisco TSP settings user interface.

Additional Features Supported on SIP Phones

Cisco Unified Communications Manager extends support for phones that are running SIP with these new features:

- PhoneSetLamp (but only for setting the MWI lamp)
- PhoneSetDisplay
- PhoneDevSpecific (CPDST_SET_DEVICE_UNICODE_DISPLAY)
- LineGenerateTone
- Park and UnPark
- The LINECALLREASON_REMINDER reason for CallPark reminder calls
- PhoneGetDisplay (but only after a PhoneSetDisplay)

Note TSP does not pass unicode name for phones that are running SIP.

Silent Monitoring

Silent monitoring is a feature that enables a supervisor to eavesdrop on a conversation between an agent and a customer without the agent detecting the monitoring session. TSP provides monitoring type in line DevSpecific request for applications to monitor calls on a per call basis. Both monitored and monitoring party have to be in controlled list of the user.

The Application is required to send permanent lineID, monitoring Mode and toneDirection as input to CCiscoLineDevSpecificStartCallMonitoring. Only silent monitoring mode is supported and the supervisor cannot talk to the agent.

The Application can specify if a tone should be played when monitoring starts. ToneDirection can be none (no tone played), local (tone played to Agent only), remote (tone played to Customer and Supervisor), both local and remote (tone played to agent, customer, and supervisor).

```
enum PlayToneDirection
{
 PlayToneDirection_LocalOnly = 0,
 PlayToneDirection_RemoteOnly,
 PlayToneDirection_BothLocalAndRemote,
 PlayToneDirection_NoLocalOrRemote
};
```

Monitoring of call which is in connected state on that line will start if the request is successful. This will result in a new call between supervisor and agent. However, the call will be automatically answered with Built-in Bridge (BIB) and agent is not be aware of the call. The call established between supervisor and agent will be one-way audio call. Supervisor will get the mixed stream of agent and customer voices. The application can only invoke the monitoring session for a call after it becomes active.

TSP will send LINE_CALLDEVSPECIFIC (SLDSMT_MONITORING_STARTED) event to the agent call when supervisor starts monitoring the call. TSP will provide monitored party's call attribute information (deviceName, DN, Partition) to the monitoring party in DEVSPECIFIC portion of LINECALLINFO after monitoring has started. Similarly, TSP will provide monitoring party's call attribute information (deviceName, DN, Partition) to the monitored party in devspecific data of LINECALLINFO after monitoring has started.

The monitoring session will be terminated when the agent-customer call is ended by either the agent or the customer. The supervisor can also terminate the monitoring session by dropping the monitoring call.TSP will inform agent by sending LINE_CALLDEVSPECIFIC (SLDSMT_MONITORING_ENDED) when supervisor drops the call. The event will not be sent if monitoring session has been ended after agent dropped the call.

Silent Recording

Call recording is a feature that provides two ways of recording the conversations between the agent and the customer: the automatic call recording and the application invoked selective call recording. A line appearance configuration determines which mode is enabled. Administrators can configure no recording, automatically record all calls or per call based recording through application control. The recording configuration on a line appearance cannot be overridden by an application. TSP will report ‘Recording type’ info to app in devSpecificData of LineDevCaps structure. Whenever there is a change in ‘Recording Type’, TSP will send LINE_DEVSPECIFIC (SLDSMT_LINE_PROPERTY_CHANGED with indication of LPCT_RECORDING_TYPE) event to application.

If the automatic call recording is enabled, a recording session will be triggered whenever a call is received or initiated from the line appearance. When the application invoked call recording is enabled, application can start a recording session by using CCiscoLineDevSpecificStartCallRecording (SLDST_START_CALL_RECORDING) on the call after it becomes active. The selective recording can occur in the middle of the call, whereas the automatic recording always starts at the beginning of the call.The recorder is configured in CallManager as a SIP trunk device. Recorder DN can not be overridden by an application.

TSP will provide start recording request in lineDevSpecific to app for establishing a recording session. Application need to provide toneDirection as input to TSP in the start recording request. The result of the recording session is that the two media streams of the recorded call (agent-customer call) is being relayed from agent’s phone to the recorder. TSP will provide agent’s CCM Call Handle in the devSpecificData of LINECALLINFO.

TSP will inform application when recording starts on its call by sending LINE_CALLDEVSPECIFIC (SLDSMT_RECORDING_STARTED) event. TSP will provide recording call attribute info (deviceName, DN, Partition) in devspecific data of LINECALLINFO after recording starts.

The recording session will be terminated when the call is ended or if app sends stop recording request to TSP through lineDevSpecific – CciscoLineDevSpecificStopCallRecording (SLDST_STOP_CALL_RECORDING).TSP will inform agent by sending LINE_CALLDEVSPECIFIC (SLDSMT_RECORDING_ENDED) when recording is stopped by stop recording request.

Both recording and monitoring get supported only for IP phones/CTI supported phones that are running SIP and within one cluster. It can be invoked only on phones that support built in bridges. Also built in bridge should be turned on to monitor or record calls on a device. Monitoring party does not need to have a BIB configured. Recording and monitoring will not be supported for secure calls in this phase.

Call Attributes

Call Attributes can be found in DEVSPECIFIC porting of LINECALLINFO structure. The Call Attribute Info is presented in the format of an array since Silent Monitoring and Recoding could happen at the same time.

```
DWORD CallAttrtibuteInfoOffset;
    DWORD CallAttrtibuteInfoSize;
    DWORD CallAttrtibuteInfoElementCount;
DWORD CallAttrtibuteInfoElementFixedSize;
```

Offset pointing to array of the following structure:

```
typedef struct CallAttributeInfo
{
    DWORD CallAttributeType;
    DWORD PartyDNOffset;
    DWORD PartyDNSize;
    DWORD PartyPartitionOffset;
    DWORD PartyPartitionSize;
    DWORD DeviceNameOffset;
    DWORD DeviceNameSize;
}CallAttributeInfo;
```

enum CallAttributeType

```
{
    CallAttribute_Regular                    = 0,
    CallAttribute_SilentMonitorCall,
    CallAttribute_SilentMonitorCall_Target,
    CallAttribute_RecordedCall
} ;
```

Calling Party IP Address

The Calling Party IP Address feature provides the IP address of the calling party. The calling party device, which must be supported, must be an IP phone. The IP address is provided to applications in the devspecific data of LINECALLINFO. A value of zero (0) indicates that the information in not available.

The enhancement provides the IP address to the destination side of basic calls, consultation calls for transfer and conference, and basic redirect and forwarding. If the calling party changes, no support is provided.

Message Sequence

See Calling Party IP Address, page A-57.

Partition Support

The CiscoTSP support of this feature will provide Partition support information. Prior to release 5.1, CiscoTSP only reported partial information about the DNs to the applications in that it would report the numbers assigned but not the information about the partitions with which they were associated.

Thus, if a phone has two lines that are configured with same DNs – one in Partition P1 and the other in P2, a TSP application would cannot distinguish between these two and consequently open only one line of these two.

CiscoTSP provides the partition information about all DNs to the applications. Thus, the preceding limitation gets overcome and applications can distinguish between and open two lines on a device, which share the same DN but belong to different partitions.

TSP applications can query for LINEDEVCAPS where the partition information is stored in the devSpecific portion of the structure. Application will receive the Partition info for the CallerID, CalledID, ConnectedID, RedirectionID, and RedirectingID in a call. This gets provided as a part of DevSpecific Portion of the LINECALLINFO structure.

Also, the Partition info of the Call Park DN at which the call was parked will also be sent to the applications. The value of the partition info gets sent to applications in ASCII format.

Note Opening of a line from the application point of view remains unchanged.

Alternate Script

Certain IP phone types support an alternate language script other than the default script that corresponding to the phone configurable locale. For example, the Japanese phone locale associates two written scripts. Some phone types support only the default “Katakana” script, while other phones types support both the default “Katakana” script and the alternate “Kanji” script. Because applications can send text information to the phone for display purposes, they need to know what alternate script a phone supports – if any.

Secure RTP

The secure RTP (SRTP) feature allows Cisco TSP to report SRTP information to application as well as allow application to specify SRTP algorithm IDs during device registration. The SRTP information that Cisco TSP provides will include master key, master salt, algorithmID, isMKIPresent, and keyDerivation. To receive those key materials, administrator needs to configure TLS Enabled and SRTP Enabled flag in Cisco Unified Communications Manager Admin User windows and establish TLS link between TSP and CTIManager.

Besides, during device registration, application can provide SRTP algorithm IDs for CTI port and CTI Route Point in case of media termination by application. Application should use new Cisco extension for Line_devSpecific - CciscoLineDevSpecificUserSetSRTPAlgorithmID to set supported SRTP algorithm IDs after calling LineOpen with 0x80070000 version or higher negotiated, then followed by either CCiscoLineDevSpecificUserControlRTPStream or CciscoLineDevSpecificPortRegistrationPerCall to allow TSP to open device on CTI Manager.

When call arrives on an opened line, TSP will send LINE_CALLDEVSPECIFIC event to application with secure media indicator; then, application should query LINECALLINFO to get detail SRTP information if SRTP information is available. The SRTP information resides in the DevSpecific portion of the LINECALLINFO structure.

In case of mid-call monitoring, Cisco TSP will send LINE_CALLDEVSPECIFIC with secure media indicator, however there will be no SRTP info available for retrieval under this scenario. The event is only sent upon application request via PhoneDevSpecific with CPDST_REQUEST_RTP_SNAPSHOT_INFO message type.

To support SRTP that is using static registration, a generic mechanism for delayed device/line now exists. The following ones apply:

- Extension version bit SELSIUSTSP_LINE_EXT_VER_FOR_DELAYED_OPEN = 0x40000000
- CiscoLineDevSpecificType - SLDST_SEND_LINE_OPEN
- CCiscoLineDevSpecific - CciscoLineDevSpecificSendLineOpen

If application negotiates with 0x00070000 in lineOpen against CTI port, TSP will do LineOpen/DeviceOpen immediately. If application negotiates with 0x40070000 in LineOpen against CTI port, TSP will delay the LineOpen/DeviceOpen. Application can specify SRTP algorithm ID by using CciscoLineDevSpecificUserSetSRTPAlgorithmID (SLDST_USER_SET_SRTP_ALGORITHM_ID). However, to trigger actual device/line open in TSP, application needs to send CciscoLineDevSpecificSendLineOpen(SLDST_SEND_LINE_OPEN)

If application negotiates with 0x80070000 in LineOpen against CTI port/RP, TSP will delay the LineOpen/DeviceOpen until application specifies media information in CCiscoLineDevSpecific; however, application can use CciscoLineDevSpecificUserSetSRTPAlgorithmID (SLDST_USER_SET_SRTP_ALGORITHM_ID) to specify SRTP algorithm ID before specifying the media information.

If application negotiates with 0x40070000 in LineOpen against RP, TSP should fail CciscoLineDevSpecificUserSetSRTPAlgorithmID (SLDST_USER_SET_SRTP_ALGORITHM_ID) request because RP should have media terminated by application.

Currently, the SELSIUSTSP_LINE_EXT_VER_FOR_DELAYED_OPEN bit only gets used on CTI port when TSP Wave Driver is used to terminate media. Under conference scenario, the SRTP information gets stored in conference parent call for each party. An application that negotiates with correct version and interested in SRTP info in conference scenario should retrieve SRTP information from CONNECTED call of particular conference party.

Backwards Compatibility

CCiscoLineDevSpecific extension

CciscoLineDevSpecificUserSetSRTPAlgorithmID is defined.

CCiscoLineDevSpecific extension CciscoLineDevSpecificSendLineOpen is defined. An extra LINE_CALLDEVSPECIFIC event gets sent if negotiated version of application supports this feature while keeping existing LINE_CALLDEVSPECIFIC for reporting existing RTP parameters.

Wave driver (media terminating endpoint) uses the strip_policy to create a crypto context. It should match the encrypt and decrypt packets sent/received by IPPhones/CTIPorts. Phone uses one hardcoded srtp_policy for all phone types including phones that are using SIP.

```
policy->cipher_type      = AES_128_ICM;
policy->cipher_key_len  = 30;
policy->auth_type        = HMAC_SHA1;
policy->auth_key_len    = 20;
policy->auth_tag_len     = 4;
policy->sec_serv         = sec_serv_conf_and_auth;
```

Note Applications should not store key material and must use proper security method to ensure confidentially of the key material. Application should clear out memory after key info is processed. Be aware that applications are subject to export control when they provide mechanism to encrypt the data (SRTP). Applications should get export clearance for that.

SuperProvider

SuperProvider functionality allows a TSP application to control any CTI-controllable device in the system (IP Phones, CTI Ports, CTI Route Points and so on). Normally, a TSP application must have an associated list of devices that it can control. It cannot control any devices that are outside this list; however, certain applications would want to control a large number (possibly all) the CTI controllable devices in the system.

SuperProvider functionality enables the administrator to configure a CTI application as a SuperProvider. This will mean that particular application can control absolutely any CTI controllable device in the system.

The SuperProvider functionality never gets exposed to TSP apps; that is, TSP application needed to have the device in the controllable list. In release 5.1 and later, TSP apps can control any CTI-controllable device in the Unified CM system.

The SuperProvider apps need to explicitly ‘Acquire’ the device before opening them. TSP exposes new interfaces to the apps to explicitly Acquire/Deacquire any device in the system. The device info will get fetched for the explicitly acquired device by using the SingleGetInfoFetch API exposed via QBE by CTI. Later, TSP will fetch the line info for this device by using the LineInfoFetch API. The lines of this device will get reported to the app by using the LINE_CREATE/PHONE_CREATE events.

The apps can explicitly ‘De-Acquire’ the device. After the device is successfully de-acquired, TSP will fire LINE_REMOVE/PHONE_REMOVE events to the apps.

TSP also supports Change Notification of “Super-Provider” flag. If the administrator has configured a User as a “Super-Provider” in the Unified CM Administration, the status of this changes and the user no longer represents a Super-Provider, then CTI will inform TSP in ProviderUserChangedEvent. If any device had been explicitly acquired and opened in super-provider mode, TSP will fire PHONE_REMOVE/LINE_REMOVE to the app and indicates that the device/line is no longer available for the app to use.

In release 5.1 and later, TSP supports change notification of CallParkDN Monitoring as well. If the user has been configured to allow monitoring of CallParkDN in the Unified CM Administration, the status of this flag is disabled. Then TSP will fire LINE_REMOVE for the CallParkDNs.

If the CallParkDN Monitoring is disabled, the status changes to enabled, TSP fetches all the CallParkDNs from CTI and fire LINE_CREATE for each of the CallParkDNs.

Refer and Replaces for Phones that are Running SIP

As part of CTI support for phones that are running SIP, TSP will support new SIP features Refer and Replaces. Refer, Refer with Replaces, Invite with Replaces represent different mechanisms to initiate different call control features. For example, Refer with Replaces in SIP terms can be translated to Transfer operation in Unified CM. Invite with Replaces can be used for Pickup call feature across SIP trunks. TSP will support event handling corresponding to the features that are initiated by Unified CM phones that are running SIP / third party phones that are running SIP. TSP will not support Refer / Replaces request initiation from the API. Because TAPI does not have Refer/Replaces feature related reason codes defined, the standard TAPI reason will be LINECALLREASON_UNKNOWN. TSP will provide new call reason to user as part of LINE_CALLINFO::dwDevSpecificData if the application negotiated extension version greater or equal to 0x00070000.

For In-dialog refer, TSP places Referrer in LINECALLSTATE_UNKNOWN | CLDSMT_CALL_WAITING_STATE call state with extended call reason as CtiCallReasonRefer. This helps present the Referrer’s call leg such that applications cannot call any other call functions on this call. Any request on this call when it is in LINECALLSTATE_UNKNOWN | CLDSMT_CALL_WAITING_STATE will return an error as LINEERR_INVALCALLSTATE.

The Referrer must clear this call after the Refer request is initiated. If Referrer does not drop the call, Refer feature will clear this call if the refer is successful. LINECALLSTATE_IDLE with extended reason as CtiCallReasonRefer will get reported.

If Referrer does not drop the call and if Refer request fails (For example, Refer to target is busy), refer feature will restore the call between Referrer and Referee.

With Unified CM Phones that are running SIP, Unified CM makes all the existing call features transparent such that applications will get the existing events when the phone invokes a SIP feature whose behavior matches with the existing feature of Unified CM. For example, when Refer with Replaces is called by a phone that runs SIP (with both primary and secondary/consult call legs on same SIP line) within Unified CM cluster, all the events will get reported the same as Transfer feature.

SIP URL Address

As part of CTI support for phones that are using SIP, TSP will expose SIP URL that is received in Device/Call event that is received from CTIManager. The SIP URL will get presented for each corresponding party in extended call info structure of LINE_CALLINFO::dwDevSpecificData.

When a SIP trunk is involved in a call, the DN may not get presented in party information. Application then needs to consider SIP URL information under this call scenario for information.

TSP will provide SIP URL information to user as part of LINE_CALLINFO::dwDevSpecificData if the application negotiated extension version greater than or equal to 0x00070000.

CTI phones that are running SIP support the following features or functions:

- Call Initiate
- Call Answer
- Call Close/Disconnect
- Consult Transfer
- Direct Transfer
- Call Join
- Conference
- Hold/unhold
- Line Dial
- Redirect
- lineDevSpecific (SLDST_MSG_WAITING)
- lineDevSpecific (SLDST_MSG_WAITING_DIRN)

3XX

Cisco TSP maps the CTI reason code for 3XX to REDIRECT. When a call arrives on a monitored line due to 3XX feature, the call reason for the incoming call will get REDIRECT in this case. No interface change for TSP 3XX support.

Backward Compatibility

This feature is backward compatible.

Secure TLS Support

Establishing secure connection to CTIManager involves application user to configure more information through Cisco TSP UI. This information will help TSP to create its own client certificate. This certificate is used to create a mutually authenticated secure channel between TSP and CTIManager.

TSP UI adds a new tab called Security and the options that are available on this tab follows:

- Check box for Secure Connection to CTIManager: If checked, TSP will connect over TLS CTIQBE port (2749); otherwise, TSP will connect over CTIQBE port (2748).
- Default setting specifies non secure connection and the setting will remain unchecked.

Ensure that the security flag for the TSP user is enabled through Cisco Unified Communications Manager Administration as well. CTIManager will perform a verification check whether a user who is connecting on TLS is allowed to have secure access. CTIManager will allow only security enabled users to connect to TLS port 2749 and only non secure users to connect to CTIQBE port 2748.

The user flag to enable security depends on the cluster security mode. If cluster security mode is set to secure, user security settings will have a meaning; otherwise, the connection has to be non secure. If secure connection to CTIManager is checked, the following settings will get enabled for editing.

- CAPF Server: CAPF server IP address from which to fetch the client certificate.
- CAPF Port: (Default 3804) – CAPF Server Port to connect to for Certificate download.
- Authorization Code (AuthCode): Required for Client authentication with CAPF Server and Private Key storage on client machine.
- Instance ID(IID): Each secure connection to CTIManager must have its own certificate for authentication. With the restriction of having a distinct certificate per connection, CAPF Server needs to verify that the user with appropriate AuthCode and IID is requesting the certificate. CAPF server will use AuthCode and IID to verify the user identity. After CAPF server provides a certificate, it clears the AuthCode to make sure only one instance of an app requests a certificate based on a single AuthCode. CCM admin will allow user configuration to provide multiple IID and AuthCode.
- TFTP Server: TFTP server IP address to fetch the CTL file. CTL, which file is required to verify the server certificate, gets sent while mutually authenticating the TLS connection.
- Check box to Fetch Certificate: This setting is not stored anywhere, instead only gets used to update the Client certificate when it is checked and will get cleared automatically.
- Number of Retries for Certificate Fetch: This indicates the number of retries TSP will perform to connect to CAPF Server for certificate download in case an error. (Default - 0) (Range – 0 to 3)
- Retry Interval for Certificate Fetch: This will be used when the retry is configured. It indicates the (secs) for which TSP will wait during retries. (Default – 0) (Range – 0 to 15)

Because user is not expected to update the client certificate every time it changes, TSP UI will pop up a message when this box is checked by user that says “This will trigger a certificate update. Please make sure that you really want to update the TSP certificate now.” This will ensure that if user selects this check box in an error. TSP will fail to establish a secure connection to CTIManager if a valid certificate cannot be obtained. Each secure connection to CTIManager needs to have a unique certificate for authentication.

If an application tries to create more than one Provider simultaneously with the same certificate or when a session with the same certificate already exists/is open, CTI Manager disconnects both providers. TSP will try reconnecting to CTIManager to bring the provider in service. However, if both providers continuously try to connect by using the same duplicated certificate, both providers will be closed after a certain number of retries, and the certificate will be marked as compromised by CTIManager on Unified CM server. The number of retries after which the certificate should be marked as compromised is configurable from the CTIManager Service Parameter “CTI InstanceId Retry Count.” CTI manager rejects further attempt to open provider with the certificate that is compromised. In this case, the CAPF profile of the compromised certificate should be deleted and a new CAPF Profile must be created for the user. The new CAPF profile for the user should use new instance ID. Otherwise, the old certificate, which was compromised previously, can be used again.

A new error code, TSPERR_INIT_CERTIFICATE_COMPROMISED, with value as 0x00000011 where TSPERR_UNKNOWN is 0x00000010 now exists. Application should not have checks like "if (err < TSPERR_UNKNOWN))" because error codes exists that have a value greater than that.

When TLS is used, the initial handshake will be slow as expected due to heavy use of public key cryptography. After the initial handshake is complete and the session is established, the overhead gets significantly reduced. The following profiling result applies on ProviderOpen for both secure and non-secure CTI connection.

Controlled Devices	Type of CTI Connection	Duration on ProviderOpen	Duration on OpenAllLines	Comments
0	Non-Secure	1 sec 382 ms	N/A	
	Secure	4 sec 987 ms	N/A	With certificate retrieval.
	Secure	3 sec 736 ms	N/A	
100	Non-secure	1 sec 672 ms	3 sec 164ms	
	Secure	5 sec 758 ms	3 sec 445ms	
2500	Non-Secure	29 sec 513 ms	3 min 26 sec 728 ms	
	Secure	34 sec 219 ms	3 min 26 sec 928 ms	

Monitoring Call Park Directory Numbers

Cisco TSP supports monitoring calls on lines that represent Cisco Unified Communications Manager Administration Call Park Directory Numbers (Call Park DNs). Cisco TSP uses a device-specific extension in the LINEDEVCAPS structure that enables TAPI applications to differentiate Call Park DN lines from other lines. If an application opens a Call Park DN line, all calls that are parked to the Call Park DN are reported to the application. The application cannot perform any call-control functions on any of the calls at a Call Park DN.

In order to open Call Park DN lines, the Monitor Call Park DNs check box in the Cisco Unified Communications Manager Administration for the TSP user must be checked. Otherwise, the application will not see any of the Call Park DN lines upon initialization.

Super Provider Support

The Super Provider functionality allows a TSP application to control any CTI controllable device in the system (IP Phones, CTI Ports, CTI Route Points etc). The TSP application has to have an associated list of devices that it can control. It cannot control any devices that are outside of this list. However, certain applications would want to control a large number (possibly all) of the CTI controllable devices in the system. Super Provider enables the administrator to configure a CTI application as a "Super-Provider." This will mean that particular application can control absolutely any CTI controllable device in the system.

Previously, the Super Provider functionality was never exposed to TSP apps, that is the TSP application needed to have the device in the controllable list. In this release, TSP apps can control any CTI controllable device in the CallManager system. The Super-Provider apps need to explicitly "Acquire" the device before opening them.

TSP exposes new interfaces to the apps to explicitly Acquire/Deacquire any device in the system. The device info will be fetched for the explicitly acquired device using the SingleGetInfoFetch API exposed via QBE by CTI. Later, TSP will fetch the line info for this device using the LineInfoFetch API. The lines of this device will be reported to the app using the LINE_CREATE/PHONE_CREATE events.

The apps can explicitly 'De-Acquire' the device. After the device is successfully de-acquired, TSP will fire LINE_REMOVE/PHONE_REMOVE events to the apps.

TSP also supports Change Notification of "Super-Provider" flag. If the administrator has configured a User as a "Super-Provider" in the admin pages, then the status of this is changed and the user is no more a Super-Provider, then CTI will inform the same to TSP in ProviderUserChangedEvent.

If any device had been explicitly acquired and opened in super-provider mode, then TSP will fire PHONE_REMOVE/LINE_REMOVE to the app indicating that the device/line is no more available for the app to use.

In this release, TSP supports change notification of CallParkDN Monitoring as well. Say, the user has been configured to allow monitoring of CallParkDN in the admin pages, now the status of this flag is disabled. Then TSP will fire LINE_REMOVE for the CallParkDNs.

Say, initially the CallParkDN Monitoring is disabled, now the status is changed to enabled, then TSP will fetch all the CallParkDNs from CTI and fire LINE_CREATE for each of the CallParkDNs.

Unicode Support

Cisco TSP supports unicode character sets. TSP will send unicode party names to the application in all call events. The party name needs to be configured in Cisco Unified Communications Manager Administration. This support is limited to only party names. The locale information also gets sent to the application. The UCS-2 encoding for unicode gets used.

The party names will exist in the DevSpecific portion of the LINECALLINFO structure. In SIP call scenarios, where a call comes back into Unified CM from SIP proxy over a SIP trunk, only ASCII name will get passed because SIP has no way of populating both ASCII and unicode. As the result, the Connected and Redirection Unicode Name will get reported as empty to application.

Line–Side Phones That Runs SIP

TSP supports controlling and monitoring of TNP-based phones that are running SIP. Existing phones (7960 and 7940) that are running SIP cannot be controlled or monitored by the TSP and should not get included in the control list. Though the general behavior of a phone that is running is similar to a phone that is running SCCP not all TSP features get supported for phones that are running SIP.

CCiscoPhoneDevSpecificDataPassThrough functionality does not support on phones that are running SIP configured with UDP transport due to UDP limitations. Phones that are running SIP must be configured to use TCP (default) if the CCiscoPhoneDevSpecificDataPassThrough functionality is needed.

TSP application registration state for TNP phones that are running SIP with UDP as transport may not remain consistent to the registration state of the phone. TNP phone that are running SIP with UDP as transport may appear to be registered when application reports the devices as out of service. This may happen when CTIManager determines that Unified CM is down and puts the device as out of service, but, because of the inherent delay in UDP to determine the lost connectivity, phone may appear to be in service.

The way applications open devices and lines on phones that are running SIP remains the same as that of phone that is running SCCP. It is the phone that controls when and how long to play reorder tone. When a SIP phone gets a request to play reorder tone, the phone that is running SIP releases the resources from Unified CM and plays reorder tone. The call appears to be IDLE to a TSP application even though reorder tone is being played on the phone. Applications can still receive and initiate calls from the phone even when reorder tone plays on the phone. Because resources have been released on Unified CM, this call does not count against the busy trigger and maximum number of call counters.

When consult call scenario is invoked on the SIP, the order of new call event (for consult call) and on hold call state change event (for original call).

Redirect and Blind Transfer

The Cisco Unified TSP supports several different methods of Redirect and Blind Transfer. This section outlines the different methods as well as the differences between methods.

lineRedirect

This standard TAPI lineRedirect function redirects calls to a specified destination. The Calling Search Space and Original Called Party that Cisco Unified TSP uses for this function follows:

- Calling Search Space (CSS) — Uses CSS of the CallingParty (the party being redirected) for all cases unless the call is in a conference or a member of a two-party conference where it uses the CSS of the RedirectingParty (the party that is doing the redirect).
- Original Called Party — Not changed.

lineDevSpecific – Redirect Reset Original Called ID

This function redirects calls to a specified destination while resetting the Original Called Party to the party that is redirecting the call. The Calling Search Space and Original Called Party that Cisco Unified TSP uses for this function follow:

- Calling Search Space (CSS) — Uses CSS of the CallingParty (the party being redirected).
- Original Called Party — Set to the RedirectingParty (the party that is redirecting the call).

lineDevSpecific – Redirect Set Original Called ID

This function redirects calls to a specified destination while allowing the application to set the Original Called Party to any value. The Calling Search Space and Original Called Party that Cisco Unified TSP uses for this function follow:

- Calling Search Space (CSS) — Uses CSS of the CallingParty (the party being redirected).
- Original Called Party — Set to the preferredOriginalCalledID that the lineDevSpecific function specifies.

You can use this request to implement the Transfer to Voice Mail feature (TxToVM). Using this feature, applications can transfer the call on a line directly to the voice mailbox on another line. You can achieve TxToVM by specifying the following fields in the above request: voice mail pilot as the destination DN and the DN of the line to whose voice mail box the call is to be transferred as the preferredOriginalCalledID.

lineDevSpecific – Redirect FAC CMC

This function redirects calls to a specified destination that requires either a FAC, CMC, or both. The Calling Search Space and Original Called Party that Cisco Unified TSP uses for this function follow:

- Calling Search Space (CSS) — Uses CSS of the CallingParty (the party being redirected).
- Original Called Party — Not changed.

lineBlindTransfer

Use the standard TAPI lineBlindTransfer function to blind transfer calls to a specified destination. The Calling Search Space and Original Called Party that Cisco Unified TSP uses for this function follow:

- Calling Search Space (CSS) — Uses CSS of the TransferringParty (the party that is transferring the call).
- Original Called Party — Set to the TransferringParty (the party that is transferring the call).

lineDevSpecific - BlindTransfer FAC CMC

This function blind transfers calls to a specified destination that requires a FAC, CMC, or both. The Calling Search Space and Original Called Party that Cisco Unified TSP uses for this function follow:

- Calling Search Space (CSS) — Uses CSS of the TransferringParty (the party that is transferring the call).
- Original Called Party — Set to the TransferringParty (the party that is transferring the call).

Direct Transfer

In Cisco Unified Communications Manager, the “Direct Transfer” softkey lets users transfer the other end of one established call to the other end of another established call, while dropping the feature initiator from those two calls. Here, an established call refers to a call that is either in the on hold state or in the connected state. The “Direct Transfer” feature does not initiate a consultation call and does not put the active call on hold.

A TAPI application can invoke the “Direct Transfer” feature by using the TAPI lineCompleteTransfer() function on two calls that are already in the established state. This also means that the two calls do not have to be set up initially by using the lineSetupTransfer() function.

Join

In Cisco Unified Communications Manager, the “Join” softkey joins all the parties of established calls (at least two) into one conference call. The “Join” feature does not initiate a consultation call and does not put the active call on hold. It also can include more than two calls, which results in a call with more than three parties.

Cisco Unified TSP exposes the “Join” feature as a new device-specific function that is known as lineDevSpecific() – Join. Applications can apply this function to two or more calls that are already in the established state. This also means that the two calls do not need to be set up initially by using the lineSetupConference() or linePrepareAddToConference() functions.

Cisco Unified TSP also supports the lineCompleteTransfer() function with dwTransferMode=Conference. This function allows a TAPI application to join all the parties of two, and only two, established calls into one conference call.

Cisco Unified TSP also supports the lineAddToConference() function to join a call to an existing conference call that is in the ONHOLD state.

A feature interaction issue involves Join, Shared Lines, and the Maximum Number of Calls. The issue occurs when you have two shared lines and the maximum number of calls on one line is less than the maximum number of calls on the other line.

For example, in a scenario where one shared line, A, has a maximum number of calls set to 5 and another shared line, A', has a maximum number of calls set to 2, the scenario involves the following steps:

A calls B. B answers. A puts the call on hold.

C calls A. A answers. C puts the call on hold.

At this point, line A has two calls in the ONHOLD state, and line A' has two calls in the CONNECTED_INACTIVE state.

D calls A. A answers.

At this point, the system presents the call to A, but not to A'. This happens because the maximum calls for A' specifies 2.

A performs a Join operation either through the phone or by using the lineDevSpecific – Join API to join all the parties in the conference. It uses the call between A and D as the primary call of the Join operation.

Because the call between A and D was used as the primary call of the Join, the system does not present the ensuing conference call to A'. Both calls on A' will go to the IDLE state. As the end result, A' will not see the conference call that exists on A.

Set the Original Called Party upon Redirect

Two extensions enable setting the original called party upon redirect as follows:

- CCiscoLineDevSpecificRedirectResetOrigCalled
- CCiscoLineDevSpecificRedirectSetOrigCalled

See lineDevSpecific, page 5-10 for more information.

Cisco Unified TSP Auto Update

Cisco Unified TSP supports auto update functionality, so the latest plug-in can be downloaded and installed on a client machine. Be aware that the new plug-in will be QBE compatible with the connected CTIManager. When the Cisco Unified Communications Manager is upgraded to a newer version, and Cisco Unified TSP auto update functionality is enabled, the user will receive the latest compatible Cisco Unified TSP, which will work with the upgraded Cisco Unified Communications Manager. This ensures that the applications work as expected with the new release (provided the new Unified CM interface is backward compatible with the TAPI interface). The locally installed Cisco Unified TSP on the client machine allows applications to set the auto update options as part of the Cisco Unified TSP configuration. The user can opt for updating Cisco Unified TSP in the following different ways:

- Update Cisco Unified TSP whenever a different version (higher version than the existing version) is available on the Cisco Unified Communications Manager server.

- Update Cisco Unified TSP whenever a QBE protocol version mismatch exists between the existing Cisco Unified TSP and the Cisco Unified Communications Manager version.
- Do not update Cisco Unified TSP by using Auto Update functionality.

Multiple Calls per Line Appearance

The following topics describe the conditions of Line Appearance.

Maximum Number of Calls

The maximum number of calls per Line Appearance remains database configurable, which means that the Cisco TSP supports more than two calls per line on Multiple Call Display (MCD) devices. An MCD device comprises a device that can display more than two call instances per DN at any given time. For non-MCD devices, the Cisco TSP supports a maximum of two calls per line. The absolute maximum number of calls per line appearance equals 200.

Busy Trigger

In Cisco Unified CM, a setting, busy trigger, indicates the limit on the number of calls per line appearance before the Cisco Unified CM will reject an incoming call. Be aware that the busy trigger setting is database configurable, per line appearance, per cluster. The busy trigger setting replaces the old call waiting flag per DN. You cannot modify the busy trigger setting using the CiscoTSP.

Call Forward No Answer Timer

Be aware that the Call Forward No Answer timer is database configurable, per DN, per cluster. You cannot configure this timer using the CiscoTSP.

Shared Line Appearance

Cisco Unified TSP supports opening two different lines that each share the same DN. Each of these lines represents a Shared Line Appearance.

The Cisco Unified Communications Manager allows multiple active calls to exist concurrently on each of the different devices that share the same line appearance. The system still limits each device to, at most, one active call and multiple hold or incoming calls at any given time. Applications that use the Cisco Unified TSP to monitor and control shared line appearances can support this functionality.

If a call is active on a line that is a shared line appearance with another line, the call appears on the other line with the dwCallState=CONNECTED and the dwCallStateMode=INACTIVE. Even though the call party information may not display on the actual IP phone for the call at the other line, Cisco Unified TSP still reports the call party information on the call at the other line. This gives the application the ability to decide whether to block this information. Also, the system does not allow call control functions on a call that is in the CONNECTED INACTIVE call state.

Cisco Unified TSP does not support shared lines on CTI Route Point devices.

In the scenario where a line is calling a DN that contains multiple shared lines, the dwCalledIDName in the LINECALLINFO structure for the line with the outbound call may be empty or set randomly to the name of one of the shared DNs. The reason for this should be obvious as Cisco Unified TSP and the Cisco Unified Communications Manager cannot resolve which of the shared DN's you are calling until the call is answered.

Select Calls

The "Select" softkey on IP phones lets a user select call instances to perform feature activation, such as transfer or conference, on those calls. The action of selecting a call on a line locks that call, so it cannot be selected by any of the shared line appearances. Pressing the "Select" key on a selected call will deselect the call.

Cisco Unified TSP does not support the "Select" function to select calls because all transfer and conference functions contain parameters that indicate on which calls the operation should be invoked.

Cisco Unified TSP supports the events that a user who selects a call on an application-monitored line causes. When a call on a line is selected, all other lines that share the same line appearance will see the call state change to dwCallState=CONNECTED and dwCallStateMode=INACTIVE.

Forced Authorization Code and Client Matter Code

Cisco Unified TSP supports and interacts with two Cisco Unified Communications Manager features: Forced Authorization Code (FAC) and Client Matter Code (CMC). The FAC feature lets the System Administrator require users to enter an authorization code to reach certain dialed numbers. The CMC feature lets the System Administrator require users to enter a client matter code to reach certain dialed numbers.

The system alerts a user of a phone that a FAC or CMC must be entered by sending a "ZipZip" tone to the phone that the phone in turn plays to the user. Cisco Unified TSP will send a new LINE_DEVSPECIFIC event to the application whenever the application should play a "ZipZip" tone. Applications can use this event to indicate when a FAC or CMC is required. For an application to start receiving the new LINE_DEVSPECIFIC event, it must perform the following steps:

1. lineOpen with dwExtVersion set to 0x00050000 or higher
2. lineDevSpecific – Set Status Messages to turn on the Call Tone Changed device specific events

The application can enter the FAC or CMC code with the lineDial() API. Applications can enter the code in its entirety or one digit at a time. An application may also enter the FAC and CMC code in the same string as long as they are separated by a "#" character and also ended with a "#" character. The optional "#" character at the end only serves to indicate dialing is complete.

If an application does a lineRedirect() or a lineBlindTransfer() to a destination that requires a FAC or CMC, Cisco Unified TSP returns an error. The error that Cisco Unified TSP returns indicates whether a FAC, a CMC, or both are required. Cisco Unified TSP supports two new lineDevSpecific() functions, one for Redirect and one for BlindTransfer, that allows an application to enter a FAC or CMC, or both, when a call gets redirected or blind transferred.

CTI Port Third-Party Monitoring Port

Opening a CTI port device in first-party mode means that either the application is terminating the media itself at the CTI port or that the application is using the Cisco Wave Drivers to terminate the media at the CTI port. This also comprises registering the CTI port device.

Opening a CTI port in third-party mode means that the application is interested in just opening the CTI port device, but it does not want to handle the media termination at the CTI port device. An example of this would be a case where an application would want to open a CTI port in third-party mode because it is interested in monitoring a CTI port device that has already been opened/registered by another application in first party mode. Opening a CTI Port in third-party mode does not prohibit the application from performing call control operations on the line or on the calls of that line.

Cisco Unified TSP allows TAPI applications to open a CTI port device in third-party mode via the lineDevSpecific API, if the application has negotiated at least extension version 6.0(1) and set the high order bit, so the extension version is set to at least 0x80050000.

The TAPI architecture lets two different TAPI applications that are running on the same PC use the same Cisco Unified TSP. In this situation, if both applications want to open the CTI port, problems could occur. Therefore, the first application to open the CTI port will control the mode in which the second application is allowed to open the CTI port. In other words, all applications that are running on the same PC, using the same Cisco Unified TSP, must open CTI ports in the same mode. If a second application tries to open the CTI port in a different mode, the lineDevSpecific() request fails.

Translation Pattern

Warning **TSP does not support the translation pattern because it may cause a dangling call in a conference scenario. The application needs to clear the call to remove this dangling call or simply close and reopen the line.**

Forwarding

Cisco Unified TSP now provides added support for the lineForward() request to set and clear ForwardAll information on a line. This will allow TAPI applications to set the Call Forward All setting for a particular line device. Activating this feature will allow users to set the call forwarding Unconditionally to a forward destination.

Cisco Unified TSP sends LINE_ADDRESSSTATE messages when lineForward() requests successfully complete. These events also get sent when call forward indications are obtained from the CTI, indicating that a change in forward status has been received from a third party, such as Cisco Unified Communications Manager Administration or another application setting call forward all.

Extension Mobility

Extension Mobility, a Cisco Unified Communications Manager feature, allows a user to log in and log out of a phone. Cisco Extension Mobility loads a user Device Profile (including line, speed dial numbers, and so on) onto the phone when the user logs in.

Cisco Unified TSP recognizes a user who is logged into a device as the Cisco Unified TSP User.

Using Cisco Unified Communications Manager Administration, you can associate a list of controlled devices with a user.

When the Cisco Unified TSP user logs into the device, the system places the lines that are listed in the user Cisco Extension Mobility profile on the phone device and removes lines that were previously on the phone. If the device is not in the controlled device list for the Cisco Unified TSP User, the application receives a PHONE_CREATE or LINE_CREATE message. If the device is in the controlled list, the application receives a LINE_CREATE message for the added line and a LINE_REMOVE message for the removed line.

When the user logs out, the original lines get restored. For a non-controlled device, the application perceives a PHONE_REMOVE or LINE_REMOVE message. For a controlled device, it perceives a LINE_CREATE message for an added line and a LINE_REMOVE message for a removed line.

Directory Change Notification

The Cisco Unified TSP sends notification events when a device has been added to or removed from the user-controlled device list in the directory. Cisco Unified TSP sends events when the user is deleted from Cisco Unified Communications Manager Administration.

Cisco Unified TSP sends a LINE_CREATE or PHONE_CREATE message when a device is added to a users control list.

It sends a LINE_REMOVE or PHONE_REMOVE message when a device is removed from the user controlled list or the device is removed from database.

When the system administrator deletes the current user, Cisco Unified TSP generates a LINE_CLOSE and PHONE_CLOSE message for each open line and open phone. After it does this, it sends a LINE_REMOVE and PHONE_REMOVE message for all lines and phones.

Note Cisco Unified TSP generates PHONE_REMOVE / PHONE_CREATE messages only if the application called the phoneInitialize function earlier.

The system generates a change notification if the device is added to or removed from the user by using Cisco Unified Communications Manager Administration or the Bulk Administration Tool (BAT).

If you program against the LDAP directory, change notification does not generate.

Privacy Release

The Cisco Unified Communications Manager Privacy Release feature allows the user to dynamically alter the privacy setting. The privacy setting affects all existing and future calls on the device.

Cisco Unified TSP does not support the Privacy Release feature.

Barge and cBarge

Cisco Unified Communications Manager supports the Barge and cBarge features. The Barge feature uses the built-in conference bridge. The cBarge feature uses the shared conference resource.

Cisco Unified TSP supports the events that are caused by the invocation of the Barge and cBarge features. It does not support invoking either Barge or cBarge through an API of Cisco Unified TSP.

XSI Object Pass Through

XSI-enabled IP phones allow applications to directly communicate with the phone and access XSI features, such as manipulate display, get user input, play tone, and so on. To allow TAPI applications access to the XSI capabilities without having to set up and maintain an independent connection directly to the phone, TAPI provides the ability to send the device data through the CTI interface. The system exposes this feature as a Cisco Unified TSP device-specific extension.

The system only supports the PhoneDevSpecificDataPassthrough request for IP phone devices.

Silent Install Support

The Cisco TSP installer supports silent install, silent upgrade, and silent reinstall from the command prompt. Users do not see any dialog boxes during the silent installation.

CHAPTER 4

Cisco Unified TAPI Installation

This chapter describes how to install and configure the Cisco Unified Telephony Application Programming Interface (TAPI) client software for Cisco Unified Communications Manager. It contains the following sections:

Note The upgraded TAPI client software does not work with previous releases of Cisco Unified Communications Manager.

Installing the Cisco Unified TSP

Install the Cisco Unified TSP software either directly from the Cisco Unified Communications Manager CD-ROM or from Cisco Unified Communications Manager Administration. For information on installing plug-ins from the Cisco Unified Communications Manager, see the *Cisco Unified Communications Manager Administration Guide*. The installation wizard varies depending on whether you have a previous version of Cisco Unified TSP installed.

Note If you are installing multiple TSPs, multiple copies of CiscoTSPXXX.tsp and CiscoTUISPXXX.dll files will exist in the same Windows system directory.

To install the Cisco Unified TSP from the Cisco Unified Communications Manager Administration CD-ROM, perform the following steps:

Procedure

Step 1 Insert the Cisco Unified Communications Manager CD-ROM.

Step 2 Double-click **My Computer**.

Step 3 Double-click the CD-ROM drive.

Step 4 Double-click the **Installs** folder.

Step 5 Double-click **Cisco TSP.exe**.

Step 6 Follow the online instructions.

Next Step

Install the Cisco Wave Driver if you plan to use first-party call control. Perform this step even if you are performing your own media termination. For more information, see the "Installing the Wave Driver" section.

Silent Installation

You can silently install, upgrade, or reinstall Cisco TSP. Use the following commands on the Windows command line:

Installation

```
CiscoTSP.exe /s /v"/qn"
```

Upgrade

```
CiscoTSP.exe /s /v"/qn"
```

Reinstallation

```
CiscoTSP.exe /s /v"/qn REINSTALL=\"ALL\" REBOOT=\"ReallySuppress\""
```

Activating the Cisco Unified TSP

You can install up to 10 TSPs on a computer. Use the following procedure to activate each of these TSPs. When you install a Cisco Unified TSP, you add it to the set of active TAPI service providers. The TSP displays as CiscoTSPXXX, where X ranges between 001 and 010. If a TSP has been removed or if some problem has occurred, you can manually add it to this set.

To manually add the Cisco Unified TSP to the list of telephony drivers, perform the following steps.

Procedure for Windows 2000 and Windows XP

Step 1 Open the Control Panel.

Step 2 Double-click **Phone and Modem Options**.

Step 3 On the Phone and Modem Options dialog box, click the **Advanced** tab.

Note If the Cisco Unified TSP is either not there or you removed it previously and want to add it now, you can do so from this window.

Step 4 Click **Add**.

Step 5 On the Add Provider dialog box, choose the appropriate TSP. Labels identify the TSPs in the Telephony providers window as CiscoTSPXXX, where XXX ranges between 001 and 010.

Step 6 Click **Add**.

The TSP that you chose displays in the provider list in the Phone and Modem Options window.

Step 7 Configure the Cisco Unified TSP as described in "Configuring the Cisco Unified TSP" or click **Close** to complete the setup.

Procedure for Windows NT, Windows 98, and Windows 95

Step 1 Open the Control Panel.

Step 2 Double-click **Telephony**.

Step 3 Click the **Telephony Drivers** tab.

Note If the Cisco Unified TSP is either not there or you removed it previously and want to add it now, you can do so from this window.

Step 4 Click **Add**.

Step 5 On the Add Provider dialog box, choose the appropriate TSP. Labels identify the TSPs in the Telephony providers window as CiscoTSPXXX, where XXX ranges between 001 and 010.

Step 6 Click **Add**.

The Provider list in the Telephony Drivers window now includes the CiscoTSPXXX range 001 - 010.

Step 7 Configure Cisco Unified TSP as described in "Configuring the Cisco Unified TSP" or click **Close** to complete the setup.

Configuring the Cisco Unified TSP

You configure the Cisco Unified TSP by setting parameters in the Cisco IP-PBX Service Provider configuration window. Perform the following steps to configure Cisco Unified TSP.

Procedure for Windows 2000 and Windows XP

Step 1 Open the Control Panel.

Step 2 Double-click **Phone and Modem Options**.

Step 3 Choose the Cisco Unified TSP that you want to configure.

Step 4 Click **Configure**.

The system displays the Cisco IP PBX Service Provider dialog box.

Step 5 Enter the appropriate settings as described in the "Cisco Unified TSP Configuration Settings" section.

Step 6 To save changes, click **OK**.

Note After the TSP is configured, you must restart the telephony service before an application can run and connect with its devices.

Procedure for Windows NT, Windows 98, and Windows 95

Step 1 Open the Control Panel.

Step 2 Double-click **Telephony**.

Step 3 Choose the Cisco Unified TSP that you want to configure.

Step 4 Click **Configure**.

The system displays the Cisco IP PBX Service Provider dialog box.

Step 5 Enter the appropriate settings as described in the "Cisco Unified TSP Configuration Settings" section.

Step 6 Click **OK** to save changes.

Note After configuring the TSP, you must restart the telephony service before an application can run and connect with its devices.

Cisco Unified TSP Configuration Settings

The following sections describe the tabs in the Cisco-IP PBX Service Provider dialog box:

- General
- User
- CTI Manager
- Wave
- Trace
- Advanced
- Language

General

The General Tab displays TSP and TSPUI version information, as illustrated in Figure 4-1.

Figure 4-1 Cisco IP PBX Service Provider General Tab

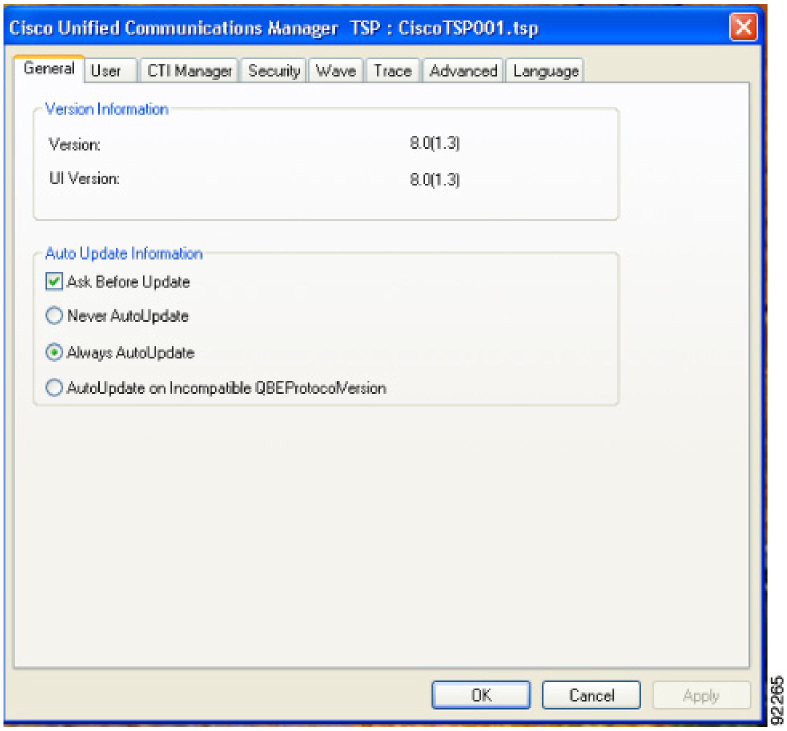


Table 4-1 contains a list of the General tab fields that must be set and their descriptions.

Table 4-1 Auto Update Information Fields

Field	Description
Ask Before Update	Enables the user to control the auto update process. This check box is disabled by default.
Never AutoUpdate	Figure 4-1 shows the default value. Choosing this radio button does not perform an auto update even after an upgradeable plug-in version is detected on the Cisco Unified Communications Manager.
Always AutoUpdate	Choose this radio button to allow the Cisco TSP to auto update after it detects an upgradeable plug-in version on the Cisco Unified Communications Manager.
AutoUpdate on Incompatible QBEProtocolVersion	Choose this radio button to allow the Cisco TSP to auto update only when the local TSP version is incompatible with the Cisco Unified Communications Manager, and upgrading the TSP to the plug-in version represents the only choice to continue.

User

The User tab allows you to configure security information, as illustrated in Figure 4-2.

Figure 4-2 Cisco IP PBX Service Provider User Tab

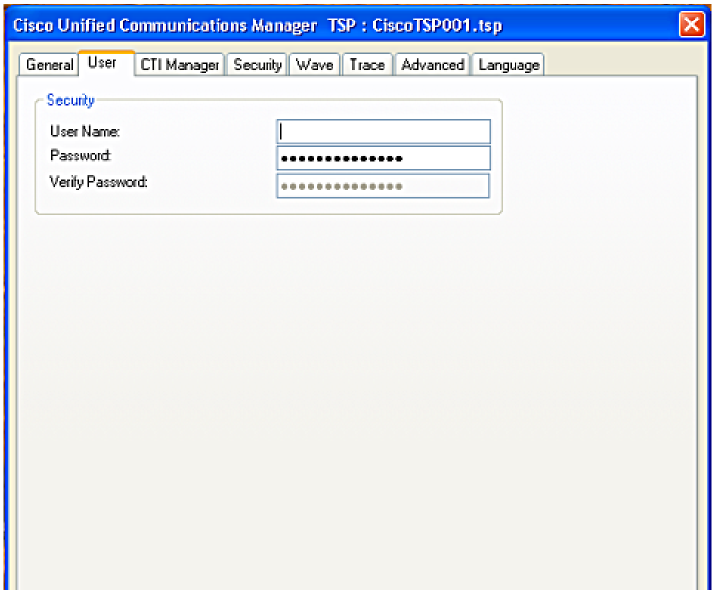


Table 4-2 contains a list of the fields for the User tab that must be set and their descriptions.

Table 4-2 User Tab Configuration Fields

Field	Description
User Name	Enter the user name of the user that you want to access devices. This TSP can access devices and lines that are associated with this user. Make sure that this user is also configured in the Cisco Unified Communications Manager, so TSP can connect. The TSP configuration registry keys store the user name and password that you enter. **Note** You can designate only one user name and password to be active at any time for a TSP.
Password	Enter the password that is associated with the user that you entered in the User Name field. The computer encrypts the password and stores it in the registry.
Verify Password	Reenter the user password.

CTI Manager

The CTI Manager tab allows you to configure primary and secondary CTI Manager information, as illustrated in Figure 4-3.

Figure 4-3 Cisco-IP PBX Service Provider CTI Manager Tab

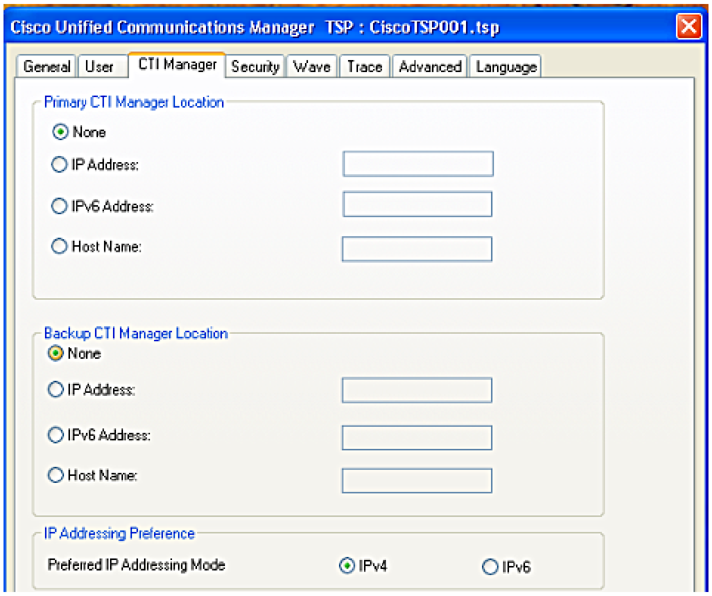


Table 4-3 contains a list of the CTI Manager tab fields that must be set and their descriptions.

Table 4-3 CTI Manager Configuration Fields

Field	Description
Primary CTI Manager Location	Use this field to specify the CTI Manager to which the TSP attempts to connect first. If the TSP is on the same computer as the primary CTI Manager, choose the Local Host radio button. If the primary CTI Manager is on a different computer, choose the IP Address radio button and enter the IP address of primary CTI Manager or choose the Host Name radio button and enter the host name of primary CTI Manager.
Backup CTI Manager Location	Use this field to specify the CTI Manager to which the TSP attempts to connect if a connection to the primary CTI Manager fails. If the TSP is on the same computer as the backup CTI Manager, choose the Local Host radio button. If the backup CTI Manager is on a different computer, choose the IP Address radio button and enter the IP address of backup CTI Manager or choose the Host Name radio button and enter the host name of backup CTI Manager.

Wave

The Wave tab allows you to configure settings for your wave devices, as illustrated in Figure 4-4.

Figure 4-4 Cisco IP PBX Service Provider Wave Tab

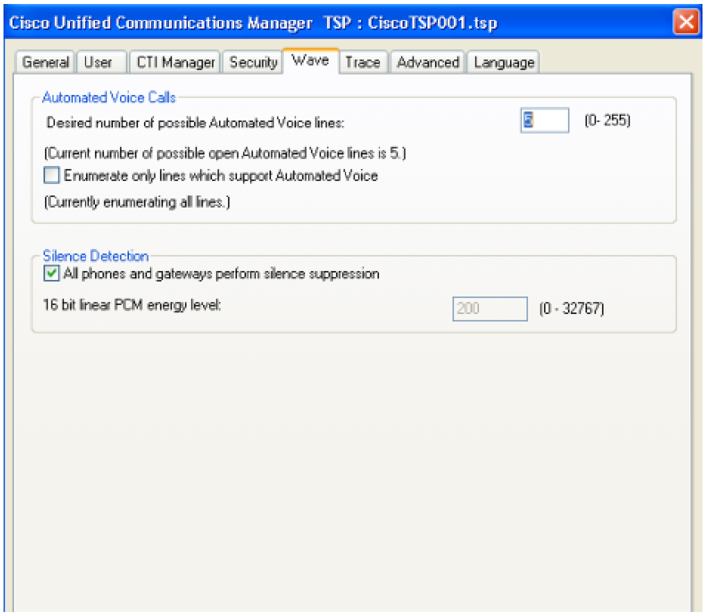


Table 4-4 contains a list of the Wave tab fields that must be set and their descriptions.

Table 4-4 Wave Tab Configuration Fields

Field	Description
Automated Voice Calls	The number of Cisco wave devices that you are using determines the possible number of automated voice lines. (The default value is 5.) You can open as many CTI ports as the number of Cisco wave devices that are configured. For example, if you enter "5," you need to create five CTI port devices in Cisco Unified Communications Manager. If you change this number, you need to remove and then reinstall any Cisco wave devices that you installed. You can only configure a maximum of 255 wave devices for all installed TSPs because Microsoft limits the number of wave devices per wave driver to 255. When you configure 256 or more wave devices (including Cisco or other wave devices), Windows displays the following message when you access the Sounds and Multimedia control panel: "An Error occurred while Windows was working with the Control Panel file C:\Winnt\System32\MMSYS.CPL." TSP can still handle the installed Cisco wave devices as long as you have not configured more than 255 Cisco devices. The current number of possible automated voice lines designates the maximum number of lines that can be simultaneously opened by using both LINEMEDIAMODE_AUTOMATEDVOICE and LINEMEDIAMODE_INTERACTIVEVOICE. If you are not developing a third-party call control application, check the Enumerate only lines that support automated voice check box, so the Cisco Unified TSP detects only lines that are associated with a CTI port device.
Silence Detection	If you use silence detection, this check box notifies the wave driver of method to use to detect silence on lines that support automated voice calls that are using the Cisco Wave Driver. If the check box is checked (default), the wave driver searches for the absence of audio-stream RTP packets. Because all devices on the network suppress silence and stop sending packets, this method provides a very efficient way for the wave driver to detect silence. However, if some phones or gateways do not perform silence suppression, the wave driver must analyze the content of the media stream and, at some threshold, declare that silence is in effect. This CPU-intensive method handles media streams from any type of device. If some phones or gateways on your network do not perform silence suppression, you must specify the energy level at which the wave driver declares that silence is in effect. This value of the 16-bit linear energy level ranges from 0 to 32767, and the default value is 200. If all phones and gateways perform silence suppression, the system ignores this value.

Trace

The Trace tab allows you to configure various trace settings, as illustrated in Figure 4-5. Changes to trace parameters take effect immediately, even if TSP is running.

Figure 4-5 Cisco IP PBX Service Provider Trace Tab

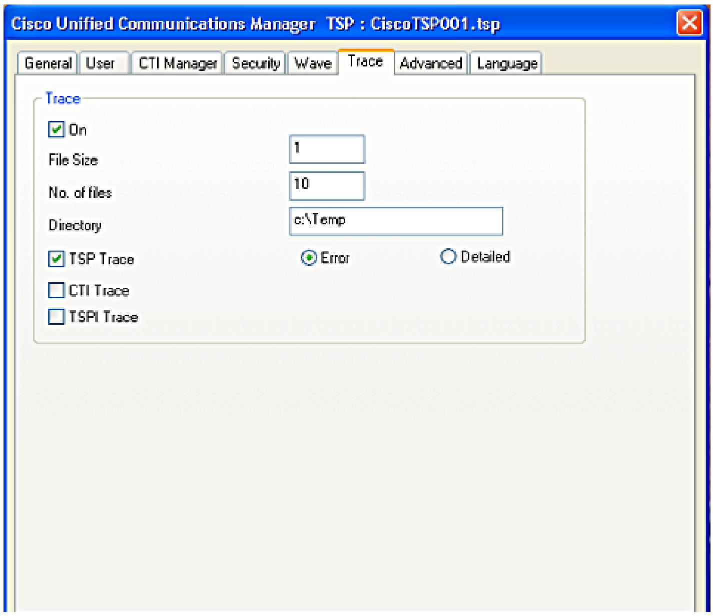


Table 4-5 contains a list of the Trace tab fields that must be set and their descriptions.

Table 4-5 Trace Tab Configuration Fields

Field	Description
On	This setting allows you to enable Global Cisco TSP trace. Select the check box to enable Cisco TSP trace. When you enable trace, you can modify other trace parameters in the dialog box. The Cisco TSP trace depends on the values that you enter in these fields. Clear the check box to disable Cisco TSP trace. When you disable trace, you cannot choose any trace parameters in the dialog box, and TSP ignores the values that are entered in these fields.
Max lines/file	Use this field to specify the maximum number of lines that the trace file can contain. The default value is 10,000. After the file contains the maximum number of lines, trace opens the next file and writes to that file.
No. of files	Use this field to specify the maximum number of trace files. The default value is 10. File numbering occurs in a rotating sequence starting at 0. The counter restarts at 0 after it reaches the maximum number of files minus one.

Table 4-5 Trace Tab Configuration Fields (continued)

Field	Description
Directory	Use this field to specify the location in which trace files for all Cisco Unified TSPs are stored. Make sure that the specified directory exists. The system creates a subdirectory for each Cisco Unified TSP. For example, the CiscoTSP001Log directory stores Cisco Unified TSP 1 log files. The system creates trace files with filename TSP001Debug000xxx.txt for each TSP in its respective subdirectory.
TSP Trace	This setting activates internal TSP tracing. When you activate TSP tracing, Cisco Unified TSP logs internal debug information that you can use for debugging purposes. You can choose one of the following levels: Error—Logs only TSP errors. Detailed—Logs all TSP details (such as log function calls in the order that they are called). The system checks the TSP Trace check box and chooses the Error radio button by default.
CTI Trace	This setting traces messages that flow between Cisco Unified TSP and CTI. Cisco Unified TSP communicates with the CTI Manager. By default, the system leaves the check box unchecked.
TSPI Trace	This setting traces all messages and function calls between TAPI and Cisco Unified TSP. The system leaves this check box unchecked by default. If you check the check box, TSP traces all the function calls that TAPI makes to Cisco Unified TSP with parameters and messages (events) from Cisco Unified TSP to TAPI.

Advanced

The Advanced tab allows you to configure timer settings, as illustrated in Figure 4-6.

Note These timer settings that are meant for advanced users only rarely change.

Figure 4-6 Cisco IP PBX Service Provider Advanced Tab

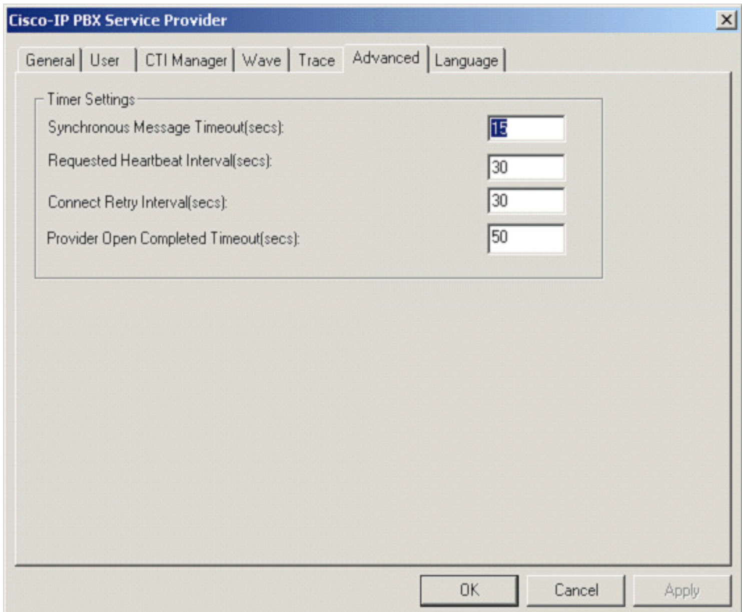


Table 4-6 contains a list of the Advanced tab fields that must be set and their descriptions.

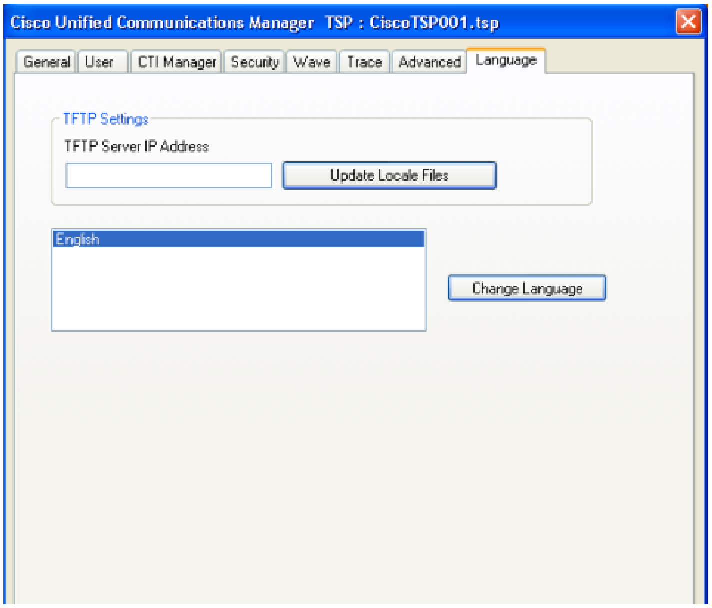
Table 4-6 Advanced Configuration Fields

Field	Description
Synchronous Message Timeout (secs)	Use this field to designate the time that the TSP waits to receive a response to a synchronous message. The value displays in seconds, and the default value is 15. Range goes from 5 to 60 seconds.
Requested Heartbeat Interval (secs)	Use this field to designate the interval at which the heartbeat messages are sent from TSP to detect whether the CTI Manager connection is still alive. TSP sends heartbeats when no traffic exists between the TSP and CTI Manager for 30 seconds or more. The default interval is 30 seconds. Range goes from 30 to 300 seconds.
Connect Retry Interval (secs)	Use this field to designate the interval between reconnection attempts after a CTI Manager connection failure. The default value is 30 seconds. Range goes from 15 to 300 seconds.
Provider Open Completed Timeout (secs)	Use this field to designate the time that Cisco Unified TSP waits for a Provider Open Completed Event, which indicates the CTI Manager is initialized and ready to serve TSP requests. Be aware that CTI initialization time is directly proportional to the number of devices that are configured in the system. The default value is 50 seconds. Range goes from 5 to 900 seconds.

Language

The Language tab allows you to choose one of the installed languages to view the configuration settings in that language, as illustrated in Figure 4-7.

Figure 4-7 Cisco IP PBX Service Provider Language Tab



Choose a language and click **Change Language** to reload the tabs with the text in that language.

Installing the Wave Driver

You can use the Cisco Wave Driver with Windows 2000 and Windows NT only. Windows 98 and Windows 95 do not support the Cisco Wave Driver.

You should install Cisco Wave Driver if you plan to use first-party call control. (Do this even if you are performing your own media termination.)

Caution Because of a restriction in Windows NT, the software may overwrite or remove existing wave drivers from the system when you install or remove the Cisco wave driver on a Windows NT system. The procedures in this section for installing and uninstalling the Cisco wave driver on Windows NT include instructions on how to prevent existing wave drivers from being overwritten or removed.

To install the Cisco Wave Driver, perform the following steps.

Procedure for Windows XP

Step 1 Open the Control Panel.

Step 2 Open **Add/Remove Hardware**.

Step 3 Click **Next**.

Step 4 Select **Yes, I have already connected the hardware**.

Step 5 Select **Add a New Hardware Device**.

Step 6 Click **Next**.

Step 7 Select **Install the Hardware that I manually select from a list**.

Step 8 Click **Next**.

Step 9 For the hardware type, choose **Sound, video and game controller**.

Step 10 Click **Next**.

Step 11 Click **Have Disk**.

Step 12 Click **Browse** and navigate to the Wave Drivers folder in the folder where the Cisco Unified TSP is installed.

Step 13 Choose **OEMSETUP.INF** and click **Open**.

Step 14 In the Install From Disk window, click **OK**.

Step 15 In the Select a Device Driver window, select the **Cisco Unified TAPI Wave Driver** and click **Next**.

Step 16 In the Start Hardware Installation window, click **Next**.

Step 17 If Prompted for Digital signature Not Found, click **Continue Anyway**.

Step 18 The installation may issue the following prompt:

> The file avaudio32.dll on Windows NT Setup Disk #1 is needed,
> Type the path where the file is located and then click ok.

If so, navigate to the same location where you chose OEMSETUP.INF, select avaudio32.dll, and click **OK**.

Step 19 Click **Yes**.

Step 20 Click **Finish**.

Step 21 To restart to restart the computer, click **Yes**.

Procedure for Windows 2000

Step 1 Open the Control Panel.

Step 2 Double-click **Add/Remove Hardware**.

Step 3 Click **Next**.

Step 4 Click **Add/Troubleshoot a Device** and click **Next**.

Step 5 Click **Add a New Device** and click **Next**.

Step 6 Click **No, I want to select the hardware from a list**.

Step 7 Choose **Sound, video and game controllers** and click **Next**.

Step 8 Click **Have Disk**.

Step 9 Click **Browse** and change to the Wave Drivers folder in the folder where the Cisco Unified TSP is installed.

Step 10 Choose **OEMSETUP.INF** and click **Open**.

Step 11 In the Install From Disk window, click **OK**.

Step 12 The Cisco Unified TAPI Wave Driver displays. Click **Next**.

Step 13 Click **Next**.

Step 14 The installation may issue the following prompt:

Digital Signature Not Found

Step 15 Click **Yes**.

Step 16 The installation may issue the following prompt:

The file avaudio32.dll on Windows NT Setup Disk #1 is needed,
Type the path where the file is located and then click ok.

If so, enter the same location as where you chose OEMSETUP.INF and click **OK**.

Step 17 Click **Yes**.

Step 18 Click **Finish**.

Step 19 To restart, click **Yes**.

Procedure for Windows NT

Step 1 Before you add the Cisco Wave Driver, you must save the wave driver information from the registry in a separate file as described in the "Saving Wave Driver Information" section.

Step 2 Open the Control Panel.

Step 3 Double-click **Multimedia**.

Step 4 Click **Next**.

Step 5 Click **Add**.

Step 6 Click **Unlisted** or **Updated Driver**.

Step 7 Click **OK.**

Step 8 Click **Browse** and change to the Wave Drivers folder in the folder where the Cisco Unified TSP is installed.

Step 9 Click **OK**. Follow the online instruction, but do not restart the system when prompted.

Step 10 Examine the contents of the registry to verify the new driver was installed and the old drivers still exist, as described in the "Verifying the Wave Driver Exists" section.

Step 11 Restart the computer.

Saving Wave Driver Information

Use the following steps to save wave driver information from the registry in a separate file. You must perform this procedure when installing or uninstalling the Cisco Wave Driver on a Windows NT computer.

Procedure

Step 1 Click **Start > Run.**

Step 2 In the text box, enter **regedit**.

Step 3 Click **OK.**

Step 4 Choose the Drivers32 key that is located in the following path:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\ CurrentVersion

Step 5 Choose **Registry > Export Registry File**.

Step 6 Enter a filename and choose the location to save.

Step 7 Click **Save**.

The file receives a .reg extension.

Verifying the Wave Driver Exists

When you install or uninstall the Cisco Wave Driver, you must verify whether it exists on your system. Use these steps to verify whether the wave driver exists.

Procedure

Step 1 Click **Start > Run**.

Step 2 In the text box, enter **regedit**.

Step 3 Click **OK**.

Step 4 Choose the Drivers32 key that is located in the following path:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\ CurrentVersion

Step 5 If you are installing the wave driver, make sure that the driver "avaudio32.dll" displays in the data column. If you are uninstalling the wave driver, make sure that the driver "avaudio32.dll" does not display in the data column. This designates the Cisco Wave Driver.

Step 6 Verify that the previously existing wave values appear in the data column for wave1, wave2, wave3, and so on. You can compare this registry list to the contents of the .reg file that you saved in the "Saving Wave Driver Information" procedure by opening the .reg file in a text editor and viewing it and the registry window side by side.

Step 7 If necessary, add the appropriate waveX string values for any missing wave values that should be installed on the system. For each missing wave value, choose **Edit > New > String Value** and enter a value name. Then, choose **Edit > Modify**, enter the value data, and click **OK**.

Step 8 Close the registry by choosing **Registry > Exit**.

Verifying the Cisco Unified TSP Installation

You can use the Microsoft Windows Phone Dialer Application to verify that the Cisco Unified TSP is operational. For Windows NT and Windows 2000, locate the dialer application in C:\Program Files\Windows NT\dialer.exe

For Windows 95 and Windows 98, locate the dialer application in C:\Windows\dialer.exe

Procedure For Windows 2000 and Windows XP

Step 1 Open the Dialer application by locating it in Windows Explorer and double-clicking it.

Step 2 Choose **Edit > Options**.

Step 3 Choose **Phone** as the Preferred Line for Calling.

Step 4 In the Line Used For area, choose one Cisco Line in the Phone Calls drop-down menu.

Step 5 Click **OK**.

Step 6 Click **Dial**.

Step 7 Enter a number to dial, choose **Phone Call** in the Dial as box, and then click **Place Call**.

Procedure for Windows NT, Windows 98, and Windows 95

Step 1 Open the Dialer application by locating it in Windows Explorer and double-clicking it:

A dialog box displays that requests the line and address that you want to use. If no lines are listed in the **Line** drop-down list box, a problem may exist between the Cisco Unified TSP and the Cisco Unified Communications Manager.

Step 2 Choose a line from the Line drop-down menu. Make sure Address is set to **Address 0**.

Step 3 Click **OK**.

Step 4 Enter a number to dial.

If the call is successful, you have verified that the Cisco Unified TSP is operational on the machine where the Cisco Unified TSP is installed.

If you encounter problems during this procedure, or if no lines appear in the line drop-down list on the dialer application, check the following items:

- Make sure that the Cisco Unified TSP is configured properly.
- Test the network link between the Cisco Unified TSP and the Cisco Unified Communications Manager by using the ping command to check connectivity.
- Make sure that the Cisco Unified Communications Manager server is functioning.

Setting Up Client-Server Configuration

For information on setting up a client-server configuration (Remote TSP) in Windows 2000, refer to the Microsoft Windows Help feature. For information on client-server configuration in Windows NT, refer to Microsoft White Papers.

Uninstalling the Wave Driver

To remove the Cisco Wave Driver, perform the following steps.

Procedure for Windows XP

Step 1 Open the Control Panel.

Step 2 Select **Sound and Audio Devices**.

Step 3 Click the **Hardware** tab.

Step 4 Select **Cisco TAPI Wave Driver**.

Step 5 Click **Properties.**

Step 6 Click the **Driver** tab.

Step 7 Click **Uninstall** and **OK** to remove.

Step 8 If the Cisco TAPI Wave Driver entry is still displayed, close and open the window again to verify that it has been removed.

Step 9 Restart the computer.

Procedure for Windows 2000

Step 1 Open the Control Panel.

Step 2 Double-click **Add/Remove Hardware**.

Step 3 Click **Next**.

Step 4 Choose **Uninstall/Unplug a device** and click **Next**.

Step 5 Choose **Uninstall a device** and click **Next**.

Step 6 Choose **Cisco TAPI Wave Driver** and click **Next**.

Step 7 Choose **Yes, I want to uninstall this device** and click **Next**.

Step 8 Click **Finish**.

Step 9 Restart the computer.

Procedure for Windows NT

Step 1 Before you uninstall the Cisco Wave Driver, you must save the wave driver information from the registry in a separate file. For information on how to save the wave drive information to a separate file, see the "Saving Wave Driver Information" section.

Step 2 After the registry information is saved, open the Control Panel.

Step 3 Double-click **Multimedia**.

Step 4 Click the **Devices** tab.

Step 5 To view all the audio devices, click the '**+**' symbol next to Audio Devices.

Step 6 Click **Audio** for Cisco Sound System.

Step 7 Click **Remove**.

Step 8 Click **Finish**. Do not restart the system.

Step 9 Verify that the Cisco Wave Driver was removed and the old drivers still exist. For information on how to do this, see the "Verifying the Wave Driver Exists" section.

> **Note** When you verify the removal of the driver, make sure that Cisco Wave Driver "avaudio32.dll" does not appear in the data column.

Step 10 Restart the computer.

Removing the Cisco Unified TSP

This process removes the Cisco Unified TSP from the provider list but does not uninstall the TSP. To make these changes, perform the following steps.

Procedure for Windows 2000

Step 1 Open the Control Panel.

Step 2 Double-click the **Phone and Modem** icon.

Step 3 Click the **Advanced** tab.

Step 4 Choose the Cisco Unified TSP that you want to remove.

Step 5 To delete the Cisco Unified TSP from the list, click **Remove**.

Procedure for Windows NT, Windows 98, and Windows 95

Step 1 Open the Control Panel.

Step 2 Double-click the **Telephony** icon.

Step 3 Click the **Advanced** tab.

Step 4 Choose the Cisco Unified TSP that you want to remove.

Step 5 To delete the Cisco Unified TSP from the list, click **Remove**.

Managing the Cisco Unified TSP

You can perform the following actions on all installed TSPs:

- Reinstall the existing Cisco Unified TSP version.
- Upgrade to the newer version of the Cisco Unified TSP.
- Uninstall the Cisco Unified TSP.

You cannot change the number of installed Cisco Unified TSPs when you reinstall or upgrade the Cisco Unified TSPs.

Related Topics

- Reinstalling the Cisco Unified TSP
- Upgrading the Cisco Unified TSP
- Auto Update for Cisco Unified TSP Upgrades
- Uninstalling the Cisco Unified TSP

Reinstalling the Cisco Unified TSP

Use the following procedure to reinstall the Cisco Unified TSP on all supported platforms.

Procedure

Step 1 Open the Control Panel and double-click **Add/Remove Programs**.

Step 2 Choose Cisco Unified TSP and click **Add/Remove**.

The Cisco Unified TSP maintenance install dialog box displays.

Step 3 Click **Reinstall TSP 4.1(X.X)** radio button and click **Next**.

Step 4 Follow the online instructions.

Note If TSP files are already locked, the installation program prompts you to restart the computer.

Upgrading the Cisco Unified TSP

Use the following procedure to upgrade the Cisco Unified TSP on all supported platforms.

Procedure

Step 1 Choose the type of installation for Cisco Unified Communications Manager TSP 4.1(X.X).

Step 2 Choose **Upgrade from TSP X.X(X.X) option** radio button and click **Next**.

Step 3 Follow the online instructions.

Note If TSP files are already locked, the installation program prompts you to restart the computer.

Step 4 The Cisco TSP maintenance install dialog box displays.

If CiscoTSP.exe contains different version of Cisco Unified TSP than you have installed, the installation program displays one of the following prompts, depending upon the previous Cisco Unified TSP version:

Choose the type of installation for TSP Version 4.1(X.X).

- If the previous installed version is Cisco Unified TSP 3.1(X.X), the following prompt displays:

  Upgrade from TSP 3.1(X.X)
- If the previous installed version is Cisco Unified TSP 3.2(X.X), the following prompt displays:

  Upgrade from TSP 3.2(X.X)
- If the previous installed version is Cisco Unified TSP 3.3(X.X), the following prompt displays:

  Upgrade from 3.3(X.X)
- If the previous installed version is Cisco Unified TSP 4.1(X.X), the following prompt displays:

  Upgrade from TSP 4.1(X.X)

Auto Update for Cisco Unified TSP Upgrades

Cisco TSP supports auto update functionality, so you can download the latest plug-in and install it on the client machine. When the Cisco Unified Communications Manager is upgraded to a higher version, and Cisco TSP auto update functionality is enabled, this means that the latest compatible Cisco TSP is available, which is compatible with the upgraded Unified CM. This ensures that the applications work as expected with the new release (provided the new call manager interface is backward compatible with the TAPI interface). The Cisco TSP that is installed locally on the client server allows the application to set the auto update options as part of the Cisco TSP configuration. You can opt for updating the Cisco TSP in the following different ways.

- Update Cisco TSP whenever a different (has to be higher version than existing one) version is available on the Cisco Unified Communications Manager server.
- Update Cisco TSP whenever a QBE protocol version mismatch occurs between the existing Cisco TSP and the Cisco Unified Communications Manager version.
- Do not update Cisco TSP by using the auto update functionality.

Auto Update Behavior

As part of initialization of Cisco TSP, when the application does lineInitializeEx, Cisco TSP queries the current TSP plug-in version information that is available on Cisco Unified Communications Manager server. After this information is available, Cisco TSP compares the installed Cisco TSP version with the plug-in version. If user chose an option for Auto Update, Cisco TSP triggers the update process. As part of Auto Update, Cisco TSP behaves in the following ways on different platforms.

Windows 95, Windows 98, Windows ME

Because Cisco TSP is in use and locked when the application does lineInitializeEx, the auto update process requests that you close all the running applications to install the new TSP version on the client setup. When all the running applications get closed, Cisco TSP auto update process can continue, and you will be informed about the upgrade success. If the running applications do not get closed and the installation continues, the new version of Cisco TSP will not get installed, and a corresponding error gets reported to the applications.

Windows NT

After Cisco TSP detects that an upgradeable version is available on the Cisco Unified Communications Manager server and Auto Update gets chosen, Cisco TSP reports 0 lines to the application and removes the Cisco TSP provider from the provider list. It will then try to stop the telephony service to avoid any locked files during Auto Update. If the telephony service can be stopped, Cisco TSP gets silently auto updated, and the service gets restarted. Applications must be reinitialized to start using Cisco TSP. If the telephony service could not be stopped, Cisco TSP installs the new version and displays a message to restart the system. You must restart the system to use the new Cisco TSP.

Windows 2000 or XP

After Cisco TSP detects that an upgradeable version is available on the Cisco Unified Communications Manager server and Auto Update option gets chosen, Cisco TSP reports 0 lines to the application and removes the Cisco TSP provider from the provider list. If a new TSP version is detected during the reconnect time, the running applications receive LINE_REMOVE on all the lines, which are already initialized and are in OutOfService state. Cisco TSP silently upgrades to the new version that was downloaded from the Cisco Unified Communications Manager and puts the Cisco TSP provider back on the provider list. All the running applications receive LINE_CREATE messages.

WinXP supports multiple user logon sessions (fast user switching); however, the system supports Auto Update only for the first logon user. If multiple active logon sessions exist, Cisco TSP only supports the Auto Update functionality for the first logged-on user.

Note If a user has multiple Cisco TSPs installed on the client machine, the system enables only the first Cisco TSP instance to set up the Auto Update configuration. All Cisco TSPs get upgraded to a common version upon version mismatch. From “Control Panel/Phone & Modem Options/Advanced/CiscoTSP001,” the General window displays the options for Auto Update.

Because it is a CTI service parameter, which can be configured, you can change the plug-in location to a different machine than the Cisco Unified Communications Manager server. The default location is “//<CMServer>//ccmpluginsserver”.

If Silent upgrade fails on any listed platforms for any reason (such as locked files that are encountered during upgrade on Win95/98/ME), the old Cisco TSP provider(s) do not get put back on the provider list to avoid any looping of the Auto Update process. Ensure that the update options get cleared and the providers get added to provider list manually. Update the Cisco TSP manually or by fixing the problem(s) that are encountered during Auto Update and reinitializing Cisco Unified TAPI to trigger the Auto Update process.

Note TSPAutoinstall.exe, which has user interface windows, can proceed to display these windows only when the telephony service enables the LocalSystem logon option with “Allow Service to interact with Desktop.” If the logon option is not set as LocalSystem or logon option is LocalSystem but “Allow Service to interact with Desktop” is disabled, Cisco TSP cannot launch the AutoInstall UI windows and will not continue with AutoInstall.

Ensure that the following logon options are set for the telephony service.

Step 1 Logon as: **LocalSystem.**

Step 2 Enable the check box: "Allow Service to interact with Desktop."

These telephony service settings, when changed, require manual restart of the service to take effect.

Step 3 If, after changing the settings to the preceding values, the service does not restart, Cisco TSP checks for "Allow Service to interact with user" to be positive (as the configuration is updated for the service in the database), but AutoInstall UI cannot display. Cisco TSP continues to put the entry for TSPAutoInstall.exe under Registry key RUNONCE. This will help autoinstall to run when the machine reboots the next time.

Uninstalling the Cisco Unified TSP

Use the following procedure to uninstall the Cisco Unified TSP on all supported platforms.

Procedure

Step 1 Open the Control Panel and double-click **Add/Remove Programs**.

Step 2 Choose Cisco Unified TSP and click **Add/Remove**.

The Cisco Unified TSP maintenance install dialog box displays.

Step 3 Choose **Uninstall: Remove the installed TSP** radio button and click **Next**.

Step 4 Follow the online instructions.

Note If TSP files are already locked, the installation program prompts you to restart the computer.

CHAPTER 5

Basic TAPI Implementation

This chapter outlines the TAPI 2.1 functions, events, and messages that the Cisco Unified TAPI Service Provider (TSP) supports. This chapter contains functions in the following sections:

Overview

TAPI comprises a set of classes that expose the functionality of the Cisco Unified Communications Solutions. TAPI enables developers to create customized IP telephony applications for Cisco Unified Communications Manager without specific knowledge of the communication protocols between the Cisco Unified Communications Manager and the service provider. For example, a developer could create a TAPI application that communicates with an external voice-messaging system.

TAPI Line Functions

The number of TAPI devices that are configured in the Cisco Unified Communications Manager determines the number of available lines. To terminate an audio stream by using first-party control, you must first install the Cisco wave device driver.

Table 5-1 TAPI Line Functions Supported

TAPI Line Functions Supported
lineAccept
lineAddProvider
lineAddToConference

Table 5-1 TAPI Line Functions Supported (continued)

TAPI Line Functions Supported
lineAnswer
lineBlindTransfer
lineCallbackFunc
lineClose
lineCompleteTransfer
lineConfigProvider
lineDeallocateCall
lineDevSpecific
lineDevSpecificFeature
lineDial
lineDrop
lineForward
lineGenerateDigits
lineGenerateTone
lineGetAddressCaps
lineGetAddressID
lineGetAddressStatus
lineGetCallInfo
lineGetCallStatus
lineGetConfRelatedCalls
lineGetDevCaps
lineGetID
lineGetLineDevStatus
lineGetMessage
lineGetNewCalls
lineGetNumRings
lineGetProviderList
lineGetRequest
lineGetStatusMessages
lineGetTranslateCaps
lineHandoff
lineHold
lineInitialize
lineInitializeEx
lineMakeCall
lineMonitorDigits

Table 5-1 TAPI Line Functions Supported (continued)

TAPI Line Functions Supported
lineMonitorTones
lineNegotiateAPIVersion
lineNegotiateExtVersion
lineOpen
linePark
linePrepareAddToConference
lineRedirect
lineRegisterRequestRecipient
lineRemoveFromConference
lineSetAppPriority
lineSetCallPrivilege
lineSetNumRings
lineSetStatusMessages
lineSetTollList
lineSetupConference
lineSetupTransfer
lineShutdown
lineTranslateAddress
lineTranslateDialog
lineUnhold
lineUnpark

lineAccept

The lineAccept function accepts the specified offered call.

Function Details

```
LONG lineAccept(
    HCALL hCall,
    LPCSTR lpsUserUserInfo,
    DWORD dwSize
);
```

Parameters

hCall

A handle to the call to be accepted. The application must be an owner of the call. Call state of hCall must be offering.

lpsUserUserInfo

A pointer to a string that contains user-user information to be sent to the remote party as part of the call accept. Leave this pointer NULL if you do not want to send user-user information. User-user information is sent only if supported by the underlying network. The protocol discriminator member for the user-user information, if required, should appear as the first byte of the buffer that is pointed to by lpsUserUserInfo and must be accounted for in dwSize.

Note The Cisco Unified TSP does not support user-user information.

dwSize

The size in bytes of the user-user information in lpsUserUserInfo. If lpsUserUserInfo is NULL, no user-user information gets sent to the calling party, and dwSize is ignored.

lineAddProvider

The lineAddProvider function installs a new telephony service provider into the telephony system.

Function Details

```
LONG WINAPI lineAddProvider(
  LPCSTR lpszProviderFilename,
  HWND hwndOwner,
  LPDWORD lpdwPermanentProviderID
);
```

Parameters

lpszProviderFilename

A pointer to a null-terminated string that contains the path of the service provider to be added.

hwndOwner

A handle to a window in which dialog boxes that need to be displayed as part of the installation process (for example, by the service provider's TSPI_providerInstall function) would be attached. Can be NULL to indicate that any window created during the function should have no owner window.

lpdwPermanentProviderID

A pointer to a DWORD-sized memory location into which TAPI writes the permanent provider identifier of the newly installed service provider.

Return Values

Returns zero if request succeeds or a negative number if an error occurs. Possible return values are:

- LINEERR_INIFILECORRUPT
- LINEERR_NOMEM
- LINEERR_INVALPARAM
- LINEERR_NOMULTIPLEINSTANCE
- LINEERR_INVALPOINTER
- LINEERR_OPERATIONFAILED

lineAddToConference

This function takes the consult call that is specified by hConsultCall and adds it to the conference call that is specified by hConfCall.

Function Details

```
LONG lineAddToConference(
  HCALL hConfCall,
  HCALL hConsultCall
);
```

Parameters

hConfCall

A pointer to the conference call handle. The state of the conference call must be OnHoldPendingConference or OnHold.

hConsultCall

A pointer to the consult call that will be added to the conference call. The application must be the owner of this call, and it cannot be a member of another conference call. The allowed states of the consult call comprise connected, onHold, proceeding, or ringback

lineAnswer

The lineAnswer function answers the specified offering call.

Note CallProcessing requires previous calls on the device to be in connected call state before answering further calls on the same device. If calls are answered without checking for the call state of previous calls on the same device, then Cisco Unified TSP might return a successful answer response but the call will not go to connected state and needs to be answered again.

Function Details

```
LONG lineAnswer(
  HCALL hCall,
  LPCSTR lpsUserUserInfo,
  DWORD dwSize
);
```

Parameters

hCall

A handle to the call to be answered. The application must be an owner of this call. The call state of hCall must be offering or accepted.

lpsUserUserInfo

A pointer to a string that contains user-user information to be sent to the remote party at the time the call is answered. You can leave this pointer NULL if no user-user information will be sent.

User-user information only gets sent if supported by the underlying network. The protocol discriminator field for the user-user information, if required, should be the first byte of the buffer that is pointed to by lpsUserUserInfo and must be accounted for in dwSize.

Note The Cisco Unified TSP does not support user-user information.

dwSize

The size in bytes of the user-user information in lpsUserUserInfo. If lpsUserUserInfo is NULL, no user-user information gets sent to the calling party, and dwSize is ignored.

lineBlindTransfer

The lineBlindTransfer function performs a blind or single-step transfer of the specified call to the specified destination address.

Note The lineBlindTransfer function that is implemented until Cisco Unified TSP 3.3 does not comply with the TAPI specification. This function actually gets implemented as a consultation transfer and not a single-step transfer. From Cisco Unified TSP 4.0, the lineBlindTransfer complies with the TAPI specs wherein the transfer is a single-step transfer.

If the application tries to blind transfer a call to an address that requires a FAC, CMC, or both, then the lineBlindTransfer function will return an error. If a FAC is required, the TSP will return the error LINEERR_FACREQUIRED. If a CMC is required, the TSP will return the error LINEERR_CMCREQUIRED. If both a FAC and a CMC are required, the TSP will return the error LINEERR_FACANDCMCREQUIRED. An application that wants to blind transfer a call to an address that requires a FAC, CMC, or both, should use the lineDevSpecific - BlindTransferFACCMC function.

Function Details

```
LONG lineBlindTransfer(
  HCALL hCall,
  LPCSTR lpszDestAddress,
  DWORD dwCountryCode
);
```

Parameters

hCall

A handle to the call to be transferred. The application must be an owner of this call. The call state of hCall must be connected.

lpszDestAddress

A pointer to a NULL-terminated string that identifies the location to which the call is to be transferred. The destination address uses the standard dial number format.

dwCountryCode

The country code of the destination. The implementation uses this parameter to select the call progress protocols for the destination address. If a value of 0 is specified, the defined default call-progress protocol is used.

lineCallbackFunc

The lineCallbackFunc function provides a placeholder for the application-supplied function name.

Function Details

```
VOID FAR PASCAL lineCallbackFunc(
  DWORD hDevice,
  DWORD dwMsg,
  DWORD dwCallbackInstance,
  DWORD dwParam1,
  DWORD dwParam2,
  DWORD dwParam3
);
```

Parameters

hDevice

A handle to either a line device or a call that is associated with the callback. The context that dwMsg provides determines the nature of this handle (line handle or call handle). Applications must use the DWORD type for this parameter because using the HANDLE type may generate an error.

dwMsg

A line or call device message.

dwCallbackInstance

Callback instance data that is passed back to the application in the callback. TAPI does not interpret DWORD.

dwParam1

A parameter for the message.

dwParam2

A parameter for the message.

dwParam3

A parameter for the message.

Further Details

For information about parameter values that are passed to this function, see “TAPI Line Functions.”

lineClose

The lineClose function closes the specified open line device.

Function Details

```
LONG lineClose(
  HLINE hLine
);
```

Parameter

hLine

A handle to the open line device to be closed. After the line has been successfully closed, this handle no longer remains valid.

lineCompleteTransfer

The lineCompleteTransfer function completes the transfer of the specified call to the party that is connected in the consultation call.

Function Details

```
LONG lineCompleteTransfer(
  HCALL hCall,
  HCALL hConsultCall,
  LPHCALL lphConfCall,
  DWORD dwTransferMode
);
```

Parameters

hCall

A handle to the call to be transferred. The application must be an owner of this call. The call state of hCall must be onHold, onHoldPendingTransfer.

hConsultCall

A handle to the call that represents a connection with the destination of the transfer. The application must be comprise an owner of this call. The call state of hConsultCall must be connected, ringback, busy, or proceeding.

lphConfCall

A pointer to a memory location where an hCall handle can be returned. If dwTransferMode is LINETRANSFERMODE_CONFERENCE, the newly created conference call is returned in lphConfCall and the application becomes the sole owner of the conference call. Otherwise, TAPI ignores this parameter.

dwTransferMode

Specifies how the initiated transfer request is to be resolved. This parameter uses the following LINETRANSFERMODE_ constant:

- LINETRANSFERMODE_TRANSFER—Resolve the initiated transfer by transferring the initial call to the consultation call.
- LINETRANSFERMODE_CONFERENCE—The transfer gets resolved by establishing a three-way conference among the application, the party connected to the initial call, and the party connected to the consultation call. Selecting this option creates a conference call.

lineConfigProvider

The lineConfigProvider function causes a service provider to display its configuration dialog box. This basically provides a straight pass-through to TSPI_providerConfig.

Function Details

```
LONG WINAPI lineConfigProvider(
  HWND hwndOwner,
  DWORD dwPermanentProviderID
);
```

Parameters

hwndOwner

A handle to a window to which the configuration dialog box (displayed by TSPI_providerConfig) is attached. This parameter can equal NULL to indicate that any window that is created during the function should have no owner window.

dwPermanentProviderID

The permanent provider identifier of the service provider to be configured.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INIFILECORRUPT
- LINEERR_NOMEM
- LINEERR_INVALPARAM
- LINEERR_OPERATIONFAILED

lineDeallocateCall

The lineDeallocateCall function deallocates the specified call handle.

Function Details

```
LONG lineDeallocateCall(
  HCALL hCall
);
```

Parameter

hCall

The call handle to be deallocated. An application with monitoring privileges for a call can always deallocate its handle for that call. An application with owner privilege for a call can deallocate its handle unless it is the sole owner of the call and the call is not in the idle state. The call handle is invalid after it is deallocated.

lineDevSpecific

The lineDevSpecific function enables service providers to provide access to features that other TAPI functions do not offer. The extensions are device-specific and the applications must be able to read the extensions to take advantage of these extensions.

When used with the Cisco Unified TSP, lineDevSpecific can be used to:

- Enable the message waiting lamp for a particular line.
- Handle the audio stream (instead of using the provided Cisco wave driver).
- Turn On or Off the reporting of media streaming messages for a particular line.
- Register a CTI port or route point for dynamic media termination.
- Set the IP address and the UDP port of a call at a CTI port or route point with dynamic media termination.
- Redirect a Call and Reset the OriginalCalledID of the call to the party that is the destination of the redirect.
- Redirect a call and set the OriginalCalledID of the call to any party.
- Join two or more calls into one conference call.
- Redirect a Call to a destination that requires a FAC, CMC, or both.

- Blind Transfer a Call to a destination that requires a FAC, CMC, or both.
- Open a CTI port in third party mode.
- Set the SRTP algorithm IDs that a CTI port supports.
- Acquire any CTI-controllable device in the Cisco Unified Communications Manager system, which needs to be opened in super provider mode.
- Deacquire any CTI-controllable device in the Cisco Unified Communications Manager system.
- Trigger the actual line open from the TSP side. This is used for the delayed open mechanism.
- Initiate TalkBack on the Intercom Whisper call of the Intercom line
- Query SpeedDial and Label setting of a Intercom line.
- Set SpeedDial and Label setting of a Intercom line.
- Start monitoring a call
- Start recording of a call
- Stop recording of a call
- Direct call with feature priority (see Do Not Disturb–Reject, page 3-9 for more information.
- Transfer without media
- Direct Transfer
- Message Summary

Note In Cisco Unified TSP Releases 4.0 and later, the TSP no longer supports the ability to perform a SwapHold/SetupTransfer on two calls on a line in the CONNECTED and the ONHOLD call states. Therefore, these calls can be transferred by using lineCompleteTransfer. Cisco Unified TSP Releases 4.0 and later enable to transfer these calls using the lineCompleteTransfer function without having to perform the SwapHold/SetupTransfer beforehand.

Function Details

```
LONG lineDevSpecific(
  HLINE hLine,
  DWORD dwAddressID,
  HCALL hCall,
  LPVOID lpParams,
  DWORD dwSize
);
```

Parameters

hLine

A handle to a line device. This parameter is required.

dwAddressID

An address identifier on the given line device.

hCall

A handle to a call. Although this parameter is optional, if it is specified, the call that it represents must belong to the hLine line device. The call state of hCall is device specific.

lpParams

A pointer to a memory area that is used to hold a parameter block. The format of this parameter block specifies device specific, and TAPI passes its contents to or from the service provider.

dwSize

The size in bytes of the parameter block area.

lineDevSpecificFeature

The lineDevSpecificFeature function enables service providers to provide access to features that other TAPI functions do not offer. The extensions are device-specific and the applications must be able to read the extensions to take advantage of these extensions. When used with the Cisco TSP, lineDevSpecificFeature can be used to enable/disable Do-Not-Disturb feature on a device.

Function Details

```
LONG lineDevSpecificFeature(
    HLINE hLine,
    DWORD dwFeature,
    LPVOID lpParams,
    DWORD dwSize
);
```

Parameters

hLine

A handle to a line device. This parameter is required.

dwFeature

Feature to invoke on the line device. This parameter uses the PHONEBUTTONFUNCTION_ TAPI constants. When used with the Cisco TSP, the only value that is considered valid is PHONEBUTTONFUNCTION_DONOTDISTURB (0x0000001A).

lpParams

A pointer to a memory area used to hold a parameter block. The format of this parameter block is device-specific and TAPI passes its contents to or from the service provider.

dwSize

The size in bytes of the parameter block area.

Return Values

Returns a positive request identifier if the function is completed asynchronously or a negative number if an error occurs. The dwParam2 parameter of the corresponding LINE_REPLY message is zero if the function succeeds or it is a negative number if an error occurs.

Possible return values follow:

- LINEERR_INVALFEATURE
- LINEERR_OPERATIONUNAVAIL
- LINEERR_INVALLINEHANDLE

- LINEERR_OPERATIONFAILED
- LINEERR_INVALPOINTER
- LINEERR_RESOURCEUNAVAIL
- LINEERR_NOMEM
- LINEERR_UNINITIALIZED.

Error Codes

The following new error can be returned by Cisco TSP for Do-Not-Disturb feature:

LINERR_ALREADY_IN_REQUESTED_STATE 0xC0000009

lineDial

The lineDial function dials the specified number on the specified call.

The application can use this function to enter a FAC or CMC. The FAC or CMC can be entered one digit at a time or multiple digits at a time. The application may also enter both the FAC and CMC if required in one lineDial() request as long as the FAC and CMC are separated by a "#" character. If sending both a FAC and CMC in one lineDial() request, Cisco recommends that you terminate the lpszDestAddress with a "#" character to avoid waiting for the T.302 interdigit time-out.

You cannot use this function to enter a dial string along with a FAC and/or a CMC. You must enter the FAC and/or CMC in a separate lineDial request.

Function Details

```
LONG lineDial(
  HCALL hCall,
  LPCSTR lpszDestAddress,
  DWORD dwCountryCode
);
```

Parameters

hCall

A handle to the call on which a number is to be dialed. Ensure the application is an owner of the call. The call state of hCall can be any state except idle and disconnected.

lpszDestAddress

The destination to be dialed by using the standard dial number format.

dwCountryCode

The country code of the destination. The implementation uses this code to select the call progress protocols for the destination address. If a value of 0 is specified, the default call progress protocol is used.

lineDrop

The lineDrop function drops or disconnects the specified call. The application can specify user-user information to be transmitted as part of the call disconnect.

Function Details

```
LONG lineDrop(
  HCALL hCall,
  LPCSTR lpsUserUserInfo,
  DWORD dwSize
);
```

Parameters

hCall

A handle to the call to be dropped. Ensure the application is an owner of the call. The call state of hCall can be any state except an Idle state.

lpsUserUserInfo

A pointer to a string that contains user-user information to be sent to the remote party as part of the call disconnect. You can leave this pointer NULL if no user-user information is to be sent. User-user information is sent only if it is supported by the underlying network. The protocol discriminator field for the user-user information, if required, should appear as the first byte of the buffer that is pointed to by lpsUserUserInfo and must be accounted for in dwSize.

Note The Cisco Unified TSP does not support user-user information.

dwSize

The size in bytes of the user-user information in lpsUserUserInfo. If lpsUserUserInfo is NULL, no user-user information gets sent to the calling party, and dwSize is ignored.

lineForward

The lineForward function forwards calls that are destined for the specified address on the specified line, according to the specified forwarding instructions. When an originating address (dwAddressID) is forwarded, the switch deflects the specified incoming calls for that address to the other number. This function provides a combination of forward all feature. This API allows calls to be forwarded unconditionally to a forwarded destination. This function can also cancel forwarding that is currently in effect.

To indicate that the forward is set/reset, upon completion of lineForward, TAPI fires LINEADDRESSSTATE events that indicate the change in the line forward status.

Change forward destination with a call to lineForward without canceling the current forwarding set on that line.

Note lineForward implementation of Cisco Unified TSP allows user to set up only one type for forward as dwForwardMode = UNCOND. The lpLineForwardList data structure accepts LINEFORWARD entry with dwForwardMode = UNCOND.

Function Details

```
LONG lineForward(
  HLINE hLine,
  DWORD bAllAddresses,
  DWORD dwAddressID,
  LPLINEFORWARDLIST const lpForwardList,
  DWORD dwNumRingsNoAnswer,
  LPHCALL lphConsultCall,
  LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hLine

A handle to the line device.

bAllAddresses

Specifies whether all originating addresses on the line or just the one that is specified gets forwarded. If TRUE, all addresses on the line get forwarded, and dwAddressID is ignored; if FALSE, only the address that is specified as dwAddressID gets forwarded.

dwAddressID

The address of the specified line whose incoming calls are to be forwarded. This parameter gets ignored if bAllAddresses is TRUE.

Note If bAllAddresses is FALSE, dwAddressID must equal 0.

lpForwardList

A pointer to a variably sized data structure that describes the specific forwarding instructions of type LINEFORWARDLIST.

Note To cancel the forwarding that currently is in effect, ensure lpForwardList Parameter is set to NULL.

dwNumRingsNoAnswer

The number of rings before a call is considered a no answer. If dwNumRingsNoAnswer is out of range, the actual value gets set to the nearest value in the allowable range.

Note This parameter is not used because this version of Cisco Unified TSP does not support call forward no answer.

lphConsultCall

A pointer to an HCALL location. In some telephony environments, this location is loaded with a handle to a consultation call that is used to consult the party to which the call is being forwarded, and the application becomes the initial sole owner of this call. This pointer must be valid even in environments where call forwarding does not require a consultation call. This handle is set to NULL if no consultation call is created.

Note This parameter is also ignored because a consult call is not created for setting up lineForward.

lpCallParams

A pointer to a structure of type LINECALLPARAMS. This pointer gets ignored unless lineForward requires the establishment of a call to the forwarding destination (and lphConsultCall is returned; in which case, lpCallParams is optional). If NULL, default call parameters get used. Otherwise, the specified call parameters get used for establishing hConsultCall.

Note This parameter must be NULL for this version of Cisco Unified TSP because we do not create a consult call.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALLINEHANDLE
- LINEERR_NOMEM
- LINEERR_INVALADDRESSID
- LINEERR_OPERATIONUNAVAIL
- LINEERR_INVALADDRESS
- LINEERR_OPERATIONFAILED
- LINEERR_INVALCOUNTRYCODE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALPOINTER
- LINEERR_STRUCTURETOOSMALL
- LINEERR_INVALPARAM
- LINEERR_UNINITIALIZED

Note For lpForwardList[0].dwForwardMode other than UNCOND, lineForward returns LINEERR_OPERATIONUNAVAIL. For lpForwardList.dwNumEntries more than 1, lineForward returns LINEERR_INVALPARAM

lineGenerateDigits

The lineGenerateDigits function initiates the generation of the specified digits on the specified call as out-of-band tones by using the specified signaling mode.

Note The Cisco Unified TSP supports neither invoking this function with a NULL value for lpszDigits to abort a digit generation that is currently in progress nor invoking lineGenerateDigits while digit generation is in progress. Cisco Unified IP Phones pass DTMF digits out of band. This means that the tone is not injected into the audio stream (in-band) but is sent as a message in the control stream. The phone on the far end then injects the tone into the audio stream to present it to the user. CTI port devices do not inject DTMF tones. Also, be aware that some gateways will not inject DTMF tones into the audio stream on the way out of the LAN.

Function Details

```
LONG lineGenerateDigits(
  HCALL hCall,
  DWORD dwDigitMode,
  LPCSTR lpszDigits,
  DWORD dwDuration
);
```

Parameters

hCall

A handle to the call. The application must be an owner of the call. Call state of hCall can be any state.

dwDigitMode

The format to be used for signaling these digits. The dwDigitMode can have only a single flag set. This parameter uses the following LINEDIGITMODE_ constant:

- LINEDIGITMODE_DTMF - Uses DTMF tones for digit signaling. Valid digits for DTMF mode include ‘0’ - ‘9’, ‘*’, ‘#’.

lpszDigits

Valid characters for DTMF mode in the Cisco Unified TSP include ‘0’ through ‘9’, ‘*’, and ‘#’.

dwDuration

Duration in milliseconds during which the tone should be sustained.

Note Cisco Unified TSP does not support dwDuration.

lineGenerateTone

The lineGenerateTone function generates the specified tone over the specified call.

Note The Cisco Unified TSP supports neither invoking this function with a 0 value for dwToneMode to abort a tone generation that is currently in progress nor invoking lineGenerateTone while tone generation is in progress. Cisco Unified IP Phones pass tones out of band. This means that the tone is not injected into the audio stream (in-band) but is sent as a message in the control stream. The phone on the far end then injects the tone into the audio stream to present it to the user. Also, be aware that some gateways will not inject tones into the audio stream on the way out of the LAN.

Function Details

```
LONG lineGenerateTone(
  HCALL hCall,
  DWORD dwToneMode,
  DWORD dwDuration,
  DWORD dwNumTones,
  LPLINEGENERATETONE const lpTones
);
```

Parameters

hCall

A handle to the call on which a tone is to be generated. The application must be an owner of the call. The call state of hCall can be any state.

dwToneMode

Defines the tone to be generated. Tones can be either standard or custom tones. A custom tone comprises a set of arbitrary frequencies. A small number of standard tones are predefined. The duration of the tone gets specified with dwDuration for both standard and custom tones. The dwToneMode parameter can have only one bit set. If no bits are set (the value 0 is passed), tone generation gets canceled.

This parameter uses the following LINETONEMODE_ constant:

- LINETONEMODE_BEEP - The tone is a beep, as used to announce the beginning of a recording. The service provider defines the exact beep tone.

dwDuration

Duration in milliseconds during which the tone should be sustained.

Note Cisco Unified TSP does not support dwDuration.

dwNumTones

The number of entries in the lpTones array. This parameter is ignored if dwToneMode ≠ CUSTOM.

lpTones

A pointer to a LINEGENERATETONE array that specifies the components of the tone. This parameter gets ignored for non-custom tones. If lpTones is a multifrequency tone, the various tones play simultaneously.

lineGetAddressCaps

The lineGetAddressCaps function queries the specified address on the specified line device to determine its telephony capabilities.

Function Details

```
LONG lineGetAddressCaps(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAddressID,
  DWORD dwAPIVersion,
  DWORD dwExtVersion,
  LPLINEADDRESSCAPS lpAddressCaps
);
```

Parameters

hLineApp

The handle by which the application is registered with TAPI.

dwDeviceID

The line device that contains the address to be queried. Only one address gets supported per line, so dwAddressID must be zero.

dwAddressID

The address on the given line device whose capabilities are to be queried.

dwAPIVersion

The version number, obtained by lineNegotiateAPIVersion, of the API that is to be used. The high-order word contains the major version number; the low-order word contains the minor version number.

dwExtVersion

The version number of the extensions to be used. This number can be left zero if no device-specific extensions are to be used. Otherwise, the high-order word contains the major version number and the low-order word contains the minor version number.

lpAddressCaps

A pointer to a variably sized structure of type LINEADDRESSCAPS. Upon successful completion of the request, this structure gets filled with address capabilities information. Prior to calling lineGetAddressCaps, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

lineGetAddressID

The lineGetAddressID function returns the address identifier that is associated with an address in a different format on the specified line.

Function Details

```
LONG lineGetAddressID(
  HLINE hLine,
  LPDWORD lpdwAddressID,
  DWORD dwAddressMode,
  LPCSTR lpsAddress,
  DWORD dwSize
);
```

Parameters

hLine

A handle to the open line device.

lpdwAddressID

A pointer to a DWORD-sized memory location that returns the address identifier.

dwAddressMode

The address mode of the address that is contained in lpsAddress. The dwAddressMode parameter can have only a single flag set. This parameter uses the following LINEADDRESSMODE_ constant:

- LINEADDRESSMODE_DIALABLEADDR - The address is specified by its dialable address. The lpsAddress parameter represents the dialable address or canonical address format.

lpsAddress

A pointer to a data structure that holds the address that is assigned to the specified line device. dwAddressMode determines the format of the address. Because the only valid value equals LINEADDRESSMODE_DIALABLEADDR, lpsAddress uses the common dialable number format and is NULL-terminated.

dwSize

The size of the address that is contained in lpsAddress.

lineGetAddressStatus

The lineGetAddressStatus function allows an application to query the specified address for its current status.

Function Details

```
LONG lineGetAddressStatus(
  HLINE hLine,
  DWORD dwAddressID,
  LPLINEADDRESSSTATUS lpAddressStatus
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

An address on the given open line device. This parameter specifies the address to be queried.

lpAddressStatus

A pointer to a variably sized data structure of type LINEADDRESSSTATUS. Prior to calling lineGetAddressStatus, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

lineGetCallInfo

The lineGetCallInfo function enables an application to obtain fixed information about the specified call.

Function Details

```
LONG lineGetCallInfo(
  HCALL hCall,
  LPLINECALLINFO lpCallInfo
);
```

Parameters

hCall

A handle to the call to be queried. The call state of hCall can be any state.

lpCallInfo

A pointer to a variably sized data structure of type LINECALLINFO. Upon successful completion of the request, call-related information fills this structure. Prior to calling lineGetCallInfo, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

lineGetCallStatus

The lineGetCallStatus function returns the current status of the specified call.

Function Details

```
LONG lineGetCallStatus(
  HCALL hCall,
  LPLINECALLSTATUS lpCallStatus
);
```

Parameters

hCall

A handle to the call to be queried. The call state of hCall can be any state.

lpCallStatus

A pointer to a variably sized data structure of type LINECALLSTATUS. Upon successful completion of the request, call status information fills this structure. Prior to calling lineGetCallStatus, the application should set the dwTotalSize member of this structure to indicate the amount of memory available to TAPI for returning information.

lineGetConfRelatedCalls

The lineGetConfRelatedCalls function returns a list of call handles that are part of the same conference call as the specified call. The specified call represents either a conference call or a participant call in a conference call. New handles get generated for those calls for which the application does not already have handles, and the application receives monitor privilege to those calls.

Function Details

```
LONG WINAPI lineGetConfRelatedCalls(
  HCALL hCall,
  LPLINECALLLIST lpCallList
);
```

Parameters

hCall

A handle to a call. This represents either a conference call or a participant call in a conference call. For a conference parent call, the call state of hCall can be any state. For a conference participant call, it must be in the conferenced state.

lpCallList

A pointer to a variably sized data structure of type LINECALLLIST. Upon successful completion of the request, call handles to all calls in the conference call return in this structure. The first call in the list represents the conference call, the other calls represent the participant calls. The application receives monitor privilege to those calls for which it does not already have handles; the privileges to calls in the list for which the application already has handles remains unchanged. Prior to calling lineGetConfRelatedCalls, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

Return Values

Returns zero if request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALCALLHANDLE
- LINEERR_OPERATIONFAILED
- LINEERR_NOCONFERENCE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALPOINTER
- LINEERR_STRUCTURETOOSMALL
- LINEERR_NOMEM
- LINEERR_UNINITIALIZED

lineGetDevCaps

The lineGetDevCaps function queries a specified line device to determine its telephony capabilities. The returned information applies for all addresses on the line device.

Function Details

```
LONG lineGetDevCaps(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  DWORD dwExtVersion,
  LPLINEDEVCAPS lpLineDevCaps
);
```

Parameters

hLineApp

The handle by which the application is registered with TAPI.

dwDeviceID

The line device to be queried.

dwAPIVersion

The version number, obtained by lineNegotiateAPIVersion, of the API to be used. The high-order word contains the major version number; the low-order word contains the minor version number.

dwExtVersion

The version number, obtained by lineNegotiateExtVersion, of the extensions to be used. It can be zero if no device-specific extensions are to be used. Otherwise, the high-order word contains the major version number; the low-order word contains the minor version number.

lpLineDevCaps

A pointer to a variably sized structure of type LINEDEVCAPS. Upon successful completion of the request, this structure gets filled with line device capabilities information. Prior to calling lineGetDevCaps, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

lineGetID

The lineGetID function returns a device identifier for the specified device class that is associated with the selected line, address, or call.

Function Details

```
LONG lineGetID(
  HLINE hLine,
  DWORD dwAddressID,
  HCALL hCall,
  DWORD dwSelect,
  LPVARSTRING lpDeviceID,
  LPCSTR lpszDeviceClass
);
```

Parameters

hLine

A handle to an open line device.

dwAddressID

An address on the given open line device.

hCall

A handle to a call.

dwSelect

Specifies whether the requested device identifier is associated with the line, address or a single call. The dwSelect parameter can only have a single flag set. This parameter uses the following LINECALLSELECT_ constants:

- LINECALLSELECT_LINE Selects the specified line device. The hLine parameter must be a valid line handle; hCall and dwAddressID are ignored.
- LINECALLSELECT_ADDRESS Selects the specified address on the line. Both hLine and dwAddressID must be valid; hCall is ignored.
- LINECALLSELECT_CALL Selects the specified call. hCall must be valid; hLine and dwAddressID are both ignored.

lpDeviceID

A pointer to a memory location of type VARSTRING, where the device identifier is returned. Upon successful completion of the request, the device identifier fills this location. The format of the returned information depends on the method that the device class API uses for naming devices. Before calling lineGetID, the application must set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

lpszDeviceClass

A pointer to a NULL-terminated ASCII string that specifies the device class of the device whose identifier is requested. Device classes include wave/in, wave/out and tapi/line.

Valid device class strings are those that are used in the SYSTEM.INI section to identify device classes.

lineGetLineDevStatus

The lineGetLineDevStatus function enables an application to query the specified open line device for its current status.

Function Details

```
LONG lineGetLineDevStatus(
  HLINE hLine,
  LPLINEDEVSTATUS lpLineDevStatus
);
```

Parameters

hLine

A handle to the open line device to be queried.

lpLineDevStatus

A pointer to a variably sized data structure of type LINEDEVSTATUS. Upon successful completion of the request, the device status of the line fills this structure. Prior to calling lineGetLineDevStatus, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

lineGetMessage

The lineGetMessage function returns the next TAPI message that is queued for delivery to an application that is using the Event Handle notification mechanism (see lineInitializeEx, page 5-33 for more information).

Function Details

```
LONG WINAPI lineGetMessage(
  HLINEAPP hLineApp,
  LPLINEMESSAGE lpMessage,
  DWORD dwTimeout
);
```

Parameters

hLineApp

The handle that lineInitializeEx returns. Ensure that the application has set the LINEINITIALIZEEXOPTION_USEEVENT option in the dwOptions member of the LINEINITIALIZEEXPARAMS structure.

lpMessage

A pointer to a LINEMESSAGE structure. Upon successful return from this function, the structure contains the next message that had been queued for delivery to the application.

dwTimeout

The time-out interval, in milliseconds. The function returns if the interval elapses, even if no message can be returned. If dwTimeout is zero, the function checks for a queued message and returns immediately. If dwTimeout is INFINITE, the function time-out interval never elapses.

Return Values

Returns zero if the request succeeds or returns a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALAPPHANDLE
- LINEERR_OPERATIONFAILED
- LINEERR_INVALPOINTER
- LINEERR_NOMEM

lineGetNewCalls

The lineGetNewCalls function returns call handles to calls on a specified line or address for which the application currently does not have handles. The application receives monitor privilege for these calls.

An application can use lineGetNewCalls to obtain handles to calls for which it currently has no handles. The application can select the calls for which handles are to be returned by basing this selection on scope (calls on a specified line, or calls on a specified address). For example, an application can request call handles to all calls on a given address for which it currently has no handle.

Function Details

```
LONG WINAPI lineGetNewCalls(
  HLINE hLine,
  DWORD dwAddressID,
  DWORD dwSelect,
  LPLINECALLLIST lpCallList
);
```

Parameters

hLine

A handle to an open line device.

dwAddressID

An address on the given open line device. An address identifier permanently associates with an address; the identifier remains constant across operating system upgrades.

dwSelect

The selection of calls that are requested. This parameter uses one and only one of the LINECALLSELECT_ Constants.

lpCallList

A pointer to a variably sized data structure of type LINECALLLIST. Upon successful completion of the request, call handles to all selected calls get returned in this structure. Prior to calling lineGetNewCalls, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALADDRESSID
- LINEERR_OPERATIONFAILED
- LINEERR_INVALCALLSELECT
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALLINEHANDLE
- LINEERR_STRUCTURETOOSMALL
- LINEERR_INVALPOINTER

- LINEERR_UNINITIALIZED
- LINEERR_NOMEM

lineGetNumRings

The lineGetNumRings function determines the number of rings that an incoming call on the given address should ring before the call is answered.

Function Details

```
LONG WINAPI lineGetNumRings(
  HLINE hLine,
  DWORD dwAddressID,
  LPDWORD lpdwNumRings
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

An address on the line device. An address identifier permanently associates with an address; the identifier remains constant across operating system upgrades.

lpdwNumRings

The number of rings that is the minimum of all current lineSetNumRings requests.

Return Values

Returns zero if request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALADDRESSID
- LINEERR_OPERATIONFAILED
- LINEERR_INVALLINEHANDLE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALPOINTER
- LINEERR_UNINITIALIZED
- LINEERR_NOMEM

lineGetProviderList

The lineGetProviderList function returns a list of service providers that are currently installed in the telephony system.

Function Details

```
LONG WINAPI lineGetProviderList(
  DWORD dwAPIVersion,
  LPLINEPROVIDERLIST lpProviderList
);
```

Parameters

dwAPIVersion

The highest version of TAPI that the application supports (not necessarily the value that lineNegotiateAPIVersion negotiates on some particular line device).

lpProviderList

A pointer to a memory location where TAPI can return a LINEPROVIDERLIST structure. Prior to calling lineGetProviderList, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

Return Values

Returns zero if request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INCOMPATIBLEAPIVERSION
- LINEERR_NOMEM
- LINEERR_INIFILECORRUPT
- LINEERR_OPERATIONFAILED
- LINEERR_INVALPOINTER
- LINEERR_STRUCTURETOOSMALL

lineGetRequest

The lineGetRequest function retrieves the next by-proxy request for the specified request mode.

Function Details

```
LONG WINAPI lineGetRequest(
  HLINEAPP hLineApp,
  DWORD dwRequestMode,
  LPVOID lpRequestBuffer
);
```

Parameters

hLineApp

The application's usage handle for the line portion of TAPI.

dwRequestMode

The type of request that is to be obtained. dwRequestMode can have only one bit set. This parameter uses one and only one of the LINEREQUESTMODE_ Constants.

lpRequestBuffer

A pointer to a memory buffer where the parameters of the request are to be placed. The size of the buffer and the interpretation of the information that is placed in the buffer depends on the request mode. The application-allocated buffer provides sufficient size to hold the request. If dwRequestMode is LINEREQUESTMODE_MAKECALL, interpret the content of the request buffer by using the LINEREQMAKECALL structure. If dwRequestMode is LINEREQUESTMODE_MEDIACALL, interpret the content of the request buffer by using the LINEREQMEDIACALL structure.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALAPPHANDLE
- LINEERR_NOTREGISTERED
- LINEERR_INVALPOINTER
- LINEERR_OPERATIONFAILED
- LINEERR_INVALREQUESTMODE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_NOMEM
- LINEERR_UNINITIALIZED
- LINEERR_NOREQUEST

lineGetStatusMessages

The lineGetStatusMessages function enables an application to query the notification messages that the application receives for events related to status changes for the specified line or any of its addresses.

Function Details

```
LONG WINAPI lineGetStatusMessages(
  HLINE hLine,
  LPDWORD lpdwLineStates,
  LPDWORD lpdwAddressStates
);
```

Parameters

hLine

Handle to the line device.

lpdwLineStates

A bit array that identifies the line device status changes for which a message is to be sent to the application. If a flag is TRUE, that message is enabled; if FALSE, it is disabled. This parameter uses one or more LINEDEVSTATE_ Constants.

lpdwAddressStates

A bit array that identifies for which address status changes a message is to be sent to the application. If a flag is TRUE, that message is enabled; if FALSE, disabled. This parameter uses one or more LINEADDRESSSTATE_ Constants.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALLINEHANDLE
- LINEERR_OPERATIONFAILED
- LINEERR_INVALPOINTER
- LINEERR_RESOURCEUNAVAIL
- LINEERR_NOMEM
- LINEERR_UNINITIALIZED

lineGetTranslateCaps

The lineGetTranslateCaps function returns address translation capabilities.

Function Details

```
LONG WINAPI lineGetTranslateCaps(
  HLINEAPP hLineApp,
  DWORD dwAPIVersion,
  LPLINETRANSLATECAPS lpTranslateCaps
);
```

Parameters

hLineApp

The application handle that lineInitializeEx returns. If an application has not yet called the lineInitializeEx function, it can set the hLineApp parameter to NULL.

dwAPIVersion

The highest version of TAPI that the application supports (not necessarily the value that lineNegotiateAPIVersion negotiates on some particular line device).

lpTranslateCaps

A pointer to a location to which a LINETRANSLATECAPS structure is loaded. Prior to calling lineGetTranslateCaps, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INCOMPATIBLEAPIVERSION
- LINEERR_NOMEM
- LINEERR_INIFILECORRUPT
- LINEERR_OPERATIONFAILED
- LINEERR_INVALAPPHANDLE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALPOINTER
- LINEERR_STRUCTURETOOSMALL
- LINEERR_NODRIVER.

lineHandoff

The lineHandoff function gives ownership of the specified call to another application. Specify the application either directly by its file name or indirectly as the highest priority application that handles calls of the specified media mode.

Function Details

```
LONG WINAPI lineHandoff(
  HCALL hCall,
  LPCSTR lpszFileName,
  DWORD dwMediaMode
);
```

Parameters

hCall

A handle to the call to be handed off. The application must be an owner of the call. The call state of hCall can be any state.

lpszFileName

A pointer to a null-terminated string. If this pointer parameter is non-NULL, it contains the file name of the application that is the target of the handoff. If NULL, the handoff target represents the highest priority application that has opened the line for owner privilege for the specified media mode. A valid file name does not include the path of the file.

dwMediaMode

The media mode that is used to identify the target for the indirect handoff. The dwMediaMode parameter indirectly identifies the target application that is to receive ownership of the call. This parameter gets ignored if lpszFileName is not NULL. This parameter uses one and only one of the LINEMEDIAMODE_ Constants.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALCALLHANDLE

- LINEERR_OPERATIONFAILED
- LINEERR_INVALMEDIAMODE
- LINEERR_TARGETNOTFOUND
- LINEERR_INVALPOINTER
- LINEERR_TARGETSELF
- LINEERR_NOMEM
- LINEERR_UNINITIALIZED
- LINEERR_NOTOWNER

lineHold

The lineHold function places the specified call on hold.

Function Details

```
LONG lineHold(
  HCALL hCall
);
```

Parameter

hCall

A handle to the call that is to be placed on hold. Ensure that the application is an owner of the call and the call state of hCall is connected.

lineInitialize

Although the lineInitialize function is obsolete, tapi.dll and tapi32.dll continue to export it for backward compatibility with applications that are using API versions 1.3 and 1.4.

Function Details

```
LONG WINAPI lineInitialize(
  LPHLINEAPP lphLineApp,
  HINSTANCE hInstance,
  LINECALLBACK lpfnCallback,
  LPCSTR lpszAppName,
  LPDWORD lpdwNumDevs
);
```

Parameters

lphLineApp

A pointer to a location that is filled with the application's usage handle for TAPI.

hInstance

The instance handle of the client application or DLL.

lpfnCallback

The address of a callback function that is invoked to determine status and events on the line device, addresses, or calls. For more information, see lineCallbackFunc.

lpszAppName

A pointer to a null-terminated text string that contains only displayable characters. If this parameter is not NULL, it contains an application-supplied name for the application. The LINECALLINFO structure provides this name to indicate, in a user-friendly way, which application originated, originally accepted, or answered the call. This information can prove useful for call logging purposes. If lpszAppName is NULL, the application's file name gets used instead.

lpdwNumDevs

A pointer to a DWORD-sized location. Upon successful completion of this request, this location gets filled with the number of line devices that is available to the application.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALAPPNAME
- LINEERR_OPERATIONFAILED
- LINEERR_INIFILECORRUPT
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALPOINTER
- LINEERR_REINIT
- LINEERR_NODRIVER
- LINEERR_NODEVICE
- LINEERR_NOMEM
- LINEERR_NOMULTIPLEINSTANCE.

lineInitializeEx

The lineInitializeEx function initializes the use of TAPI by the application for the subsequent use of the line abstraction. It registers the specified notification mechanism of the application and returns the number of line devices that are available. A line device represents any device that provides an implementation for the line-prefixed functions in the telephony API.

Function Details

```
LONG lineInitializeEx(
  LPHLINEAPP lphLineApp,
  HINSTANCE hInstance,
  LINECALLBACK lpfnCallback,
  LPCSTR lpszFriendlyAppName,
  LPDWORD lpdwNumDevs,
  LPDWORD lpdwAPIVersion,
```

```
  LPLINEINITIALIZEEXPARAMS lpLineInitializeExParams
);
```

Parameters

lphLineApp

A pointer to a location that is filled with the TAPI usage handle for the application.

hInstance

The instance handle of the client application or DLL. The application or DLL can pass NULL for this parameter, in which case TAPI uses the module handle of the root executable of the process (for purposes of identifying call handoff targets and media mode priorities).

lpfnCallback

The address of a callback function that is invoked to determine status and events on the line device, addresses, or calls, when the application is using the "hidden window" method of event notification. This parameter gets ignored and should be set to NULL when the application chooses to use the "event handle" or "completion port" event notification mechanisms.

lpszFriendlyAppName

A pointer to a NULL-terminated ASCII string that contains only standard ASCII characters. If this parameter is not NULL, it contains an application-supplied name for the application. The LINECALLINFO structure provides this name to indicate, in a user-friendly way, which application originated, originally accepted, or answered the call. This information can prove useful for call-logging purposes. If lpszFriendlyAppName is NULL, the module filename of the application gets used instead (as returned by the Windows API GetModuleFileName).

lpdwNumDevs

A pointer to a DWORD-sized location. Upon successful completion of this request, this location gets filled with the number of line devices that are available to the application.

lpdwAPIVersion

A pointer to a DWORD-sized location. The application must initialize this DWORD, before calling this function, to the highest API version that it is designed to support (for example, the same value that it would pass into dwAPIHighVersion parameter of lineNegotiateAPIVersion). Make sure that artificially high values are not used; ensure that the value is set to 0x00020000. TAPI translates any newer messages or structures into values or formats that the application supports. Upon successful completion of this request, this location is filled with the highest API version that TAPI supports, which allows the application to adapt to being installed on a system with an older TAPI version.

lpLineInitializeExParams

A pointer to a structure of type LINEINITIALIZEEXPARAMS that contains additional parameters that are used to establish the association between the application and TAPI (specifically, the selected event notification mechanism of the application and associated parameters).

lineMakeCall

The lineMakeCall function places a call on the specified line to the specified destination address. Optionally, you can specify call parameters if anything but default call setup parameters are requested.

Function Details

```
LONG lineMakeCall(
HLINE hLine,
LPHCALL lphCall,
LPCSTR lpszDestAddress,
DWORD dwCountryCode,
LPLINECALLPARAMS const lpCallParams
);
typedef struct LineParams {
DWORD FeaturePriority;
}LINE_PARAMS;
```

Parameters

hLine

A handle to the open line device on which a call is to be originated.

lphCall

A pointer to an HCALL handle. The handle is only valid after the application receives LINE_REPLY message that indicates that the lineMakeCall function successfully completed. Use this handle to identify the call when you invoke other telephony operations on the call. The application initially acts as the sole owner of this call. This handle registers as void if the reply message returns an error (synchronously or asynchronously).

lpszDestAddress

A pointer to the destination address. This parameter follows the standard dialable number format. This pointer can be NULL for non-dialed addresses or when all dialing is performed by using lineDial. In the latter case, lineMakeCall allocates an available call appearance that would typically remain in the dial tone state until dialing begins.

dwCountryCode

The country code of the called party. If a value of 0 is specified, the implementation uses a default.

lpCallParams

The dwNoAnswerTimeout attribute of the lpCallParams field is checked and if specified as non-zero, automatically disconnects a call if not answered after the specified time.

Note Beginning with Cisco Unified Communications Manager Release 7.0(1), feature priority is introduced for DoNotDisturb-Reject feature. Feature priority can be specified in DevSpecific part of CallParams. as typedef struct LineParams {DWORD FeaturePriority; } LINE_PARAMS;.

lineMonitorDigits

The lineMonitorDigits function enables and disables the unbuffered detection of digits that are received on the call. Each time that a digit of the specified digit mode is detected, a message gets sent to the application to indicate which digit has been detected.

Function Details

```
LONG lineMonitorDigits(
```

```
  HCALL hCall,
  DWORD dwDigitModes
);
```

Parameters

hCall

A handle to the call on which digits are to be detected. The call state of hCall can be any state except idle or disconnected.

dwDigitModes

The digit mode or modes that are to be monitored. If dwDigitModes is zero, the system cancels digit monitoring. This parameter which can have multiple flags set, uses the following LINEDIGITMODE_ constant:

LINEDIGITMODE_DTMF - Detect digits as DTMF tones. Valid digits for DTMF include ‘0’ through ‘9’, ‘*’, and ‘#’.

lineMonitorTones

The lineMonitorTones function enables and disables the detection of inband tones on the call. Each time that a specified tone is detected, a message gets sent to the application.

Function Details

```
LONG lineMonitorTones(
  HCALL hCall,
  LPLINEMONITORTONE const lpToneList,
  DWORD dwNumEntries
);
```

Parameters

hCall

A handle to the call on which tones are to be detected. The call state of hCall can be any state except idle.

lpToneList

A list of tones to be monitored, of type LINEMONITORTONE. Each tone in this list has an application-defined tag field that is used to identify individual tones in the list to report a tone detection. Calling this operation with either NULL for lpToneList or with another tone list cancels or changes tone monitoring in progress.

dwNumEntries

The number of entries in lpToneList. This parameter gets ignored if lpToneList is NULL.

lineNegotiateAPIVersion

The lineNegotiateAPIVersion function allows an application to negotiate an API version to use. The Cisco Unified TSP supports TAPI 2.0 and 2.1.

Function Details

```
LONG lineNegotiateAPIVersion(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPILowVersion,
  DWORD dwAPIHighVersion,
  LPDWORD lpdwAPIVersion,
  LPLINEEXTENSIONID lpExtensionID
);
```

Parameters

hLineApp

The handle by which the application is registered with TAPI.

dwDeviceID

The line device to be queried.

dwAPILowVersion

The least recent API version with which the application is compliant. The high-order word specifies the major version number; the low-order word specifies the minor version number.

dwAPIHighVersion

The most recent API version with which the application is compliant. The high-order word specifies the major version number; the low-order word specifies the minor version number.

lpdwAPIVersion

A pointer to a DWORD-sized location that contains the API version number that was negotiated. If negotiation succeeds, this number falls in the range between dwAPILowVersion and dwAPIHighVersion.

lpExtensionID

A pointer to a structure of type LINEEXTENSIONID. If the service provider for the specified dwDeviceID supports provider-specific extensions, upon a successful negotiation, this structure gets filled with the extension identifier of these extensions. This structure contains all zeros if the line provides no extensions. An application can ignore the returned parameter if it does not use extensions.

The Cisco Unified TSP extensionID specifies 0x8EBD6A50, 0x138011d2, 0x905B0060, 0xB03DD275.

lineNegotiateExtVersion

The lineNegotiateExtVersion function allows an application to negotiate an extension version to use with the specified line device. Do not call this operation if the application does not support extensions.

Function Details

```
LONG lineNegotiateExtVersion(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  DWORD dwExtLowVersion,
  DWORD dwExtHighVersion,
```

```
  LPDWORD lpdwExtVersion
);
```

Parameters

hLineApp

The handle by which the application is registered with TAPI.

dwDeviceID

The line device to be queried.

dwAPIVersion

The API version number that was negotiated for the specified line device by using lineNegotiateAPIVersion.

dwExtLowVersion

The least recent extension version of the extension identifier that lineNegotiateAPIVersion returns and with which the application is compliant. The high-order word specifies the major version number; the low-order word specifies the minor version number.

dwExtHighVersion

The most recent extension version of the extension identifier that lineNegotiateAPIVersion returns and with which the application is compliant. The high-order word specifies the major version number; the low-order word specifies the minor version number.

lpdwExtVersion

A pointer to a DWORD-sized location that contains the extension version number that was negotiated. If negotiation succeeds, this number falls between dwExtLowVersion and dwExtHighVersion.

lineOpen

The lineOpen function opens the line device that its device identifier specifies and returns a line handle for the corresponding opened line device. Subsequent operations on the line device use this line handle.

Function Details

```
LONG lineOpen(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  LPHLINE lphLine,
  DWORD dwAPIVersion,
  DWORD dwExtVersion,
  DWORD dwCallbackInstance,
  DWORD dwPrivileges,
  DWORD dwMediaModes,
  LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hLineApp

The handle by which the application is registered with TAPI.

dwDeviceID

Identifies the line device to be opened. It can either be a valid device identifier or the value LINEMAPPER

Note The Cisco Unified TSP does not support LINEMAPPER at this time.

lphLine

A pointer to an HLINE handle that is then loaded with the handle that represents the opened line device. Use this handle to identify the device when you are invoking other functions on the open line device.

dwAPIVersion

The API version number under which the application and Telephony API operate. Obtain this number with lineNegotiateAPIVersion.

dwExtVersion

The extension version number under which the application and the service provider operate. This number remains zero if the application does not use any extensions. Obtain this number with lineNegotiateExtVersion.

dwCallbackInstance

User-instance data that is passed back to the application with each message that is associated with this line or with addresses or calls on this line. The Telephony API does not interpret this parameter.

dwPrivileges

The privilege that the application wants for the calls for which it is notified. This parameter can be a combination of the LINECALLPRIVILEGE_ constants. For applications that are using TAPI version 2.0 or later, values for this parameter can also be combined with the LINEOPENOPTION_ constants:

- LINECALLPRIVILEGE_NONE - The application can make only outgoing calls.
- LINECALLPRIVILEGE_MONITOR - The application can monitor only incoming and outgoing calls.
- LINECALLPRIVILEGE_OWNER - The application can own only incoming calls of the types that are specified in dwMediaModes.
- LINECALLPRIVILEGE_MONITOR + LINECALLPRIVILEGE_OWNER - The application can own only incoming calls of the types that are specified in dwMediaModes, but if the application does not represent an owner of a call, it acts as a monitor.
- Other flag combinations return the LINEERR_INVALPRIVSELECT error.

dwMediaModes

The media mode or modes of interest to the application. Use this parameter to register the application as a potential target for incoming call and call handoff for the specified media mode. This parameter proves meaningful only if the bit LINECALLPRIVILEGE_OWNER in dwPrivileges is set (and ignored if it is not).

This parameter uses the following LINEMEDIAMODE_ constant:

- LINEMEDIAMODE_INTERACTIVEVOICE - The application can handle calls of the interactive voice media type; that is, it manages voice calls with the user on this end of the call. Use this parameter for third-party call control of physical phones and CTI port and CTI route point devices that other applications opened.
- LINEMEDIAMODE_AUTOMATEDVOICE - Voice energy exists on the call. An automated application locally handles the voice. This represents first-party call control and is used with CTI port and CTI route point devices.

lpCallParams

The dwNoAnswerTimeout attribute of the lpCallParams field is checked and if it is non-zero, automatically disconnects a call if it is not answered after the specified time.

linePark

The linePark function parks the specified call according to the specified park mode.

Function Details

```
LONG WINAPI linePark(
    HCALL hCall,
    DWORD dwParkMode,
    LPCSTR lpszDirAddress,
    LPVARSTRING lpNonDirAddress
);
```

Parameters

hCall

Handle to the call to be parked. The application must act as an owner of the call. The call state of hcall must be connected.

dwParkMode

Park mode with which the call is parked. This parameter can have only a single flag set and uses one of the LINEPARKMODE_Constants.

Note Ensure that LINEPARKMODE_Constants is set to LINEPARKMODE_NONDIRECTED.

lpszDirAddress

Pointer to a null-terminated string that indicates the address where the call is to be parked when directed park is used. The address specifies in dialable number format. This parameter gets ignored for nondirected park.

Note This parameter gets ignored.

lpNonDirAddress

Pointer to a structure of type VARSTRING. For nondirected park, the address where the call is parked gets returned in this structure. This parameter gets ignored for directed park. Within the VARSTRING structure, ensure that dwStringFormat is set to STRINGFORMAT_ASCII (an ASCII string buffer that contains a null-terminated string), and the terminating NULL must be accounted for in the dwStringSize. Before calling linePark, the application must set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

linePrepareAddToConference

The linePrepareAddToConference function prepares an existing conference call for the addition of another party.

If LINEERR_INVALLINESTATE is returned, that means that the line is currently not in a state in which this operation can be performed. The dwLineFeatures member includes a list of currently valid operations (of the type LINEFEATURE) in the LINEDEVSTATUS structure. (Calling lineGetLineDevStatus updates the information in LINEDEVSTATUS.)

Obtain a conference call handle with lineSetupConference or with lineCompleteTransfer that is resolved as a three-way conference call. The linePrepareAddToConference function typically places the existing conference call in the onHoldPendingConference state and creates a consultation call that can be added later to the existing conference call with lineAddToConference.

You can cancel the consultation call by using lineDrop. You may also be able to swap an application between the consultation call and the held conference call with lineSwapHold.

Function Details

```
LONG WINAPI linePrepareAddToConference(
  HCALL hConfCall,
  LPHCALL lphConsultCall,
  LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hConfCall

A handle to a conference call. The application must act as an owner of this call. Ensure that the call state of hConfCall is connected.

lphConsultCall

A pointer to an HCALL handle. This location then gets loaded with a handle that identifies the consultation call to be added. Initially, the application serves as the sole owner of this call.

lpCallParams

A pointer to call parameters that gets used when the consultation call is established. You can set this parameter to NULL if no special call setup parameters are desired.

Return Values

Returns a positive request identifier if the function completes asynchronously, or a negative number if an error occurs. The dwParam2 parameter of the corresponding LINE_REPLY message specifies zero if the function succeeds, or it is a negative number if an error occurs.

Possible return values follow:

- LINEERR_BEARERMODEUNAVAIL

- LINEERR_INVALPOINTER
- LINEERR_CALLUNAVAIL
- LINEERR_INVALRATE
- LINEERR_CONFERENCEFULL
- LINEERR_NOMEM
- LINEERR_INUSE
- LINEERR_NOTOWNER
- LINEERR_INVALADDRESSMODE
- LINEERR_OPERATIONUNAVAIL
- LINEERR_INVALBEARERMODE
- LINEERR_OPERATIONFAILED
- LINEERR_INVALCALLPARAMS
- LINEERR_RATEUNAVAIL
- LINEERR_INVALCALLSTATE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALCONFCALLHANDLE
- LINEERR_STRUCTURETOOSMALL
- LINEERR_INVALLINESTATE
- LINEERR_USERUSERINFOTOOBIG
- LINEERR_INVALMEDIAMODE
- LINEERR_UNINITIALIZED

lineRedirect

The lineRedirect function redirects the specified offered or accepted call to the specified destination address.

Note If the application tries to redirect a call to an address that requires a FAC, CMC, or both, the lineRedirect function returns an error. If a FAC is required, the TSP returns the message LINEERR_FACREQUIRED. If a CMC is required, the TSP returns the message LINEERR_CMCREQUIRED. If both a FAC and a CMC are required, the TSP returns the message LINEERR_FACANDCMCREQUIRED. An application that wants to redirect a call to an address that requires a FAC, CMC, or both, should use the lineDevSpecific RedirectFACCMC function.

Function Details

```
LONG lineRedirect(
  HCALL hCall,
  LPCSTR lpszDestAddress,
  DWORD dwCountryCode
);
```

Parameters

hCall

A handle to the call to be redirected. The application must act as an owner of the call. The call state of hCall must be offering, accepted, or connected.

Note The Cisco Unified TSP supports redirecting of calls in the connected call state.

lpszDestAddress

A pointer to the destination address. This follows the standard dialable number format.

dwCountryCode

The country code of the party to which the call is redirected. If a value of 0 is specified, the implementation uses a default.

lineRegisterRequestRecipient

The lineRegisterRequestRecipient function registers the invoking application as a recipient of requests for the specified request mode.

Function Details

```
LONG WINAPI lineRegisterRequestRecipient(
  HLINEAPP hLineApp,
  DWORD dwRegistrationInstance,
  DWORD dwRequestMode,
  DWORD bEnable
);
```

Parameters

hLineApp

The application's usage handle for the line portion of TAPI.

dwRegistrationInstance

An application-specific DWORD that is passed back as a parameter of the LINE_REQUEST message. This message notifies the application that a request is pending. This parameter gets ignored if bEnable is set to zero. TAPI examines this parameter only for registration, not for deregistration. The dwRegistrationInstance value that is used while deregistering need not match the dwRegistrationInstance that is used while registering for a request mode.

dwRequestMode

The type or types of request for which the application registers. This parameter uses one or more LINEREQUESTMODE_ Constants.

bEnable

If TRUE, the application registers the specified request modes; if FALSE, the application deregisters for the specified request modes.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALAPPHANDLE
- LINEERR_OPERATIONFAILED
- LINEERR_INVALREQUESTMODE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_NOMEM
- LINEERR_UNINITIALIZED

lineRemoveFromConference

The lineRemoveFromConference function removes a specified call from the conference call to which it currently belongs. The remaining calls in the conference call are unaffected.

Function Details

```
LONG WINAPI lineRemoveFromConference(
  HCALL hCall
);
```

Parameters

hCall

Handle to the call that is to be removed from the conference. The application must be an owner of this call. The call state of hCall must be conference.

Return Values

Returns a positive request identifier if the function is completed asynchronously, or a negative number if an error occurs. The dwParam2 parameter of the corresponding LINE_REPLY message is zero if the function succeeds or it is a negative number if an error occurs. The following table shows the return values for this function:

Value	Description
LINEERR_INVALCALLHANDLE	The handle to the call that is to be removed is invalid.
LINEERR_OPERATIONUNAVAIL	The operation is unavailable.
LINEERR_INVALCALLSTATE	The call state is something other than conferenced.
LINEERR_OPERATIONFAILED	The operation failed.
LINEERR_NOMEM	Not enough memory.
LINEERR_RESOURCEUNAVAIL	The resources are unavailable.

Value	Description
LINEERR_NOTOWNER	The application is not the owner of this call.
LINEERR_UNINITIALIZED	A parameter is uninitialized.

lineRemoveProvider

The lineRemoveProvider function removes an existing telephony service provider from the system.

Function Details

```
LONG WINAPI lineRemoveProvider(
  DWORD dwPermanentProviderID,
  HWND hwndOwner
);
```

Parameters

dwPermanentProviderID

The permanent provider identifier of the service provider that is to be removed.

hwndOwner

A handle to a window to which any dialog boxes that need to be displayed as part of the removal process (for example, a confirmation dialog box by the service provider's TSPI_providerRemove function) would be attached. The parameter can be a NULL value to indicate that any window that is created during the function should have no owner window.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INIFILECORRUPT
- LINEERR_NOMEM
- LINEERR_INVALPARAM
- LINEERR_OPERATIONFAILED

lineSetAppPriority

The lineSetAppPriority function allows an application to set its priority in the handoff priority list for a particular media type or Assisted Telephony request mode or to remove itself from the priority list.

Function Details

```
LONG WINAPI lineSetAppPriority(
  LPCSTR lpszAppFilename,
  DWORD dwMediaMode,
  LPLINEEXTENSIONID lpExtensionID,
  DWORD dwRequestMode,
  LPCSTR lpszExtensionName,
```

```
  DWORD dwPriority
);
```

Parameters

lpszAppFilename

A pointer to a string that contains the application executable module filename (without directory information). In TAPI version 2.0 or later, the parameter can specify a filename in either long or 8.3 filename format.

dwMediaMode

The media type for which the priority of the application is to be set. The value can be one LINEMEDIAMODE_ Constant; only a single bit may be on. Use the value zero to set the application priority for Assisted Telephony requests.

lpExtensionID

A pointer to a structure of type LINEEXTENSIONID. This parameter gets ignored.

dwRequestMode

If the dwMediaMode parameter is zero, this parameter specifies the Assisted Telephony request mode for which priority is to be set. It must be either LINEREQUESTMODE_MAKECALL or LINEREQUESTMODE_MEDIACALL. This parameter gets ignored if dwMediaMode is nonzero.

lpszExtensionName

This parameter gets ignored.

dwPriority

The new priority for the application. If the value 0 is passed, the application gets removed from the priority list for the specified media or request mode (if it was already not present, no error gets generated). If the value 1 is passed, the application gets inserted as the highest priority application for the media or request mode (and removed from a lower-priority position, if it was already in the list). Any other value generates an error.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INIFILECORRUPT
- LINEERR_INVALREQUESTMODE
- LINEERR_INVALAPPNAME
- LINEERR_NOMEM
- LINEERR_INVALMEDIAMODE
- LINEERR_OPERATIONFAILED
- LINEERR_INVALPARAM
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALPOINTER

lineSetCallPrivilege

The lineSetCallPrivilege function sets the application privilege to the specified privilege.

Function Details

```
LONG WINAPI lineSetCallPrivilege(
  HCALL hCall,
  DWORD dwCallPrivilege
);
```

Parameters

hCall

A handle to the call whose privilege is to be set. The call state of hCall can be any state.

dwCallPrivilege

The privilege that the application can have for the specified call. This parameter uses one and only one LINECALLPRIVILEGE_ Constant.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALCALLHANDLE
- LINEERR_OPERATIONFAILED
- LINEERR_INVALCALLSTATE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALCALLPRIVILEGE
- LINEERR_UNINITIALIZED
- LINEERR_NOMEM

lineSetNumRings

The lineSetNumRings function sets the number of rings that must occur before an incoming call is answered. Use this function to implement a toll saver-style function. It allows multiple, independent applications to each register the number of rings. The function lineGetNumRings returns the minimum number of rings that are requested. The application that answers incoming calls can use it to determine the number of rings that it should wait before answering the call.

Function Details

```
LONG WINAPI lineSetNumRings(
  HLINE hLine,
  DWORD dwAddressID,
  DWORD dwNumRings
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

An address on the line device. An address identifier permanently associates with an address; the identifier remains constant across operating system upgrades.

dwNumRings

The number of rings before a call should be answered to honor the toll saver requests from all applications.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_INVALLINEHANDLE
- LINEERR_OPERATIONFAILED
- LINEERR_INVALADDRESSID
- LINEERR_RESOURCEUNAVAIL
- LINEERR_NOMEM
- LINEERR_UNINITIALIZED

lineSetStatusMessages

The lineSetStatusMessages function enables an application to specify the notification messages to receive for events that are related to status changes for the specified line or any of its addresses.

Function Details

```
LONG lineSetStatusMessages(
  HLINE hLine,
  DWORD dwLineStates,
  DWORD dwAddressStates
);
```

Parameters

hLine

A handle to the line device.

dwLineStates

A bit array that identifies for which line-device status changes a message is to be sent to the application. This parameter uses the following LINEDEVSTATE_ constants:

- LINEDEVSTATE_OTHER - Device-status items other than the following ones changed. The application should check the current device status to determine which items changed.
- LINEDEVSTATE_RINGING - The switch tells the line to alert the user. Service providers notify applications on each ring cycle by sending LINE_LINEDEVSTATE messages that contain this constant. For example, in the United States, service providers send a message with this constant every 6 seconds.
- LINEDEVSTATE_NUMCALLS - The number of calls on the line device changed.
- LINEDEVSTATE_REINIT - Items changed in the configuration of line devices. To become aware of these changes (as with the appearance of new line devices) the application should reinitialize its use of TAPI. New lineInitialize, lineInitializeEx, and lineOpen requests get denied until applications have shut down their usage of TAPI. The hDevice parameter of the LINE_LINEDEVSTATE message remains NULL for this state change as it applies to any lines in the system. Because of the critical nature of LINEDEVSTATE_REINIT, such messages cannot be masked, so the setting of this bit is ignored, and the messages always get delivered to the application.
- LINEDEVSTATE_REMOVED - Indicates that the service provider is removing the device from the system (most likely through user action, through a control panel or similar utility). Normally, a LINE_CLOSE message on the device immediately follows LINE_LINEDEVSTATE message with this value. Subsequent attempts to access the device prior to TAPI being reinitialized result in LINEERR_NODEVICE being returned to the application. If a service provider sends a LINE_LINEDEVSTATE message that contains this value to TAPI, TAPI passes it along to applications that have negotiated TAPI version 1.4 or later; applications that negotiate a previous TAPI version do not receive any notification.

dwAddressStates

A bit array that identifies for which address status changes a message is to be sent to the application. This parameter uses the following LINEADDRESSSTATE_ constant:

- LINEADDRESSSTATE_NUMCALLS - The number of calls on the address changed. This change results from events such as a new incoming call, an outgoing call on the address, or a call changing its hold status.

lineSetTollList

The lineSetTollList function manipulates the toll list.

Function Details

```
LONG WINAPI lineSetTollList(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  LPCSTR lpszAddressIn,
```

```
  DWORD dwTollListOption
);
```

Parameters

hLineApp

The application handle that lineInitializeEx returns. If an application has not yet called the lineInitializeEx function, it can set the hLineApp parameter to NULL.

dwDeviceID

The device identifier for the line device upon which the call is intended to be dialed, so variations in dialing procedures on different lines can be applied to the translation process.

lpszAddressIn

A pointer to a null-terminated string that contains the address from which the prefix information is to be extracted for processing. Ensure that this parameter is not NULL, and also ensure that it is in the canonical address format.

dwTollListOption

The toll list operation to be performed. This parameter uses one and only one of the LINETOLLLISTOPTION_ Constants.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_BADDEVICEID
- LINEERR_NODRIVER
- LINEERR_INVALAPPHANDLE
- LINEERR_NOMEM
- LINEERR_INVALADDRESS
- LINEERR_OPERATIONFAILED
- LINEERR_INVALPARAM
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INIFILECORRUPT
- LINEERR_UNINITIALIZED
- LINEERR_INVALLOCATION

lineSetupConference

The lineSetupConference function initiates a conference for an existing two-party call that the hCall parameter specifies. A conference call and consult call are established, and the handles return to the application. Use the consult call to dial the third party and the conference call replaces the initial two-party call. The application can also specify the destination address of the consult call that will allow the PBX to dial the call for the application.

Function Details

```
LONG lineSetupConference (
HCALL hCall,
HLINE hLine,
LPHCALL lphConfCall,
LPHCALL lphConsultCall,
DWORD dwNumParties,
LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hCall

The handle of the existing two-party call. Ensure that the application is the owner of the call.

hLine

The line on which the initial two-party call was made. This parameter is not used because hCall must be set.

lphConfCall

A pointer to the conference call handle. The service provider allocates this call and returns the handle to the application.

lphConsultCall

A pointer to the consult call. If the application does not specify the destination address in the call parameters, it should use this call handle to dial the consult call. If the destination address is specified, the consult call will be made using this handle.

dwNumParties

The number of parties in the conference call. Currently the Cisco Unified TAPI Service Provider supports a three-party conference call.

lpCallParams

The call parameters that are used to set up the consult call. The application can specify the destination address if it wants the consult call to be dialed for it in the conference setup.

lineSetupTransfer

The lineSetupTransfer function initiates a transfer of the call that the hCall parameter specifies. It establishes a consultation call, lphConsultCall, on which the party can be dialed that can become the destination of the transfer. The application acquires owner privilege to the lphConsultCall parameter.

Function Details

```
LONG lineSetupTransfer(
  HCALL hCall,
  LPHCALL lphConsultCall,
  LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hCall

The handle of the call to be transferred. Ensure that the application is an owner of the call and ensure that the call state of hCall is connected.

lphConsultCall

A pointer to an hCall handle. This location is then loaded with a handle that identifies the temporary consultation call. When setting up a call for transfer, a consultation call automatically gets allocated that enables lineDial to dial the address that is associated with the new transfer destination of the call. The originating party can carry on a conversation over this consultation call prior to completing the transfer. The call state of hConsultCall does not apply.

This transfer procedure may not be valid for some line devices. The application may need to ignore the new consultation call and remove the hold on an existing held call (using lineUnhold) to identify the destination of the transfer. On switches that support cross-address call transfer, the consultation call can exist on a different address than the call that is to be transferred. It may also be necessary to set up the consultation call as an entirely new call, by lineMakeCall, to the destination of the transfer. The address capabilities of the call specifies which forms of transfer are available.

lpCallParams

The dwNoAnswerTimeout attribute of the lpCallParams field is checked and, if is non-zero, used to automatically disconnect a call if it is not answered after the specified time.

lineShutdown

The lineShutdown function shuts down the usage of the line abstraction of the API.

Function Details

```
LONG lineShutdown(
  HLINEAPP hLineApp
);
```

Parameters

hLineApp

The usage handle of the application for the line API.

lineTranslateAddress

The lineTranslateAddress function translates the specified address into another format.

Function Details

```
LONG WINAPI lineTranslateAddress(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  LPCSTR lpszAddressIn,
  DWORD dwCard,
  DWORD dwTranslateOptions,
```

```
  LPLINETRANSLATEOUTPUT lpTranslateOutput
);
```

Parameters

hLineApp

The application handle that lineInitializeEx returns. If a TAPI 2.0 application has not yet called the lineInitializeEx function, it can set the hLineApp parameter to NULL. TAPI 1.4 applications must still call lineInitialize first.

dwDeviceID

The device identifier for the line device upon which the call is intended to be dialed, so variations in dialing procedures on different lines can be applied to the translation process.

dwAPIVersion

Indicates the highest version of TAPI that the application supports (not necessarily the value that is negotiated by lineNegotiateAPIVersion on some particular line device).

lpszAddressIn

Pointer to a null-terminated string that contains the address from which the information is to be extracted for translation. This parameter must either use the canonical address format or an arbitrary string of dialable digits (non-canonical). This parameter must not be NULL. If the AddressIn contains a subaddress or name field, or additional addresses separated from the first address by CR and LF characters, only the first address gets translated.

dwCard

The credit card to be used for dialing. This parameter proves valid only if the CARDOVERRIDE bit is set in dwTranslateOptions. This parameter specifies the permanent identifier of a Card entry in the [Cards] section in the registry (as obtained from lineTranslateCaps) that should be used instead of the PreferredCardID that is specified in the definition of the CurrentLocation. It does not cause the PreferredCardID parameter of the current Location entry in the registry to be modified; the override applies only to the current translation operation. This parameter gets ignored if the CARDOVERRIDE bit is not set in dwTranslateOptions.

dwTranslateOptions

The associated operations to be performed prior to the translation of the address into a dialable string. This parameter uses one of the LINETRANSLATEOPTION_ Constants.

Note If you have set the LINETRANSLATEOPTION_CANCELCALLWAITING bit, also set the LINECALLPARAMFLAGS_SECURE bit in the dwCallParamFlags member of the LINECALLPARAMS structure (passed in to lineMakeCall through the lpCallParams parameter). This action prevents the line device from using dialable digits to suppress call interrupts.

lpTranslateOutput

A pointer to an application-allocated memory area to contain the output of the translation operation, of type LINETRANSLATEOUTPUT. Prior to calling lineTranslateAddress, the application should set the dwTotalSize member of this structure to indicate the amount of memory that is available to TAPI for returning information.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_BADDEVICEID
- LINEERR_INVALPOINTER
- LINEERR_INCOMPATIBLEAPIVERSION
- LINEERR_NODRIVER
- LINEERR_INIFILECORRUPT
- LINEERR_NOMEM
- LINEERR_INVALADDRESS
- LINEERR_OPERATIONFAILED
- LINEERR_INVALAPPHANDLE
- LINEERR_RESOURCEUNAVAIL
- LINEERR_INVALCARD
- LINEERR_STRUCTURETOOSMALL
- LINEERR_INVALPARAM

lineTranslateDialog

The lineTranslateDialog function displays an application-modal dialog box that allows the user to change the current location of a phone number that is about to be dialed, adjust location and calling card parameters, and see the effect.

Function Details

```
LONG WINAPI lineTranslateDialog(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  HWND hwndOwner,
  LPCSTR lpszAddressIn
);
```

Parameters

hLineApp

The application handle that lineInitializeEx returns. If an application has not yet called the lineInitializeEx function, it can set the hLineApp parameter to NULL.

dwDeviceID

The device identifier for the line device upon which the call is intended to be dialed, so variations in dialing procedures on different lines can be applied to the translation process.

dwAPIVersion

Indicates the highest version of TAPI that the application supports (not necessarily the value that lineNegotiateAPIVersion negotiates on the line device that dwDeviceID indicates).

hwndOwner

A handle to a window to which the dialog box is to be attached. Can be a NULL value to indicate that any window that is created during the function should have no owner window.

lpszAddressIn

A pointer to a null-terminated string that contains a phone number that is used, in the lower portion of the dialog box, to show the effect of the user's changes on the location parameters. Ensure that the number is in canonical format; if noncanonical, the phone number portion of the dialog box does not display. You can leave this pointer NULL, in which case the phone number portion of the dialog box does not display. If the lpszAddressIn parameter contains a subaddress or name field, or additional addresses separated from the first address by CR and LF characters, only the first address gets used in the dialog box.

Return Values

Returns zero if request succeeds or a negative number if an error occurs. Possible return values follow:

- LINEERR_BADDEVICEID
- LINEERR_INVALPARAM
- LINEERR_INCOMPATIBLEAPIVERSION
- LINEERR_INVALPOINTER
- LINEERR_INIFILECORRUPT
- LINEERR_NODRIVER
- LINEERR_INUSE
- LINEERR_NOMEM
- LINEERR_INVALADDRESS
- LINEERR_INVALAPPHANDLE
- LINEERR_OPERATIONFAILED

lineUnhold

The lineUnhold function retrieves the specified held call.

Function Details

```
LONG lineUnhold(
  HCALL hCall
);
```

Parameters

hCall

The handle to the call to be retrieved. The application must be an owner of this call. The call state of hCall must be onHold, onHoldPendingTransfer, or onHoldPendingConference.

lineUnpark

The lineUnpark function retrieves the call that is parked at the specified address and returns a call handle for it.

Function Details

```
LONG WINAPI lineUnpark(
    HLINE hLine,
    DWORD dwAddressID,
    LPHCALL lphCall,
    LPCSTR lpszDestAddress
);
```

Parameters

hLine

Handle to the open line device on which a call is to be unparked.

dwAddressID

Address on hLine at which the unpark is to be originated. An address identifier permanently associates with an address; the identifier remains constant across operating system upgrades.

lphCall

Pointer to the location of type HCALL where the handle to the unparked call is returned. This handle is unrelated to any other handle that previously may have been associated with the retrieved call, such as the handle that might have been associated with the call when it was originally parked. The application acts as the initial sole owner of this call.

lpszDestAddress

Pointer to a null-terminated character buffer that contains the address where the call is parked. The address displays in standard dialable address format.

TAPI Line Messages

This section describes the line messages that the Cisco Unified TSP supports. These messages notify the application of asynchronous events such as a new call arriving in the Cisco Unified Communications Manager. The messages get sent to the application by the method that the application specifies in lineInitializeEx

.

Table 5-2 TAPI Line Messages

TAPI Line Messages
LINE_ADDRESSSTATE
LINE_APPNEWCALL
LINE_CALLDEVSPECIFIC
LINE_CALLINFO
LINE_CALLSTATE

Table 5-2 TAPI Line Messages (continued)

TAPI Line Messages
LINE_CLOSE
LINE_CREATE
LINE_DEVSPECIFIC
LINE_DEVSPECIFICFEATURE
LINE_GATHERDIGITS
LINE_GENERATE
LINE_LINEDEVSTATE
LINE_MONITORDIGITS
LINE_MONITORTONE
LINE_REMOVE
LINE_REPLY
LINE_REQUEST

LINE_ADDRESSSTATE

The LINE_ADDRESSSTATE message gets sent when the status of an address changes on a line that is currently open by the application. The application can invoke lineGetAddressStatus to determine the current status of the address.

Function Details

```
LINE_ADDRESSSTATE
dwDevice = (DWORD) hLine;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) idAddress;
dwParam2 = (DWORD) AddressState;
dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

A handle to the line device.

dwCallbackInstance

The callback instance supplied when the line is opened.

dwParam1

The address identifier of the address that changed status.

dwParam2

The address state that changed. Can be a combination of these values:

LINEADDRESSSTATE_OTHER

Address-status items other than those that are in the following list changed. The application should check the current address status to determine which items changed.

LINEADDRESSSTATE_DEVSPECIFIC

The device-specific item of the address status changed.

LINEADDRESSSTATE_INUSEZERO

The address changed to idle (it is now in use by zero stations).

LINEADDRESSSTATE_INUSEONE

The address changed from idle or from being used by many bridged stations to being used by just one station.

LINEADDRESSSTATE_INUSEMANY

The monitored or bridged address changed from being used by one station to being used by more than one station.

LINEADDRESSSTATE_NUMCALLS

The number of calls on the address has changed. This change results from events such as a new inbound call, an outbound call on the address, or a call changing its hold status.

LINEADDRESSSTATE_FORWARD

The forwarding status of the address changed, including the number of rings for determining a no-answer condition. The application should check the address status to determine details about the current forwarding status of the address.

LINEADDRESSSTATE_TERMINALS

The terminal settings for the address changed.

LINEADDRESSSTATE_CAPSCHANGE

Indicates that due to configuration changes that the user made, or other circumstances, one or more of the members in the LINEADDRESSCAPS structure for the address changed. The application should use lineGetAddressCaps to read the updated structure. Applications that support API versions earlier than 1.4 receive a LINEDEVSTATE_REINIT message that requires them to shut down and reinitialize their connection to TAPI to obtain the updated information.

dwParam3 is not used.

LINE_APPNEWCALL

The LINE_APPNEWCALL message informs an application when a new call handle is spontaneously created on its behalf (other than through an API call from the application, in which case the handle would have been returned through a pointer parameter that passed into the function).

Function Details

```
LINE_APPNEWCALL
dwDevice = (DWORD) hLine;
dwCallbackInstance = (DWORD) dwInstanceData;
dwParam1 = (DWORD) dwAddressID;
dwParam2 = (DWORD) hCall;
dwParam3 = (DWORD) dwPrivilege;
```

Parameters

dwDevice

The handle of the application to the line device on which the call was created.

dwCallbackInstance

The callback instance that is supplied when the line belonging to the call is opened.

dwParam1

Identifier of the address on the line on which the call appears.

dwParam2

The handle of the application to the new call.

dwParam3

The privilege of the application to the new call (LINECALLPRIVILEGE_OWNER or LINECALLPRIVILEGE_MONITOR).

LINE_CALLDEVSPECIFIC

The TSPI LINE_CALLDEVSPECIFIC message is sent to notify TAPI about device-specific events that occur on a call. The meaning of the message and the interpretation of the dwParam1 through dwParam3 parameters are device specific.

Function Details

```
LINE_CALLDEVSPECIFIC
    htLine = (HTAPILINE) hLineDevice;
    htCall = (HTAPICALL) hCallDevice;
    dwMsg = (DWORD) LINE_CALLDEVSPECIFIC;
    dwParam1 = (DWORD) DeviceData1;
    dwParam2 = (DWORD) DeviceData2;
    dwParam3 = (DWORD) DeviceData3;
```

Parameters

htLine

The TAPI opaque object handle to the line device.

htCall

The TAPI opaque object handle to the call device.

dwMsg

The value LINE_CALLDEVSPECIFIC.

dwParam1

Device specific

dwParam2

Device specific

dwParam3

Device specific

LINE_CALLINFO

The TAPI LINE_CALLINFO message gets sent when the call information about the specified call has changed. The application can invoke lineGetCallInfo to determine the current call information.

Function Details

```
LINE_CALLINFO
hDevice = (DWORD) hCall;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) CallInfoState;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Parameters

hDevice

A handle to the call.

dwCallbackInstance

The callback instance that is supplied when the call's line is opened.

dwParam1

The call information item that changed. Can be one or more of the LINECALLINFOSTATE_ constants.

dwParam2 is not used.

dwParam3 is not used.

LINE_CALLSTATE

The LINE_CALLSTATE message gets sent when the status of the specified call changes. Typically, several such messages occur during the lifetime of a call. Applications get notified of new incoming calls with this message; the new call exists in the offering state. The application can use the lineGetCallStatus function to retrieve more detailed information about the current status of the call.

Function Details

```
LINE_CALLSTATE
dwDevice = (DWORD) hCall;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) CallState;
dwParam2 = (DWORD) CallStateDetail;
dwParam3 = (DWORD) CallPrivilege;
```

Parameters

dwDevice

A handle to the call.

dwCallbackInstance

The callback instance that is supplied when the line that belongs to this call is opened.

dwParam1

The new call state. Cisco Unified TSP supports only the following LINECALLSTATE_ values:

LINECALLSTATE_IDLE

The call remains idle; no call actually exists.

LINECALLSTATE_OFFERING

The call gets offered to the station, which signals the arrival of a new call. In some environments, a call in the offering state does not automatically alert the user. The switch that instructs the line to ring does alerts; it does not affect any call states.

LINECALLSTATE_ACCEPTED

The system offered the call and it has been accepted. This indicates to other (monitoring) applications that the current owner application claimed responsibility for answering the call. In ISDN, this also indicates that alerting to both parties started.

LINECALLSTATE_CONFERENCED

The call is a member of a conference call and is logically in the connected state.

LINECALLSTATE_DIALTONE

The call receives a dial tone from the switch, which means that the switch is ready to receive a dialed number.

LINECALLSTATE_DIALING

Destination address information (a phone number) is sent to the switch over the call. The lineGenerateDigits does not place the line into the dialing state.

LINECALLSTATE_RINGBACK

The call receives ringback from the called address. Ringback indicates that the call has reached the other station and is being alerted.

LINECALLSTATE_ONHOLDPENDCONF

The call currently remains on hold while it gets added to a conference.

LINECALLSTATE_CONNECTED

The call is established and the connection is made. Information can flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING

Dialing completes and the call proceeds through the switch or telephone network.

LINECALLSTATE_ONHOLD

The switch keeps the call on hold.

LINECALLSTATE_ONHOLDPENDTRANSFER

The call that is currently on hold awaits transfer to another number.

LINECALLSTATE_DISCONNECTED

The remote party disconnected from the call.

LINECALLSTATE_UNKNOWN

The state of the call is not known. This state may occur due to limitations of the call-progress detection implementation.

Cisco Unified TSP supports two new call states that indicate more information about the call state within the Cisco Unified Communications Manager setup. The standard TAPI call state is set to LINECALLSTATE_UNKNOWN and the following call states will be ORed with the unknown call state.

#define CLDSMT_CALL_PROGRESSING_STATE 0x0100000

The Progressing state indicates that the call is in progress over the network. The application must negotiate extension version 0x00050001 to receive this call state.

#define CLDSMT_CALL_WAITING_STATE 0x02000000

The waiting state indicates that the REFER request is in progress on Referrer's line and the application should not request any other function on this call. All the requests will result in LINEERR_INVALCALLSTATE. Application has to negotiate extension version 0x00070000 to receive this call state.

#define CLDSMT_CALL_WHISPER_STATE 0x03000000

The whisper state indicates that the Intercom call is connected in one-way audio mode. The Intercom originator cannot issue other function other that to drop the Intercom call. While at destination side, the system allows only Talkback and dropping call. All other requests result in LINEERR_OPERATIONUNAVAIL.

dwParam2

Call-state-dependent information.

- If dwParam1 is LINECALLSTATE_CONNECTED, dwParam2 contains details about the connected mode. This parameter uses the following LINECONNECTEDMODE_ constants:
  - LINECONNECTEDMODE_ACTIVE

    Call connects at the current station (the current station acts as a participant in the call).
  - LINECONNECTEDMODE_INACTIVE

    Call stays active at one or more other stations, but the current station does not participate in the call.

When a call is disconnected with cause code = DISCONNECTMODE_TEMPFAILURE and the lineState = LINEDEVSTATE_INSERVICE, applications must take care of dropping the call. If the application terminates media for a device, then it is also takes the responsibility to stop the RTP streams for the same call. Cisco Unified TSP will not provide Stop Transmission/Reception events to applications in this scenario. The behavior is exactly the same with IP phones. The user must hang up the disconnected - temp fail call on IP phone to stop the media. The application is also responsible for stopping the RTP streams in case the line goes out of service (LINEDEVSTATE_OUTOFSERVICE) and the call on a line is reported as IDLE.

Note If an application with negotiated extension version 0x00050001 or greater receives device-specific CLDSMT_CALL_PROGRESSING_STATE = 0x01000000 with LINECALLSTATE_UNKNOWN, the cause code is reported as the standard Q931 cause codes in dwParam2.

- If dwParam1 specifies LINECALLSTATE_DIALTONE, dwParam2 contains the details about the dial tone mode. This parameter uses the following LINEDIALTONEMODE_ constant:

  LINEDIALTONEMODE_UNAVAIL

    The dial tone mode is unavailable and cannot become known.

- If dwParam1 specifies LINECALLSTATE_OFFERING, dwParam2 contains details about the connected mode. This parameter uses the following LINEOFFERINGMODE_ constants:

  LINEOFFERINGMODE_ACTIVE

    The call alerts at the current station (accompanied by LINEDEVSTATE_RINGING messages) and, if an application is set up to automatically answer, it answers. For TAPI versions 1.4 and later, if the call state mode is ZERO, the application assumes that the value is active (which represents the situation on a non-bridged address).

Note The Cisco Unified TSP does not send LINEDEVSTATE_RINGING messages until the call is accepted and moves to the LINECALLSTATE_ACCEPTED state. IP_phones auto-accept calls. CTI ports and CTI route points do not auto-accept calls. Call the lineAccept() function to accept the call at these types of devices.

- If dwParam1 specifies LINECALLSTATE_DISCONNECTED, dwParam2 contains details about the disconnect mode. This parameter uses the following LINEDISCONNECTMODE_ constants:

  LINEDISCONNECTMODE_NORMAL

    This specifies a normal disconnect request by the remote party; call terminated normally.

  LINEDISCONNECTMODE_UNKNOWN

    The reason for the disconnect request remains unknown.

  LINEDISCONNECTMODE_REJECT

    The remote user rejected the call.

  LINEDISCONNECTMODE_BUSY

    The station that belongs to the remote user is busy.

  LINEDISCONNECTMODE_NOANSWER

    The station that belongs to the remote user does not answer.

  LINEDISCONNECTMODE_CONGESTION

    This message indicates that the network is congested.

  LINEDISCONNECTMODE_UNAVAIL

    The reason for the disconnect remains unavailable and cannot become known later.

  LINEDISCONNECTMODE_FACCMC

    Indicates that the FAC/CMC feature disconnected the call.

Note LINEDISCONNECTMODE_FACCMC is returned only if the extension version that is negotiated on the line is 0x00050000 (6.0(1)) or higher. If the negotiated extension version is not at least 0x00050000, TSP sets the disconnect mode to LINEDISCONNECTMODE_UNAVAIL.

dwParam3

If zero, this parameter indicates that no change in the privilege occurred for the call to this application.

If nonzero, this parameter specifies the privilege for the application to the call. This occurs in the following situations: (1) The first time that the application receives a handle to this call; (2) When the application is the target of a call hand-off (even if the application already was an owner of the call). This parameter uses the following LINECALLPRIVILEGE_ constants:

LINECALLPRIVILEGE_MONITOR

The application has monitor privilege.

LINECALLPRIVILEGE_OWNER

The application has owner privilege.

LINE_CLOSE

The LINE_CLOSE message gets sent when the specified line device has been forcibly closed. The line device handle or any call handles for calls on the line no longer remains valid after this message is sent.

Function Details

```
LINE_CLOSE
dwDevice = (DWORD) hLine;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) 0;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

A handle to the line device that was closed. This handle is no longer valid

dwCallbackInstance

The callback instance that is supplied when the line that belongs to this call is opened.

dwParam1 is not used.

dwParam2 is not used.

dwParam3 is not used.

LINE_CREATE

The LINE_CREATE message informs the application of the creation of a new line device.

Note CTI Manager cluster support, extension mobility, change notification, and user addition to the directory can generate LINE_CREATE events.

Function Details

```
LINE_CREATE
dwDevice = (DWORD) 0;
dwCallbackInstance = (DWORD) 0;
dwParam1 = (DWORD) idDevice;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Parameters

dwDevice is not used.

dwCallbackInstance is not used.

dwParam1

The dwDeviceID of the newly created device.

dwParam2 is not used.

dwParam3 is not used.

LINE_DEVSPECIFIC

The LINE_DEVSPECIFIC message notifies the application about device-specific events that occur on a line, address, or call. The meaning of the message and interpretation of the parameters are device specific.

Function Details

```
LINE_DEVSPECIFIC
dwDevice = (DWORD) hLineOrCall;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) DeviceSpecific1;
dwParam2 = (DWORD) DeviceSpecific2;
dwParam3 = (DWORD) DeviceSpecific3;
```

Parameters

dwDevice

This device-specific parameter specifies a handle to either a line device or call.

dwCallbackInstance

The callback instance that is supplied when the line opens.

dwParam1 is device specific

dwParam2 is device specific

dwParam3 is device specific

LINE_DEVSPECIFICFEATURE

This line message, added in Cisco Unified Communications Manager Release 6.0, enables a Do Not Disturb (DND) change notification event. Cisco TSP notifies applications by using the LINE_DEVSPECIFICFEATURE message about changes in the DND configuration or status. In order to receive change notifications an application needs to enable DEVSPECIFIC_DONOTDISTURB_CHANGED message flag by using lineDevSpecific SLDST_SET_STATUS_MESSAGES request.

LINE_DEVSPECIFICFEATURE message is sent to notify the application about device-specific events occurring on a line device. In case of a DND change notification, the message includes information about the type of change that occurred on a device and resulted feature status or configured option.

Function Details

```
dwDevice = (DWORD) hLine;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) PHONEBUTTONFUNCTION_DONOTDISTURB;
dwParam2 = (DWORD) typeOfChange;
dwParam3 = (DWORD) currentValue;

enum CiscoDoNotDisturbOption {
    DoNotDisturbOption_NONE      = 0,
    DoNotDisturbOption_RINGEROFF = 1,
    DoNotDisturbOption_REJECT    = 2
};

enum CiscoDoNotDisturbStatus {
    DoNotDisturbStatus_UNKNOWN  = 0,
    DoNotDisturbStatus_ENABLED  = 1,
    DoNotDisturbStatus_DISABLED = 2
};

enum CiscoDoNotDisturbNotification {
    DoNotDisturb_STATUS_CHANGED  = 1,
    DoNotDisturb_OPTION_CHANGED  = 2
};
```

Parameters

dwDevice

A handle to a line device.

dwCallbackInstance

The callback instance supplied when opening the line.

dwParam1

Always equal to PHONEBUTTONFUNCTION_DONOTDISTURB for the Do-Not-Disturb change notification

dwParam2

Indicates the type of change and can have one of the following enum values:

```
enum CiscoDoNotDisturbNotification {
    DoNotDisturb_STATUS_CHANGED  = 1,
    DoNotDisturb_OPTION_CHANGED  = 2
};
```

dwParam3

If the dwParm2 indicates status change (is equal to DoNotDisturb_STATUS_CHANGED) this parameter can have one of the following enum values:

```
enum CiscoDoNotDisturbStatus {
    DoNotDisturbStatus_UNKNOWN  = 0,
    DoNotDisturbStatus_ENABLED  = 1,
    DoNotDisturbStatus_DISABLED = 2
};
```

If the dwParm2 indicates option change (is equal to DoNotDisturb_OPTION_CHANGED) this parameter can have one of the following enum values:

```
enum CiscoDoNotDisturbOption {
    DoNotDisturbOption_NONE      = 0,
    DoNotDisturbOption_RINGEROFF = 1,
    DoNotDisturbOption_REJECT    = 2
};
```

LINE_GATHERDIGITS

The TAPI LINE_GATHERDIGITS message is sent when the current buffered digit-gathering request is terminated or canceled. You can examine the digit buffer after the application receives this message.

Function Details

```
LINE_GATHERDIGITS
    hDevice = (DWORD) hCall;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) GatherTermination;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

hDevice

A handle to the call.

dwCallbackInstance

The callback instance that is supplied when the line opens.

dwParam1

The reason why digit gathering terminated. This parameter must be one and only one of the LINEGATHERTERM_ constants.

dwParam2

Unused.

dwParam3

The tick count (number of milliseconds since Windows started) at which the digit gathering completes. For TAPI versions earlier than 2.0, this parameter is not used.

LINE_GENERATE

The TAPI LINE_GENERATE message notifies the application that the current digit or tone generation terminated. Only one such generation request can be in progress an a given call at any time. This message also gets sent when digit or tone generation is canceled.

Function Details

```
LINE_GENERATE
hDevice = (DWORD) hCall;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) GenerateTermination;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Parameters

hDevice

A handle to the call.

dwCallbackInstance

The callback instance that is supplied when the line opens.

dwParam1

The reason that digit or tone generation terminates. This parameter must be the only one of the LINEGENERATETERM_ constants.

dwParam2 is not used.

dwParam3

The tick count (number of milliseconds since Windows started) at which the digit or tone generation completes. For API versions earlier than 2.0, this parameter is not used.

LINE_LINEDEVSTATE

The TAPI LINE_LINEDEVSTATE message gets sent when the state of a line device changes. The application can invoke lineGetLineDevStatus to determine the new status of the line.

Function Details

```
LINE_LINEDEVSTATE
hDevice = (DWORD) hLine;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) DeviceState;
dwParam2 = (DWORD) DeviceStateDetail1;
dwParam3 = (DWORD) DeviceStateDetail2;
```

Parameters

hDevice

A handle to the line device. This parameter is NULL when dwParam1 is LINEDEVSTATE_REINIT.

dwCallbackInstance

The callback instance that is supplied when the line is opened. If the dwParam1 parameter is LINEDEVSTATE_REINIT, the dwCallbackInstance parameter is not valid and is set to zero.

dwParam1

The line device status item that changed. The parameter can be one or more of the LINEDEVSTATE_ constants.

dwParam2

The interpretation of this parameter depends on the value of dwParam1. If dwParam1 is LINEDEVSTATE_RINGING, dwParam2 contains the ring mode with which the switch instructs the line to ring. Valid ring modes include numbers in the range one to dwNumRingModes, where dwNumRingModes specifies a line device capability.

If dwParam1 is LINEDEVSTATE_REINIT, and TAPI issued the message as a result of translation of a new API message into a REINIT message, dwParam2 contains the dwMsg parameter of the original message (for example, LINE_CREATE or LINE_LINEDEVSTATE). If dwParam2 is zero, this indicates that the REINIT message represents a real REINIT message that requires the application to call lineShutdown at its earliest convenience.

dwParam3

The interpretation of this parameter depends on the value of dwParam1. If dwParam1 is LINEDEVSTATE_RINGING, dwParam3 contains the ring count for this ring event. The ring count starts at zero.

If dwParam1 is LINEDEVSTATE_REINIT, and TAPI issued the message as a result of translation of a new API message into a REINIT message, dwParam3 contains the dwParam1 parameter of the original message (for example, LINEDEVSTATE_TRANSLATECHANGE or some other LINEDEVSTATE_ value, if dwParam2 is LINE_LINEDEVSTATE, or the new device identifier, if dwParam2 is LINE_CREATE).

LINE_MONITORDIGITS

The LINE_MONITORDIGITS message gets sent when a digit is detected. The lineMonitorDigits function controls the sending of this message.

Function Details

```
LINE_MONITORDIGITS
dwDevice = (DWORD) hCall;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) Digit;
dwParam2 = (DWORD) DigitMode;
dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

A handle to the call.

dwCallbackInstance

The callback instance that is supplied when the line for this call is opened.

dwParam1

The low-order byte contains the last digit that is received in ASCII.

dwParam2

The digit mode that was detected. This parameter must be one and only one of the following LINEDIGITMODE_ constant:

- LINEDIGITMODE_DTMF - Detect digits as DTMF tones. Valid digits for DTMF include ‘0’ through ‘9’, ‘*’, and ‘#’.

dwParam3

The “tick count” (number of milliseconds after Windows started) at which the specified digit was detected. For API versions earlier than 2.0, this parameter is not used.

LINE_MONITORTONE

The LINE_MONITORTONE message gets sent when a tone is detected. The lineMonitorTones function controls the sending of this message.

Note Cisco Unified TSP supports only silent detection through LINE_MONITORTONE.

Function Details

```
LINE_MONITORTONE
dwDevice = (DWORD) hCall;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) dwAppSpecific;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) tick count;
```

Parameters

dwDevice

A handle to the call.

dwCallbackInstance

The callback instance supplied when the line opens for this call.

dwParam1

The application-specific dwAppSpecific member of the LINE_MONITORTONE structure for the tone that was detected.

dwParam2 is not used.

dwParam3

The “tick count” (number of milliseconds after Windows started) at which the specified digit was detected.

LINE_REMOVE

The LINE_REMOVE message informs an application of the removal (deletion from the system) of a line device. Generally, this parameter is not used for temporary removals, such as extraction of PCMCIA devices, but only for permanent removals in which, the service provider would no longer report the device, if TAPI were reinitialized.

Note CTI Manager cluster support, extension mobility, change notification, and user deletion from the directory can generate LINE_REMOVE events.

Function Details

```
LINE_REMOVE
dwDevice = (DWORD) 0;
dwCallbackInstance = (DWORD) 0;
dwParam1 = (DWORD) dwDeviceID;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Parameters

dwDevice is reserved. Set to zero.

dwCallbackInstance is reserved. Set to zero.

dwParam1

Identifier of the line device that was removed.

dwParam2 is reserved. Set to zero.

dwParam3 is reserved. Set to zero.

LINE_REPLY

The LINE_REPLY message reports the results of function calls that completed asynchronously.

Function Details

```
LINE_REPLY
dwDevice = (DWORD) 0;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) idRequest;
dwParam2 = (DWORD) Status;
dwParam3 = (DWORD) 0;
```

Parameters

dwDevice is not used.

dwCallbackInstance

Returns the callback instance for this application.

dwParam1

The request identifier for which this is the reply.

dwParam2

The success or error indication. The application should cast this parameter into a long integer:

- Zero indicates success.
- A negative number indicates an error.

dwParam3 is not used.

LINE_REQUEST

The TAPI LINE_REQUEST message reports the arrival of a new request from another application.

Function Details

```
LINE_REQUEST
hDevice = (DWORD) 0;
dwCallbackInstance = (DWORD) hRegistration;
dwParam1 = (DWORD) RequestMode;
dwParam2 = (DWORD) RequestModeDetail1;
dwParam3 = (DWORD) RequestModeDetail2;
```

Parameters

hDevice is not used.

dwCallbackInstance

The registration instance of the application that is specified on lineRegisterRequestRecipient.

dwParam1

The request mode of the newly pending request. This parameter uses the LINEREQUESTMODE_ constants.

dwParam2

If dwParam1 is set to LINEREQUESTMODE_DROP, dwParam2 contains the hWnd of the application that requests the drop. Otherwise, dwParam2 is not used.

dwParam3

If dwParam1 is set to LINEREQUESTMODE_DROP, the low-order word of dwParam3 contains the wRequestID as specified by the application requesting the drop. Otherwise, dwParam3 is not used.

TAPI Line Device Structures

Table 5-3 lists the TAPI line device structures that the Cisco Unified TSP supports. This section lists the possible values for the structure members as set by the TSP, and provides a cross reference to the functions that use them. If the value of a structure member is device, line, or call specific, the system notes the value for each condition.

Table 5-3 TAPI Line Device Structures

TAPI Line Device Structures
LINEADDRESSCAPS
LINEADDRESSSTATUS
LINEAPPINFO
LINECALLINFO
LINECALLLIST
LINECALLPARAMS
LINECALLSTATUS
LINECARDENTRY
LINECOUNTRYENTRY
LINECOUNTRYLIST
LINEDEVCAPS
LINEDEVSTATUS
LINEEXTENSIONID
LINEFORWARD
LINEFORWARDLIST
LINEGENERATETONE
LINEINITIALIZEEXPARAMS
LINELOCATIONENTRY
LINEMESSAGE
LINEMONITORTONE
LINEPROVIDERENTRY
LINEPROVIDERLIST
LINEREQMAKECALL
LINETRANSLATECAPS
LINETRANSLATEOUTPUT

LINEADDRESSCAPS

Members	Values
dwLineDeviceID	For All Devices: The device identifier of the line device with which this address is associated.
dwAddressSize dwAddressOffset	For All Devices: The size, in bytes, of the variably sized address field and the offset, in bytes, from the beginning of this data structure
dwDevSpecificSize dwDevSpecificOffset	For All Devices: 0

Members	Values
dwAddressSharing	For All Devices: 0
dwAddressStates	For All Devices (except Park DNs): LINEADDRESSSTATE_FORWARD
	For Park DNs: 0
dwCallInfoStates	For All Devices (except Park DNs): LINECALLINFOSTATE_CALLEDID LINECALLINFOSTATE_CALLERID LINECALLINFOSTATE_CALLID LINECALLINFOSTATE_CONNECTEDID LINECALLINFOSTATE_MEDIAMODE LINECALLINFOSTATE_MONITORMODES LINECALLINFOSTATE_NUMMONITORS LINECALLINFOSTATE_NUMOWNERDECR LINECALLINFOSTATE_NUMOWNERINCR LINECALLINFOSTATE_ORIGIN LINECALLINFOSTATE_REASON LINECALLINFOSTATE_REDIRECTINGID LINECALLINFOSTATE_REDIRECTIONID
	For Park DNs: LINECALLINFOSTATE_CALLEDID LINECALLINFOSTATE_CALLERID LINECALLINFOSTATE_CALLID LINECALLINFOSTATE_CONNECTEDID LINECALLINFOSTATE_NUMMONITORS LINECALLINFOSTATE_NUMOWNERDECR LINECALLINFOSTATE_NUMOWNERINCR LINECALLINFOSTATE_ORIGIN LINECALLINFOSTATE_REASON LINECALLINFOSTATE_REDIRECTINGID LINECALLINFOSTATE_REDIRECTIONID
dwCallerIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN LINECALLPARTYID_BLOCKED
dwCalledIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN
dwConnectedIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN LINECALLPARTYID_BLOCKED

Members	Values
dwRedirectionIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN LINECALLPARTYID_BLOCKED
dwRedirectingIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN
dwCallStates	For IP Phones and CTI Ports: LINECALLSTATE_ACCEPTED LINECALLSTATE_CONFERENCED LINECALLSTATE_CONNECTED LINECALLSTATE_DIALING LINECALLSTATE_DIALTONE LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_ONHOLD LINECALLSTATE_ONHOLDPENDCONF LINECALLSTATE_ONHOLDPENDTRANSFER LINECALLSTATE_PROCEEDING LINECALLSTATE_RINGBACK LINECALLSTATE_UNKNOWN
	For CTI Route Points (without media): LINECALLSTATE_ACCEPTED LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_UNKNOWN For CTI Route Points (with media): LINECALLSTATE_ACCEPTED LINECALLSTATE_CONNECTED LINECALLSTATE_DISCONNECTED LINECALLSTATE_ONHOLD LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_UNKNOWN
	For Park DNs: LINECALLSTATE_ACCEPTED LINECALLSTATE_CONFERENCED LINECALLSTATE_CONNECTED LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_ONHOLD LINECALLSTATE_UNKNOWN

Members	Values
dwDialToneModes	For IP Phones and CTI Ports: LINEDIALTONEMODE_UNAVAIL
	For CTI Route Points and Park DNs: 0
dwBusyModes	For All Devices: 0
dwSpecialInfo	For All Devices: 0
dwDisconnectModes	For All Devices: LINEDISCONNECTMODE_BADDADDRESS LINEDISCONNECTMODE_BUSY LINEDISCONNECTMODE_CONGESTION LINEDISCONNECTMODE_FORWARDED LINEDISCONNECTMODE_NOANSWER LINEDISCONNECTMODE_NORMAL LINEDISCONNECTMODE_REJECT LINEDISCONNECTMODE_TEMPFAILURE LINEDISCONNECTMODE_UNREACHABLE LINEDISCONNECTMODE_FACCMC (if negotiated extension version is 0x00050000 or greater)
dwMaxNumActiveCalls	For IP Phones, CTI Ports, and Park DNs: 1
	For CTI Route Points (without media): 0 For CTI Route Points (with media): Cisco Unified Communications Manager Administration configuration
dwMaxNumOnHoldCalls	For IP Phones, CTI Ports: 200
	For CTI Route Points: 0 For CTI Route Points (with media): Cisco Unified Communications Manager Administration configuration (same configuration as dwMaxNumActiveCalls)
	For Park DNs: 1
dwMaxNumOnHoldPendingCalls	For IP Phones and CTI Ports: 1
	For CTI Route Points and Park DNs: 0
dwMaxNumConference	For IP Phones, CTI Ports, and Park DNs: 16
	For CTI Route Points: 0

Members	Values
dwMaxNumTransConf	For All Devices: 0
dwAddrCapFlags	For IP Phones: LINEADDRCAPFLAGS_CONFERENCEHELD LINEADDRCAPFLAGS_DIALED LINEADDRCAPFLAGS_FWDSTATUSVALID LINEADDRCAPFLAGS_PARTIALDIAL LINEADDRCAPFLAGS_TRANSFERHELD
	For CTI Ports: LINEADDRCAPFLAGS_CONFERENCEHELD LINEADDRCAPFLAGS_DIALED LINEADDRCAPFLAGS_ACCEPTTOALERT LINEADDRCAPFLAGS_FWDSTATUSVALID LINEADDRCAPFLAGS_PARTIALDIAL LINEADDRCAPFLAGS_TRANSFERHELD
	For CTI Route Points: LINEADDRCAPFLAGS_ACCEPTTOALERT LINEADDRCAPFLAGS_FWDSTATUSVALID LINEADDRCAPFLAGS_ROUTEPOINT
	For Park DNs: LINEADDRCAPFLAGS_NOEXTERNALCALLS LINEADDRCAPFLAGS_NOINTERNALCALLS

Members	Values
dwCallFeatures	For IP Phones (except VG248 and ATA186) and CTI Ports: LINECALLFEATURE_ACCEPT LINECALLFEATURE_ADDTOCONF LINECALLFEATURE_ANSWER LINECALLFEATURE_BLINDTRANSFER LINECALLFEATURE_COMPLETETRANSF LINECALLFEATURE_DIAL LINECALLFEATURE_DROP LINECALLFEATURE_GATHERDIGITS LINECALLFEATURE_GENERATEDIGITS LINECALLFEATURE_GENERATETONE LINECALLFEATURE_HOLD LINECALLFEATURE_MONITORDIGITS LINECALLFEATURE_MONITORTONES LINECALLFEATURE_PARK LINECALLFEATURE_PREPAREADDTOCONF LINECALLFEATURE_REDIRECT LINECALLFEATURE_SETUPCONF LINECALLFEATURE_SETUPTRANSFER LINECALLFEATURE_UNHOLD LINECALLFEATURE_UNPARK
	For VG248 and ATA186 Devices: LINECALLFEATURE_ACCEPT LINECALLFEATURE_ADDTOCONF LINECALLFEATURE_BLINDTRANSFER LINECALLFEATURE_COMPLETETRANSF LINECALLFEATURE_DIAL LINECALLFEATURE_DROP LINECALLFEATURE_GATHERDIGITS LINECALLFEATURE_GENERATEDIGITS LINECALLFEATURE_GENERATETONE LINECALLFEATURE_HOLD LINECALLFEATURE_MONITORDIGITS LINECALLFEATURE_MONITORTONES LINECALLFEATURE_PARK LINECALLFEATURE_PREPAREADDTOCONF LINECALLFEATURE_REDIRECT LINECALLFEATURE_SETUPCONF LINECALLFEATURE_SETUPTRANSFER LINECALLFEATURE_UNHOLD LINECALLFEATURE_UNPARK

Members	Values
dwCallFeatures (continued)	For CTI Route Points (without media): LINECALLFEATURE_ACCEPT LINECALLFEATURE_DROP LINECALLFEATURE_REDIRECT For CTI Route Points (with media): LINECALLFEATURE_ACCEPT LINECALLFEATURE_ANSWER LINECALLFEATURE_DIAL LINECALLFEATURE_DROP LINECALLFEATURE_GATHERDIGITS LINECALLFEATURE_GENERATEDIGITS LINECALLFEATURE_GENERATETONE LINECALLFEATURE_HOLD LINECALLFEATURE_MONITORDIGITS LINECALLFEATURE_MONITORTONES LINECALLFEATURE_REDIRECT LINECALLFEATURE_UNHOLD
	For Park DNs: 0
dwRemoveFromConfCaps	For All Devices: 0
dwRemoveFromConfState	For All Devices: 0
dwTransferModes	For IP Phones and CTI Ports: LINETRANSFERMODE_TRANSFER LINETRANSFERMODE_CONFERENCE
	For CTI Route Points and Park DNs: 0
dwParkModes	For IP Phones and CTI Ports: LINEPARKMODE_NONDIRECTED
	For CTI Route Points and Park DNs: 0
dwForwardModes	For All Devices (except ParkDNs): LINEFORWARDMODE_UNCOND
	For Park DNs: 0
dwMaxForwardEntries	For All Devices (except ParkDNs): 1
	For Park DNs: 0
dwMaxSpecificEntries	For All Devices: 0
dwMinFwdNumRings	For All Devices: 0

Members	Values
dwMaxFwdNumRings	For All Devices: 0
dwMaxCallCompletions	For All Devices: 0
dwCallCompletionConds	For All Devices: 0
dwCallCompletionModes	For All Devices: 0
dwNumCompletionMessages	For All Devices: 0
dwCompletionMsgTextEntrySize	For All Devices: 0
dwCompletionMsgTextSize dwCompletionMsgTextOffset	For All Devices: 0
dwAddressFeatures	For IP Phones and CTI Ports: LINEADDRFEATURE_FORWARD LINEADDRFEATURE_FORWARDFWD LINEADDRFEATURE_MAKECALL
	For CTI Route Points: LINEADDRFEATURE_FORWARD LINEADDRFEATURE_FORWARDFWD
	For Park DNs: 0
dwPredictiveAutoTransferStates	For All Devices: 0
dwNumCallTreatments	For All Devices: 0
dwCallTreatmentListSize dwCallTreatmentListOffset	For All Devices: 0
dwDeviceClassesSize dwDeviceClassesOffset	For All Devices (except Park DNs): "tapi/line" "tapi/phone" "wave/in" "wave/out"
	For Park DNs: "tapi/line"
dwMaxCallDataSize	For All Devices: 0
dwCallFeatures2	For IP Phones and CTI Ports: LINECALLFEATURE2_TRANSFERNORM LINECALLFEATURE2_TRANSFERCONF
	For CTI Route Points and Park DNs: 0

Members	Values
dwMaxNoAnswerTimeout	For IP Phones and CTI Ports: 4294967295 (0xFFFFFFFF)
	For CTI Route Points and Park DNs: 0
dwConnectedModes	For IP Phones, CTI Ports LINECONNECTEDMODE_ACTIVE LINECONNECTEDMODE_INACTIVE
	For Park DNs: LINECONNECTEDMODE_ACTIVE
	For CTI Route Points (without media): 0 For CTI Route Points (with media) LINECONNECTEDMODE_ACTIVE
dwOfferingModes	For All Devices: LINEOFFERINGMODE_ACTIVE
dwAvailableMediaModes	For All Devices: 0

LINEADDRESSSTATUS

Members	Values
dwNumInUse	For All Devices: 1
dwNumActiveCalls	For All Devices: The number of calls on the address that are in call states other than idle, onhold, onholdpendingtransfer, and onholdpendingconference.
dwNumOnHoldCalls	For All Devices: The number of calls on the address in the onhold state.
dwNumOnHoldPendCalls	For All Devices: The number of calls on the address in the onholdpendingtransfer or the onholdpendingconference state.
dwAddressFeatures	For IP Phones and CTI Ports: LINEADDRFEATURE_FORWARD LINEADDRFEATURE_FORWARDFWD LINEADDRFEATURE_MAKECALL
	For CTI Route Points: LINEADDRFEATURE_FORWARD LINEADDRFEATURE_FORWARDFWD
	For Park DNs: 0
dwNumRingsNoAnswer	For All Devices: 0

Members	Values
dwForwardNumEntries	For All Devices (except Park DNs): The number of entries in the array to which dwForwardSize and dwForwardOffset refer.
	For Park DNs: 0
dwForwardSize dwForwardOffset	For All Devices (except Park DNs): The size, in bytes, and the offset, in bytes, from the beginning of this data structure of the variably sized field that describes the address forwarding information. This information appears as an array of dwForwardNumEntries elements, of type LINEFORWARD. Consider the offsets of the addresses in the array relative to the beginning of the LINEADDRESSSTATUS structure. The offsets dwCallerAddressOffset and dwDestAddressOffset in the variably sized field of type LINEFORWARD to which dwForwardSize and dwForwardOffset point are relative to the beginning of the LINEADDRESSSTATUS data structure (the root container).
	For Park DNs: 0
dwTerminalModesSize dwTerminalModesOffset	For All Devices: 0
dwDevSpecificSize dwDevSpecificOffset	For All Devices: 0

LINEAPPINFO

The LINEAPPINFO structure contains information about the application that is currently running. The LINEDEVSTATUS structure can contain an array of LINEAPPINFO structures.

Structure Details

```
typedef struct lineappinfo_tag {
  DWORD  dwMachineNameSize;
  DWORD  dwMachineNameOffset;
  DWORD  dwUserNameSize;
  DWORD  dwUserNameOffset;
  DWORD  dwModuleFilenameSize;
  DWORD  dwModuleFilenameOffset;
  DWORD  dwFriendlyNameSize;
  DWORD  dwFriendlyNameOffset;
  DWORD  dwMediaModes;
  DWORD  dwAddressID;
} LINEAPPINFO, *LPLINEAPPINFO;
```

Members	Values
dwMachineNameSize dwMachineNameOffset	Size (bytes) and offset from beginning of LINEDEVSTATUS of a string that specifies the name of the computer on which the application is executing.
dwUserNameSize dwUserNameOffset	Size (bytes) and offset from beginning of LINEDEVSTATUS of a string that specifies the user name under whose account the application is running.
dwModuleFilenameSize dwModuleFilenameOffset	Size (bytes) and offset from beginning of LINEDEVSTATUS of a string that specifies the module filename of the application. You can use this string in a call to lineHandoff to perform a directed handoff to the application.
dwFriendlyNameSize dwFriendlyNameOffset	Size (bytes) and offset from beginning of LINEDEVSTATUS of the string that the application provides to lineInitialize or lineInitializeEx, which should be used in any display of applications to the user.
dwMediaModes	The media types for which the application has requested ownership of new calls; zero if the line dwPrivileges did not include LINECALLPRIVILEGE_OWNER when it opened.
dwAddressID	If the line handle that was opened by using LINEOPENOPTION_SINGLEADDRESS contains the address identifier that is specified, set to 0xFFFFFFFF if the single address option was not used. An address identifier permanently associates with an address; the identifier remains constant across operating system upgrades.

LINECALLINFO

Members	Values
hLine	For All Devices: The handle for the line device with which this call is associated.
dwLineDeviceID	For All Devices: The device identifier of the line device with which this call is associated.
dwAddressID	For All Devices: 0
dwBearerMode	For All Devices: LINEBEARERMODE_SPEECH LINEBEARERMODE_VOICE
dwRate	For All Devices: 0

Members	Values
dwMediaMode	For IP Phones and Park DNs: LINEMEDIAMODE_INTERACTIVEVOICE
	For CTI Ports and CTI Route Points: LINEMEDIAMODE_AUTOMATEDVOICE LINEMEDIAMODE_INTERACTIVEVOICE
dwAppSpecific	For All Devices: Not interpreted by the API implementation and service provider. Any owner application of this call can set it with the lineSetAppSpecific function.
dwCallID	For All Devices: In some telephony environments, the switch or service provider can assign a unique identifier to each call. This allows the call to be tracked across transfers, forwards, or other events. The domain of these call IDs and their scope is service provider-defined. The dwCallID member makes this unique identifier available to the applications. The Cisco Unified TSP uses dwCallID to store the “GlobalCallID” of the call. The “GlobalCallID” represents a unique identifier that allows applications to identify all call handles that are related to a call.
dwRelatedCallID	For All Devices: 0
dwCallParamFlags	For All Devices: 0
dwCallStates	For IP Phones and CTI Ports: LINECALLSTATE_ACCEPTED LINECALLSTATE_CONFERENCED LINECALLSTATE_CONNECTED LINECALLSTATE_DIALING LINECALLSTATE_DIALTONE LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_ONHOLD LINECALLSTATE_ONHOLDPENDCONF LINECALLSTATE_ONHOLDPENDTRANSFER LINECALLSTATE_PROCEEDING LINECALLSTATE_RINGBACK LINECALLSTATE_UNKNOWN

Members	Values
dwCallStates (continued)	For CTI Route Points (without media): LINECALLSTATE_ACCEPTED LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_UNKNOWN For CTI Route Points (with media): LINECALLSTATE_ACCEPTED LINECALLSTATE_BUSY LINECALLSTATE_CONNECTED LINECALLSTATE_DIALING LINECALLSTATE_DIALTONE LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_ONHOLD LINECALLSTATE_PROCEEDING LINECALLSTATE_RINGBACK LINECALLSTATE_UNKNOWN
	For Park DNs: LINECALLSTATE_ACCEPTED LINECALLSTATE_CONFERENCED LINECALLSTATE_CONNECTED LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_ONHOLD LINECALLSTATE_UNKNOWN
dwMonitorDigitModes	For IP Phones, CTI Ports, and CTI Route Points (with media): LINEDIGITMODE_DTMF
	For CTI Route Points and Park DNs: 0
dwMonitorMediaModes	For IP Phones and Park DNs: LINEMEDIAMODE_INTERACTIVEVOICE
	For CTI Ports and CTI Route Points: LINEMEDIAMODE_AUTOMATEDVOICE LINEMEDIAMODE_INTERACTIVEVOICE
DialParams	For All Devices: 0
dwOrigin	For All Devices: LINECALLORIGIN_CONFERENCE LINECALLORIGIN_EXTERNAL LINECALLORIGIN_INTERNAL LINECALLORIGIN_OUTBOUND LINECALLORIGIN_UNAVAIL LINECALLORIGIN_UNKNOWN

Members	Values
dwReason	For All Devices: LINECALLREASON_DIRECT LINECALLREASON_FWDBUSY LINECALLREASON_FWDNOANSWER LINECALLREASON_FWDUNCOND LINECALLREASON_PARKED LINECALLREASON_PICKUP LINECALLREASON_REDIRECT LINECALLREASON_REMINDER LINECALLREASON_TRANSFER LINECALLREASON_UNKNOWN LINECALLREASON_UNPARK
dwCompletionID	For All Devices: 0
dwNumOwners	For All Devices: The number of application modules with different call handles with owner privilege for the call.
dwNumMonitors	For All Devices: The number of application modules with different call handles with monitor privilege for the call.
dwCountryCode	For All Devices: 0
dwTrunk	For All Devices: 0xFFFFFFFF
dwCallerIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN LINECALLPARTYID_BLOCKED
dwCallerIDSize dwCallerIDOffset	For All Devices: The size, in bytes, of the variably sized field that contains the caller party ID number information and the offset, in bytes, from the beginning of this data structure.
dwCallerIDNameSize dwCallerIDNameOffset	For All Devices: The size, in bytes, of the variably sized field that contains the caller party ID name information and the offset, in bytes, from the beginning of this data structure.
dwCalledIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN
dwCalledIDSize dwCalledIDOffset	For All Devices: The size, in bytes, of the variably sized field that contains the called-party ID number information and the offset, in bytes, from the beginning of this data structure.

Members	Values
dwCalledIDNameSize dwCalledIDNameOffset	For All Devices: The size, in bytes, of the variably sized field that contains the called-party ID name information and the offset, in bytes, from the beginning of this data structure.
dwConnectedIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN LINECALLPARTYID_BLOCKED
dwConnectedIDSize dwConnectedIDOffset	For All Devices: The size, in bytes, of the variably sized field that contains the connected party identifier number information and the offset, in bytes, from the beginning of this data structure.
dwConnectedIDNameSize dwConnectedIDNameOffset	For All Devices: The size, in bytes, of the variably sized field that contains the connected party identifier name information and the offset, in bytes, from the beginning of this data structure.
dwRedirectionIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN LINECALLPARTYID_BLOCKED
dwRedirectionIDSize dwRedirectionIDOffset	For All Devices: The size, in bytes, of the variably sized field that contains the redirection party identifier number information and the offset, in bytes, from the beginning of this data structure.
dwRedirectionIDNameSize dwRedirectionIDNameOffset	For All Devices: The size, in bytes, of the variably sized field that contains the redirection party identifier name information and the offset, in bytes, from the beginning of this data structure.
dwRedirectingIDFlags	For All Devices: LINECALLPARTYID_ADDRESS LINECALLPARTYID_NAME LINECALLPARTYID_UNKNOWN
dwRedirectingIDSize dwRedirectingIDOffset	For All Devices: The size, in bytes, of the variably sized field that contains the redirecting party identifier number information and the offset, in bytes, from the beginning of this data structure.
dwRedirectingIDNameSize dwRedirectingIDNameOffset	For All Devices: The size, in bytes, of the variably sized field that contains the redirecting party identifier name information and the offset, in bytes, from the beginning of this data structure.

Members	Values
dwAppNameSize dwAppNameOffset	For All Devices: The size, in bytes, and the offset, in bytes, from the beginning of this data structure of the variably sized field that holds the user-friendly application name of the application that first originated, accepted, or answered the call. This specifies the name that an application can specify in lineInitializeEx. If the application specifies no such name, the application module filename gets used instead.
dwDisplayableAddressSize dwDisplayableAddressOffset	For All Devices: 0
dwCalledPartySize dwCalledPartyOffset	For All Devices: 0
dwCommentSize dwCommentOffset	For All Devices: 0
dwDisplaySize dwDisplayOffset	For All Devices: 0
dwUserUserInfoSize dwUserUserInfoOffset	For All Devices: 0
dwHighLevelCompSize dwHighLevelCompOffset	For All Devices: 0
dwLowLevelCompSize dwLowLevelCompOffset	For All Devices: 0
dwChargingInfoSize dwChargingInfoOffset	For All Devices: 0
dwTerminalModesSize dwTerminalModesOffset	For All Devices: 0

Members	Values
dwDevSpecificSize dwDevSpecificOffset	For All Devices: If dwExtVersion >= 0x00060000 (6.0), this field will point to TSP_Unicode_Party_Names structure, If dwExtVersion >= 0x00070000 (7.0), this field will also point to a common structure that has a pointer to SRTP structure, DSCPValueForAudioCalls value, and Partition information. The "LINECALLINFO" section on page 6-6 defines the structure. The ExtendedCallInfo structure contains ExtendedCallReason that represents the last feature-related reason that caused a change in the callinfo/callstatus for this call. The ExtendedCallInfo will also provide SIP URL information for all call parties. If dwExtVersion >= 0x00080000 (8.0), this field will also point to common structure which has pointer to CallSecurityStatus structure. For IP Phones: If dwExtVersion >= 0x00080000 (8.0), this field will also point to common structure that has pointer to CallAtributeInfo and CCMCallID structure. The structures are defined below. If dwExtVersion >= 0x00080000 (8.0), this field will also point to common structure which has pointer to CallSecurityStatus structure. CallAttributeType: This field holds the information regarding the following info (DN.Partition.DeviceName) is for regular call, monitoring call, monitored call, recording call. PartyDNOffset, PartyDNSize, provides the size, in bytes, of the variably sized field that contains the Monitoring/Monitored/Recorder party DN information and the offset, in bytes, from the beginning of LINECALLINFO data structure. PartyPartitionOffset PartyPartitionSize, provides the size, in bytes, of the variably sized field that contains the Monitoring/Monitored/Recorder party Partition information and the offset, in bytes, from the beginning of LINECALLINFO data structure. DevcieNameSize provides the size, in bytes, of the variably sized field that contains the Monitoring/Monitored/Recorder party Device Name and the offset, in bytes, from the beginning of LINECALLINFO data structure. OverallCallSecurityStatus holds the security status of the call for two-party call as well for conference call. CCMCallID field holds the CCM call Id for each call leg.
dwCallTreatment	For All Devices: 0
dwCallDataSize dwCallDataOffset	For All Devices: 0

Members	Values
dwSendingFlowspecSize dwSendingFlowspecOffset	For All Devices: 0
dwReceivingFlowspecSize dwReceivingFlowspecOffset	For All Devices: 0

LINECALLLIST

The LINECALLLIST structure describes a list of call handles. The lineGetNewCalls and lineGetConfRelatedCalls functions return a structure of this type.

Note You must not extend this structure.

Structure Details

```
typedef struct linecalllist_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;
  DWORD  dwCallsNumEntries;
  DWORD  dwCallsSize;
  DWORD  dwCallsOffset;
} LINECALLLIST, FAR *LPLINECALLLIST;
```

Members	Values
dwTotalSize	The total size, in bytes, that is allocated to this data structure.
dwNeededSize	The size, in bytes, for this data structure that is needed to hold all the returned information.
dwUsedSize	The size, in bytes, of the portion of this data structure that contains useful information.
dwCallsNumEntries	The number of handles in the hCalls array.
dwCallsSize dwCallsOffset	The size, in bytes, and the offset, in bytes, from the beginning of this data structure of the variably sized field (which is an array of HCALL-sized handles).

LINECALLPARAMS

Members	Values
dwBearerMode	not supported
dwMinRate dwMaxRate	not supported
dwMediaMode	not supported
dwCallParamFlags	not supported
dwAddressMode	not supported
dwAddressID	not supported
DialParams	not supported
dwOrigAddressSize dwOrigAddressOffset	not supported
dwDisplayableAddressSize dwDisplayableAddressOffset	not supported
dwCalledPartySize dwCalledPartyOffset	not supported
dwCommentSize dwCommentOffset	not supported
dwUserUserInfoSize dwUserUserInfoOffset	not supported
dwHighLevelCompSize dwHighLevelCompOffset	not supported
dwLowLevelCompSize dwLowLevelCompOffset	not supported
dwDevSpecificSize dwDevSpecificOffset	not supported
dwPredictiveAutoTransferStates	not supported
dwTargetAddressSize dwTargetAddressOffset	not supported
dwSendingFlowspecSize dwSendingFlowspecOffset	not supported
dwReceivingFlowspecSize dwReceivingFlowspecOffset	not supported
dwDeviceClassSize dwDeviceClassOffset	not supported
dwDeviceConfigSize dwDeviceConfigOffset	not supported
dwCallDataSize dwCallDataOffset	not supported

Members	Values
dwNoAnswerTimeout	For All Devices: The number of seconds, after the completion of dialing, that the call should be allowed to wait in the PROCEEDING or RINGBACK state before the service provider automatically abandons it with a LINECALLSTATE_DISCONNECTED and LINEDISCONNECTMODE_NOANSWER. A value of 0 indicates that the application does not want automatic call abandonment.
dwCallingPartyIDSize dwCallingPartyIDOffset	not supported

LINECALLSTATUS

Members	Values
dwCallState	For IP Phones and CTI Ports: LINECALLSTATE_ACCEPTED LINECALLSTATE_CONFERENCED LINECALLSTATE_CONNECTED LINECALLSTATE_DIALING LINECALLSTATE_DIALTONE LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_ONHOLD LINECALLSTATE_ONHOLDPENDCONF LINECALLSTATE_ONHOLDPENDTRANSFER LINECALLSTATE_PROCEEDING LINECALLSTATE_RINGBACK LINECALLSTATE_UNKNOWN
	For CTI Route Points (without media): LINECALLSTATE_ACCEPTED LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_UNKNOWN For CTI Route Points (with media): LINECALLSTATE_ACCEPTED LINECALLSTATE_CONNECTED LINECALLSTATE_DIALING LINECALLSTATE_DIALTONE LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_ONHOLD LINECALLSTATE_PROCEEDING LINECALLSTATE_RINGBACK LINECALLSTATE_UNKNOWN

Members	Values
dwCallState (continued)	For Park DNs: LINECALLSTATE_ACCEPTED LINECALLSTATE_CONFERENCED LINECALLSTATE_CONNECTED LINECALLSTATE_DISCONNECTED LINECALLSTATE_IDLE LINECALLSTATE_OFFERING LINECALLSTATE_ONHOLD LINECALLSTATE_UNKNOWN
dwCallStateMode	For IP Phones, CTI Ports: LINECONNECTEDMODE_ACTIVE LINECONNECTEDMODE_INACTIVE LINEDIALTONEMODE_NORMAL LINEDIALTONEMODE_UNAVAIL LINEDISCONNECTMODE_BADADDRESS LINEDISCONNECTMODE_BUSY LINEDISCONNECTMODE_CONGESTION LINEDISCONNECTMODE_FORWARDED LINEDISCONNECTMODE_NOANSWER LINEDISCONNECTMODE_NORMAL LINEDISCONNECTMODE_REJECT LINEDISCONNECTMODE_TEMPFAILURE LINEDISCONNECTMODE_UNREACHABLE LINEDISCONNECTMODE_FACCMC (if negotiated extension version is 0x00050000 or greater)
	For CTI Route Points: LINEDISCONNECTMODE_BADADDRESS LINEDISCONNECTMODE_BUSY LINEDISCONNECTMODE_CONGESTION LINEDISCONNECTMODE_FORWARDED LINEDISCONNECTMODE_NOANSWER LINEDISCONNECTMODE_NORMAL LINEDISCONNECTMODE_REJECT LINEDISCONNECTMODE_TEMPFAILURE LINEDISCONNECTMODE_UNREACHABLE LINEDISCONNECTMODE_FACCMC (if negotiated extension version is 0x00050000 or greater)
	For Park DNs: LINECONNECTEDMODE_ACTIVE LINEDISCONNECTMODE_BADADDRESS LINEDISCONNECTMODE_BUSY LINEDISCONNECTMODE_CONGESTION LINEDISCONNECTMODE_FORWARDED LINEDISCONNECTMODE_NOANSWER LINEDISCONNECTMODE_NORMAL LINEDISCONNECTMODE_REJECT LINEDISCONNECTMODE_TEMPFAILURE LINEDISCONNECTMODE_UNREACHABLE

Members	Values
dwCallPrivilege	For All Devices LINECALLPRIVILEGE_MONITOR LINECALLPRIVILEGE_NONE LINECALLPRIVILEGE_OWNER
dwCallFeatures	For IP Phones (except VG248 and ATA186) and CTI Ports: LINECALLFEATURE_ACCEPT LINECALLFEATURE_ADDTOCONF LINECALLFEATURE_ANSWER LINECALLFEATURE_BLINDTRANSFER LINECALLFEATURE_COMPLETETRANSF LINECALLFEATURE_DIAL LINECALLFEATURE_DROP LINECALLFEATURE_GATHERDIGITS LINECALLFEATURE_GENERATEDIGITS LINECALLFEATURE_GENERATETONE LINECALLFEATURE_HOLD LINECALLFEATURE_MONITORDIGITS LINECALLFEATURE_MONITORTONES LINECALLFEATURE_PARK LINECALLFEATURE_PREPAREADDTOCONF LINECALLFEATURE_REDIRECT LINECALLFEATURE_SETUPCONF LINECALLFEATURE_SETUPTRANSFER LINECALLFEATURE_UNHOLD LINECALLFEATURE_UNPARK
	For VG248 and ATA186 Devices: LINECALLFEATURE_ACCEPT LINECALLFEATURE_ADDTOCONF LINECALLFEATURE_BLINDTRANSFER LINECALLFEATURE_COMPLETETRANSF LINECALLFEATURE_DIAL LINECALLFEATURE_DROP LINECALLFEATURE_GATHERDIGITS LINECALLFEATURE_GENERATEDIGITS LINECALLFEATURE_GENERATETONE LINECALLFEATURE_HOLD LINECALLFEATURE_MONITORDIGITS LINECALLFEATURE_MONITORTONES LINECALLFEATURE_PARK LINECALLFEATURE_PREPAREADDTOCONF LINECALLFEATURE_REDIRECT LINECALLFEATURE_SETUPCONF LINECALLFEATURE_SETUPTRANSFER LINECALLFEATURE_UNHOLD LINECALLFEATURE_UNPARK

Members	Values
dwCallFeatures (continued)	For CTI Route Points (without media): LINECALLFEATURE_ACCEPT LINECALLFEATURE_DROP LINECALLFEATURE_REDIRECT For CTI Route Points (with media): LINECALLFEATURE_ACCEPT LINECALLFEATURE_ANSWER LINECALLFEATURE_BLINDTRANSFER LINECALLFEATURE_DIAL LINECALLFEATURE_DROP LINECALLFEATURE_GATHERDIGITS LINECALLFEATURE_GENERATEDIGITS LINECALLFEATURE_GENERATETONE LINECALLFEATURE_HOLD LINECALLFEATURE_MONITORDIGITS LINECALLFEATURE_MONITORTONES LINECALLFEATURE_REDIRECT LINECALLFEATURE_UNHOLD
dwCallFeatures (continued)	For Park DNs: 0
dwDevSpecificSize dwDevSpecificOffset	For All Devices: 0
dwCallFeatures2	For IP Phones and CTI Ports: LINECALLFEATURE2_TRANSFERNORM LINECALLFEATURE2_TRANSFERCONF
	For CTI Route Points and Park DNs: 0
tStateEntryTime	For All Devices: The Coordinated Universal Time at which the current call state was entered.

LINECARDENTRY

The LINECARDENTRY structure describes a calling card. The LINETRANSLATECAPS structure can contain an array of LINECARDENTRY structures.

Note You must not extend this structure.

Structure Details

```
typedef struct linecardentry_tag {
  DWORD  dwPermanentCardID;
  DWORD  dwCardNameSize;
  DWORD  dwCardNameOffset;
  DWORD  dwCardNumberDigits;
  DWORD  dwSameAreaRuleSize;
  DWORD  dwSameAreaRuleOffset;
  DWORD  dwLongDistanceRuleSize;
  DWORD  dwLongDistanceRuleOffset;
  DWORD  dwInternationalRuleSize;
  DWORD  dwInternationalRuleOffset;
  DWORD  dwOptions;
} LINECARDENTRY, FAR *LPLINECARDENTRY;
```

Members

Members	Values
dwPermanentCardID	The permanent identifier that identifies the card.
dwCardNameSize dwCardNameOffset	A null-terminated string (size includes the NULL) that describes the card in a user-friendly manner.
dwCardNumberDigits	The number of digits in the existing card number. The card number itself is not returned for security reasons (TAPI stores it in scrambled form). The application can use this parameter to insert filler bytes into a text control in "password" mode to show that a number exists.
dwSameAreaRuleSize dwSameAreaRuleOffset	The offset, in bytes, from the beginning of the LINETRANSLATECAPS structure and the total number of bytes in the dialing rule that is defined for calls to numbers in the same area code. The rule specifies a null-terminated string.
dwLongDistanceRuleSize dwLongDistanceRuleOffset	The offset, in bytes, from the beginning of the LINETRANSLATECAPS structure and the total number of bytes in the dialing rule that is defined for calls to numbers in the other areas in the same country or region. The rule specifies a null-terminated string.
dwInternationalRuleSize dwInternationalRuleOffset	The offset, in bytes, from the beginning of the LINETRANSLATECAPS structure and the total number of bytes in the dialing rule that is defined for calls to numbers in other countries/regions. The rule specifies a null-terminated string.
dwOptions	Indicates other settings that are associated with this calling card, by using the LINECARDOPTION_

LINECOUNTRYENTRY

The LINECOUNTRYENTRY structure provides the information for a single country entry. An array of one or more of these structures makes up part of the LINECOUNTRYLIST structure that the lineGetCountry function returns.

Note You must not extend this structure.

Structure Details

```
typedef struct linecountryentry_tag {
  DWORD  dwCountryID;
  DWORD  dwCountryCode;
  DWORD  dwNextCountryID;
  DWORD  dwCountryNameSize;
  DWORD  dwCountryNameOffset;
  DWORD  dwSameAreaRuleSize;
  DWORD  dwSameAreaRuleOffset;
  DWORD  dwLongDistanceRuleSize;
  DWORD  dwLongDistanceRuleOffset;
  DWORD  dwInternationalRuleSize;
  DWORD  dwInternationalRuleOffset;
} LINECOUNTRYENTRY, FAR *LPLINECOUNTRYENTRY;
```

Members	Values
dwCountryID	The country or region identifier of the entry that specifies an internal identifier that allows multiple entries to exist in the country or region list with the same country code (for example, all countries in North America and the Caribbean share country code 1, but require separate entries in the list).
dwCountryCode	The actual country code of the country or region that the entry represents (that is, the digits that would be dialed in an international call). Display only this value to users (Country IDs should never display, as they could be confusing).
dwNextCountryID	The country identifier of the next entry in the country or region list. Because country codes and identifiers are not assigned in numeric sequence, the country or region list represents a single linked list, with each entry pointing to the next. The last country or region in the list includes a dwNextCountryID value of zero. When the LINECOUNTRYLIST structure is used to obtain the entire list, the entries in the list appear in sequence as linked by their dwNextCountryID members.
dwCountryNameSize dwCountryNameOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINECOUNTRYLIST structure of a null-terminated string that gives the name of the country or region.
dwSameAreaRuleSize dwSameAreaRuleOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINECOUNTRYLIST structure of a null-terminated string that contains the dialing rule for direct-dialed calls to the same area code.

Members	Values
dwLongDistanceRuleSize dwLongDistanceRuleOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINECOUNTRYLIST structure of a null-terminated string that contains the dialing rule for direct-dialed calls to other areas in the same country or region.
dwInternationalRuleSize dwInternationalRuleOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINECOUNTRYLIST structure of a null-terminated string that contains the dialing rule for direct-dialed calls to other countries/regions.

LINECOUNTRYLIST

The LINECOUNTRYLIST structure describes a list of countries/regions. This structure can contain an array of LINECOUNTRYENTRY structures. The lineGetCountry function returns LINECOUNTRYLIST.

Note You must not extend this structure.

Structure Details

```
typedef struct linecountrylist_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;
  DWORD  dwNumCountries;
  DWORD  dwCountryListSize;
  DWORD  dwCountryListOffset;
} LINECOUNTRYLIST, FAR *LPLINECOUNTRYLIST;
```

Members	Values
dwTotalSize	The total size, in bytes, that are allocated to this data structure.
dwNeededSize	The size, in bytes, for this data structure that is needed to hold all the returned information.
dwUsedSize	The size, in bytes, of the portion of this data structure that contains useful information.
dwNumCountries	The number of LINECOUNTRYENTRY structures that are present in the array dwCountryListSize and dwCountryListOffset dominate.
dwCountryListSize dwCountryListOffset	The size, in bytes, and the offset, in bytes, from the beginning of this data structure of an array of LINECOUNTRYENTRY elements that provide information on each country or region.

LINEDEVCAPS

Members	Values
dwProviderInfoSize dwProviderInfoOffset	For All Devices: The size, in bytes, of the variably sized field that contains service provider information and the offset, in bytes, from the beginning of this data structure. The dwProviderInfoSize/ Offset member provides information about the provider hardware and/or software. This information is useful when a user needs to call customer service with problems regarding the provider. The Cisco Unified TSP sets this field to "Cisco Unified TSPxxx.TSP: Cisco IP PBX Service Provider Ver. x.x(x.x)" where the text before the colon specifies the file name of the TSP and the text after "Ver." specifies the version of TSP.
dwSwitchInfoSize dwSwitchInfoOffset	For All Devices: The size, in bytes, of the variably sized device field that contains switch information and the offset, in bytes, from the beginning of this data structure. The dwSwitchInfoSize/Offset member provides information about the switch to which the line device connects, such as the switch manufacturer, the model name, the software version, and so on. This information is useful when a user needs to call customer service with problems regarding the switch. The Cisco Unified TSP sets this field to "Cisco Unified Communications Manager Ver. x.x(x.x), Cisco CTI Manager Ver x.x(x.x)" where the text after "Ver." specifies the version of the Cisco Unified Communications Manager and the version of the CTI Manager, respectively.
dwPermanentLineID	For All Devices: The permanent DWORD identifier by which the line device is known in the system configuration. This identifier specifies a permanent name for the line device. This permanent name (as opposed to dwDeviceID) does not change as lines are added or removed from the system and persists through operating system upgrades. You can therefore use it to link line-specific information in .ini files (or other files) in a way that is not affected by adding or removing other lines or by changing the operating system.
dwLineNameSize dwLineNameOffset	For All Devices: The size, in bytes, of the variably sized device field that contains a user-configurable name for this line device and the offset, in bytes, from the beginning of this data structure. You can configure this name when you configure the line device service provider, and the name gets provided for the convenience of the user. Cisco Unified TSP sets this field to “Cisco Line: [deviceName] (dirn)” where deviceName specifies the name of the device on which the line resides, and dirn specifies the directory number for the device.
dwStringFormat	For All Devices: STRINGFORMAT_ASCII

Members	Values
dwAddressModes	For All Devices: LINEADDRESSMODE_ADDRESSID
dwNumAddresses	For All Devices: 1
dwBearerModes	For All Devices: LINEBEARERMODE_SPEECH LINEBEARERMODE_VOICE
dwMaxRate	For All Devices: 0
dwMediaModes	For IP Phones and Park DNs: LINEMEDIAMODE_INTERACTIVEVOICE
	For CTI Ports and CTI Route Points: LINEMEDIAMODE_AUTOMATEDVOICE LINEMEDIAMODE_INTERACTIVEVOICE
dwGenerateToneModes	For IP Phones, CTI Ports, and CTI Route Points (with media): LINETONEMODE_BEEP
	For CTI Route Points (without media) and Park DNs: 0
dwGenerateToneMaxNumFreq	For All Devices: 0
dwGenerateDigitModes	For IP Phones, CTI Ports, and CTI Route Points (with media): LINETONEMODE_DTMF
	For CTI Route Points and Park DNs: 0
dwMonitorToneMaxNumFreq	For All Devices: 0
dwMonitorToneMaxNumEntries	For All Devices: 0
dwMonitorDigitModes	For IP Phones, CTI Ports, and CTI Route Points (with media): LINETONEMODE_DTMF
	For CTI Route Points (without media) and Park DNs: 0
dwGatherDigitsMinTimeout dwGatherDigitsMaxTimeout	For All Devices: 0
dwMedCtlDigitMaxListSize dwMedCtlMediaMaxListSize dwMedCtlToneMaxListSize dwMedCtlCallStateMaxListSize	For All Devices: 0
dwDevCapFlags	For IP Phones: 0
	For All Other Devices: LINEDEVCAPFLAGS_CLOSEDROP

Members	Values
dwMaxNumActiveCalls	For All Devices: 1
	For CTI Route Points (without media): 0 For CTI Route Points (with media): Cisco Unified Communications Manager Administration configuration
dwAnswerMode	For IP Phones (except for VG248 and ATA186), CTI Route Points (with media) and CTI Ports: LINEANSWERMODE_HOLD
	For VG248 devices, ATA186 devices, CTI Route Points (without media), and Park DNs: 0
dwRingModes	For All Devices: 1
dwLineStates	For IP Phones, CTI Ports, and Route Points (with media): LINEDEVSTATE_CLOSE LINEDEVSTATE_DEVSPECIFIC LINEDEVSTATE_INSERVICE LINEDEVSTATE_MSGWAITOFF LINEDEVSTATE_MSGWAITON LINEDEVSTATE_NUMCALLS LINEDEVSTATE_OPEN LINEDEVSTATE_OUTOFSERVICE LINEDEVSTATE_REINIT LINEDEVSTATE_RINGING LINEDEVSTATE_TRANSLATECHANGE
	For CTI Route Points (without media): LINEDEVSTATE_CLOSE LINEDEVSTATE_INSERVICE LINEDEVSTATE_OPEN LINEDEVSTATE_OUTOFSERVICE LINEDEVSTATE_REINIT LINEDEVSTATE_RINGING LINEDEVSTATE_TRANSLATECHANGE
	For Park DNs: LINEDEVSTATE_CLOSE LINEDEVSTATE_DEVSPECIFIC LINEDEVSTATE_INSERVICE LINEDEVSTATE_NUMCALLS LINEDEVSTATE_OPEN LINEDEVSTATE_OUTOFSERVICE LINEDEVSTATE_REINIT LINEDEVSTATE_TRANSLATECHANGE
dwUUIAcceptSize	For All Devices: 0

Members	Values
dwUUIAnswerSize	For All Devices: 0
dwUUIMakeCallSize	For All Devices: 0
dwUUIDropSize	For All Devices: 0
dwUUISendUserUserInfoSize	For All Devices: 0
dwUUICallInfoSize	For All Devices: 0
MinDialParams MaxDialParams	For All Devices: 0
DefaultDialParams	For All Devices: 0
dwNumTerminals	For All Devices: 0
dwTerminalCapsSize dwTerminalCapsOffset	For All Devices: 0
dwTerminalTextEntrySize	For All Devices: 0
dwTerminalTextSize dwTerminalTextOffset	For All Devices: 0
dwDevSpecificSize dwDevSpecificOffset	For All Devices (except ParkDNs): If dwExtVersion > 0x00030000 (3.0): LINEDEVCAPS_DEV_SPECIFIC.m_ DevSpecificFlags = 0
	For Park DNs: If dwExtVersion > 0x00030000 (3.0): LINEDEVCAPS_DEV_SPECIFIC.m_ DevSpecificFlags = LINEDEVCAPSDEVSPECIFIC_PARKDN For Intercom DNs: LINEDEVCAPS_DEV_SPECIFIC. M_DevSpecificFlags = LINEDEVCAPSDEVSPECIFIC_INTERCOMDN LOCALE info PARTITION_INFO INTERCOM_SPEEDDIAL_INFO

Members	Values
dwLineFeatures	For IP Phones, CTI Ports, and CTI Route Points (with media): LINEFEATURE_DEVSPECIFIC LINEFEATURE_FORWARD LINEFEATURE_FORWARDFWD LINEFEATURE_MAKECALL
	For CTI Route Points (without media): LINEFEATURE_FORWARD LINEFEATURE_FORWARDFWD
	For Park DNs: 0
dwSettableDevStatus	For All Devices: 0
dwDeviceClassesSize dwDeviceClassesOffset	For IP Phones and CTI Route Points: "tapi/line" "tapi/phone"
	For CTI Ports: "tapi/line" "tapi/phone" "wave/in" "wave/out"
	For Park DNs: "tapi/line"
PermanentLineGuid	The GUID that is permanently associated with the line device.

LINEDEVSTATUS

Members	Values
dwNumOpens	For All Devices: The number of active opens on the line device.
dwOpenMediaModes	For All Devices: Bit array that indicates for which media types the line device is currently open.
dwNumActiveCalls	For All Devices: The number of calls on the line in call states other than idle, onhold, onholdpendingtransfer, and onholdpendingconference.
dwNumOnHoldCalls	For All Devices: The number of calls on the line in the onhold state.
dwNumOnHoldPendCalls	For All Devices: The number of calls on the line in the onholdpendingtransfer or onholdpendingconference state.

Members	Values
dwLineFeatures	For IP Phones, CTI Ports, and CTI Route Points (with media): LINEFEATURE_DEVSPECIFIC LINEFEATURE_FORWARD LINEFEATURE_FORWARDFWD LINEFEATURE_MAKECALL
	For CTI Route Points (without media): LINEFEATURE_FORWARD LINEFEATURE_FORWARDFWD
	For Park DNs: 0
dwNumCallCompletions	For All Devices: 0
dwRingMode	For All Devices: 0
dwSignalLevel	For All Devices: 0
dwBatteryLevel	For All Devices: 0
dwRoamMode	For All Devices: 0
dwDevStatusFlags	For IP Phones and CTI Ports: LINEDEVSTATUSGLAGS_CONNECTED LINEDEVSTATUSGLAGS_INSERVICE LINEDEVSTATUSGLAGS_MSGWAIT
	For CTI Route Points and Park DNs: LINEDEVSTATUSGLAGS_CONNECTED LINEDEVSTATUSGLAGS_INSERVICE
dwTerminalModesSize dwTerminalModesOffset	For All Devices: 0
dwDevSpecificSize dwDevSpecificOffset	For All Devices: 0
dwAvailableMediaModes	For All Devices: 0
dwAppInfoSize dwAppInfoOffset	For All Devices: Length, in bytes, and offset from the beginning of LINEDEVSTATUS of an array of LINEAPPINFO structures. The dwNumOpens member indicates the number of elements in the array. Each element in the array identifies an application that has the line open.

LINEEXTENSIONID

Members	Values
dwExtensionID0	For All Devices: 0x8EBD6A50
dwExtensionID1	For All Devices: 0x128011D2
dwExtensionID2	For All Devices: 0x905B0060
dwExtensionID3	For All Devices: 0xB03DD275

LINEFORWARD

The LINEFORWARD structure describes an entry of the forwarding instructions.

Structure Details

```
typedef struct lineforward_tag {
    DWORD  dwForwardMode;
    DWORD  dwCallerAddressSize;
    DWORD  dwCallerAddressOffset;
    DWORD  dwDestCountryCode;
    DWORD  dwDestAddressSize;
    DWORD  dwDestAddressOffset;
} LINEFORWARD, FAR *LPLINEFORWARD;
```

Members	Values
dwForwardMode	The types of forwarding. The dwForwardMode member can have only a single bit set. This member uses the following LINEFORWARDMODE_ constants: LINEFORWARDMODE_UNCOND Forward all calls unconditionally, irrespective of their origin. Use this value when unconditional forwarding for internal and external calls cannot be controlled separately. Unconditional forwarding overrides forwarding on busy and/or no-answer conditions. **Note** LINEFORWARDMODE_UNCOND is the only forward mode that Cisco Unified TSP supports. LINEFORWARDMODE_UNCONDINTERNAL Forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

Members	Values
	LINEFORWARDMODE_UNCONDEXTERNAL Forward all external calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately. LINEFORWARDMODE_UNCONDSPECIFIC Unconditionally forward all calls that originated at a specified address (selective call forwarding). LINEFORWARDMODE_BUSY Forward all calls on busy, irrespective of their origin. Use this value when forwarding for internal and external calls both on busy and on no answer cannot be controlled separately. LINEFORWARDMODE_BUSYINTERNAL Forward all internal calls on busy. Use this value when forwarding for internal and external calls on busy and on no answer can be controlled separately. LINEFORWARDMODE_BUSYEXTERNAL Forward all external calls on busy. Use this value when forwarding for internal and external calls on busy and on no answer can be controlled separately.

Members	Values
dwForwardMode (continued)	LINEFORWARDMODE_BUSYSPECIFIC Forward on busy all calls that originated at a specified address (selective call forwarding). LINEFORWARDMODE_NOANSW Forward all calls on no answer, irrespective of their origin. Use this value when call forwarding for internal and external calls on no answer cannot be controlled separately. LINEFORWARDMODE_NOANSWINTERNAL Forward all internal calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately. LINEFORWARDMODE_NOANSWEXTERNAL Forward all external calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately. LINEFORWARDMODE_NOANSWSPECIFIC Forward all calls that originated at a specified address on no answer (selective call forwarding). LINEFORWARDMODE_BUSYNA Forward all calls on busy or no answer, irrespective of their origin. Use this value when forwarding for internal and external calls on both busy and on no answer cannot be controlled separately. LINEFORWARDMODE_BUSYNAINTERNAL Forward all internal calls on busy or no answer. Use this value when call forwarding on busy and on no answer cannot be controlled separately for internal calls. LINEFORWARDMODE_BUSYNAEXTERNAL Forward all external calls on busy or no answer. Use this value when call forwarding on busy and on no answer cannot be controlled separately for internal calls. LINEFORWARDMODE_BUSYNASPECIFIC Forward on busy or no answer all calls that originated at a specified address (selective call forwarding). LINEFORWARDMODE_UNKNOWN Calls get forwarded, but the conditions under which forwarding occurs are not known at this time. LINEFORWARDMODE_UNAVAIL Calls are forwarded, but the conditions under which forwarding occurs are not known and are never known by the service provider.

Members	Values
dwCallerAddressSize dwCallerAddressOffset	The size in bytes of the variably sized address field that contains the address of a caller to be forwarded and the offset in bytes from the beginning of the containing data structure. The dwCallerAddressSize/Offset member gets set to zero if dwForwardMode is not one of the following choices: LINEFORWARDMODE_BUSYNASPECIFIC, LINEFORWARDMODE_NOANSWSPECIFIC, LINEFORWARDMODE_UNCONDSPECIFIC, or LINEFORWARDMODE_BUSYSPECIFIC.
dwDestCountryCode	The country code of the destination address to which the call is to be forwarded.
dwDestAddressSize dwDestAddressOffset	The size in bytes of the variably sized address field that contains the address where calls are to be forwarded and the offset in bytes from the beginning of the containing data structure.

LINEFORWARDLIST

The LINEFORWARDLIST structure describes a list of forwarding instructions.

Structure Details

```
typedef struct lineforwardlist_tag {
    DWORD  dwTotalSize;
    DWORD  dwNumEntries;
    LINEFORWARD  ForwardList[1];
} LINEFORWARDLIST, FAR *LPLINEFORWARDLIST;
```

Members	Values
dwTotalSize	The total size in bytes of the data structure.
dwNumEntries	Number of entries in the array, specified as ForwardList[ ].
ForwardList[ ]	An array of forwarding instruction. The array entries specify type LINEFORWARD.

LINEGENERATETONE

The LINEGENERATETONE structure contains information about a tone to be generated. The lineGenerateTone and TSPI_lineGenerateTone functions use this structure.

Note You must not extend this structure.

This structure gets used only for the generation of tones; it is not used for tone monitoring.

Structure Details

```
typedef struct linegeneratetone_tag {
  DWORD  dwFrequency;
  DWORD  dwCadenceOn;
  DWORD  dwCadenceOff;
  DWORD  dwVolume;
} LINEGENERATETONE, FAR *LPLINEGENERATETONE;
```

Members	Values
dwFrequency	The frequency, in hertz, of this tone component. A service provider may adjust (round up or down) the frequency that the application specified to fit its resolution.
dwCadenceOn	The “on” duration, in milliseconds, of the cadence of the custom tone to be generated. Zero means no tone gets generated.
dwCadenceOff	The “off” duration, in milliseconds, of the cadence of the custom tone to be generated. Zero means no off time, that is, a constant tone.
dwVolume	The volume level at which the tone gets generated. A value of 0x0000FFFF represents full volume, and a value of 0x00000000 means silence.

LINEINITIALIZEEXPARAMS

The LINEINITIZALIZEEXPARAMS structure describes parameters that are supplied when calls are made by using LINEINITIALIZEEX.

Structure Details

```
typedef struct lineinitializeexparams_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;
  DWORD  dwOptions;

 union
 {
  HANDLE  hEvent;
  HANDLE  hCompletionPort;
 } Handles;

  DWORD  dwCompletionKey;

} LINEINITIALIZEEXPARAMS, FAR *LPLINEINITIALIZEEXPARAMS;
```

Members	Values
dwTotalSize	The total size, in bytes, that is allocated to this data structure.
dwNeededSize	The size, in bytes, for this data structure that is needed to hold all the returned information.

Members	Values
dwUsedSize	The size, in bytes, of the portion of this data structure that contains useful information.
dwOptions	One of the LINEINITIALIZEEXOPTION_ constants. Specifies the event notification mechanism that the application wants to use.
hEvent	If dwOptions specifies LINEINITIALIZEEXOPTION_USEEVENT, TAPI returns the event handle in this field.
hCompletionPort	If dwOptions specifies LINEINITIALIZEEXOPTION_USECOMPLETIONPORT, the application must specify in this field the handle of an existing completion port that was opened by using CreateIoCompletionPort.
dwCompletionKey	If dwOptions specifies LINEINITIALIZEEXOPTION_USECOMPLETIONPORT, the application must specify in this field a value that is returned through the lpCompletionKey parameter of GetQueuedCompletionStatus to identify the completion message as a telephony message.

Further Details

See "lineInitializeEx" for further information on these options.

LINELOCATIONENTRY

The LINELOCATIONENTRY structure describes a location that is used to provide an address translation context. The LINETRANSLATECAPS structure can contain an array of LINELOCATIONENTRY structures.

Note You must not extend this structure.

Structure Details

```
typedef struct linelocationentry_tag {
  DWORD  dwPermanentLocationID;
  DWORD  dwLocationNameSize;
  DWORD  dwLocationNameOffset;

  DWORD  dwCityCodeSize;
  DWORD  dwCityCodeOffset;
  DWORD  dwPreferredCardID;
  DWORD  dwLocalAccessCodeSize;
  DWORD  dwLocalAccessCodeOffset;
  DWORD  dwLongDistanceAccessCodeSize;
  DWORD  dwLongDistanceAccessCodeOffset;
  DWORD  dwTollPrefixListSize;
  DWORD  dwTollPrefixListOffset;
  DWORD  dwCountryID;
```

```
  DWORD  dwOptions;
  DWORD  dwCancelCallWaitingSize;
  DWORD  dwCancelCallWaitingOffset;
} LINELOCATIONENTRY, FAR *LPLINELOCATIONENTRY;
```

Members	Values
dwPermanentLocationID	The permanent identifier that identifies the location.
dwLocationNameSize dwLocationNameOffset	Contains a null-terminated string (size includes the NULL) that describes the location in a user-friendly manner.
dwCountryCode	The country code of the location.
dwPreferredCardID	The preferred calling card when dialing from this location.
dwCityCodeSize dwCityCodeOffset	Contains a null-terminated string that specifies the city or area code that is associated with the location (the size includes the NULL). Applications can use this information, along with the country code, to "default" entry fields for the user when you enter the phone numbers, to encourage the entry of proper canonical numbers.
dwLocalAccessCodeSize dwLocalAccessCodeOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure of a null-terminated string that contains the access code to be dialed before calls to addresses in the local calling area.
dwLongDistanceAccessCodeSize dwLongDistanceAccessCodeOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure of a null-terminated string that contains the access code to be dialed before calls to addresses outside the local calling area.
dwTollPrefixListSize dwTollPrefixListOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure of a null-terminated string that contains the toll prefix list for the location. The string contains only prefixes that consist of the digits "0" through "9" and are separated from each other by a single "," (comma) character.
dwCountryID	The country identifier of the country or region that is selected for the location. Use this identifier with the lineGetCountry function to obtain additional information about the specific country or region, such as the country or region name (you cannot use the dwCountryCode member for this purpose because country codes are not unique).
dwOptions	Indicates options in effect for this location with values taken from the LINELOCATIONOPTION_ Constants.

Members	Values
dwCancelCallWaitingSize dwCancelCallWaitingOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure of a null-terminated string that contains the dial digits and modifier characters that should be prefixed to the dialable string (after the pulse/tone character) when an application sets the LINETRANSLATEOPTION_CANCELCALLWAITING bit in the dwTranslateOptions parameter of lineTranslateAddress. If no prefix is defined, dwCancelCallWaitingSize set to zero may indicate this, or dwCancelCallWaitingSize set to 1 and dwCancelCallWaitingOffset pointing to an empty string (single NULL byte) may indicate this.

LINEMESSAGE

The LINEMESSAGE structure contains parameter values that specify a change in status of the line that the application currently has open. The lineGetMessage function returns the LINEMESSAGE structure.

Structure Details

```
typedef struct linemessage_tag {
  DWORD  hDevice;
  DWORD  dwMessageID;
  DWORD_PTR  dwCallbackInstance;
  DWORD_PTR  dwParam1;
  DWORD_PTR  dwParam2;
  DWORD_PTR  dwParam3;
} LINEMESSAGE, FAR *LPLINEMESSAGE;
```

Members	Values
hDevice	A handle to either a line device or a call. The context that dwMessageID provides can determine the nature of this handle (line handle or call handle).
dwMessageID	A line or call device message.
dwCallbackInstance	Instance data passed back to the application, which the application in the dwCallBackInstance parameter of lineInitializeEx specified. TAPI does not interpret this DWORD.
dwParam1	A parameter for the message.
dwParam2	A parameter for the message.
dwParam3	A parameter for the message.

Further Details

For details about the parameter values that are passed in this structure, see “TAPI Line Messages.”

LINEMONITORTONE

The LINEMONITORTONE structure defines a tone for the purpose of detection. Use this as an entry in an array. An array of tones gets passed to the lineMonitorTones function that monitors these tones and sends a LINE_MONITORTONE message to the application when a detection is made.

A tone with all frequencies set to zero corresponds to silence. An application can thus monitor the call information stream for silence.

Note You must not extend this structure.

Structure Details

```
typedef struct linemonitortone_tag {
  DWORD  dwAppSpecific;
  DWORD  dwDuration;
  DWORD  dwFrequency1;
  DWORD  dwFrequency2;
  DWORD  dwFrequency3;
} LINEMONITORTONE, FAR *LPLINEMONITORTONE;
```

Members	Values
dwAppSpecific	Used by the application for tagging the tone. When this tone is detected, the value of the dwAppSpecific member gets passed back to the application.
dwDuration	The duration, in milliseconds, during which the tone should be present before a detection is made.
dwFrequency1	dwFrequency2
dwFrequency3	The frequency, in hertz, of a component of the tone. If fewer than three frequencies are needed in the tone, a value of 0 should be used for the unused frequencies. A tone with all three frequencies set to zero gets interpreted as silence and can be used for silence detection.

LINEPROVIDERENTRY

The LINEPROVIDERENTRY structure provides the information for a single service provider entry. An array of these structures gets returned as part of the LINEPROVIDERLIST structure that the function lineGetProviderList returns.

Note You cannot extend this structure.

Structure Details

```
typedef struct lineproviderentry_tag {
  DWORD  dwPermanentProviderID;
  DWORD  dwProviderFilenameSize;
  DWORD  dwProviderFilenameOffset;
```

```
} LINEPROVIDERENTRY, FAR *LPLINEPROVIDERENTRY;
```

Members	Values
dwPermanentProviderID	The permanent provider identifier of the entry.
dwProviderFilenameSize dwProviderFilenameOffset	The size, in bytes, and the offset, in bytes, from the beginning of the LINEPROVIDERLIST structure of a null-terminated string that contains the filename (path) of the service provider DLL (.TSP) file.

LINEPROVIDERLIST

The LINEPROVIDERLIST structure describes a list of service providers. The lineGetProviderList function returns a structure of this type. The LINEPROVIDERLIST structure can contain an array of LINEPROVIDERENTRY structures.

Note You must not extend this structure.

Structure Details

```
typedef struct lineproviderlist_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;
DWORD  dwNumProviders;
  DWORD  dwProviderListSize;
  DWORD  dwProviderListOffset;
} LINEPROVIDERLIST, FAR *LPLINEPROVIDERLIST;
```

Members	Values
dwTotalSize	The total size, in bytes, that are allocated to this data structure.
dwNeededSize	The size, in bytes, for this data structure that is needed to hold all the returned information.
dwUsedSize	The size, in bytes, of the portion of this data structure that contains useful information.
dwNumProviders	The number of LINEPROVIDERENTRY structures that are present in the array that is denominated by dwProviderListSize and dwProviderListOffset.
dwProviderListSize dwProviderListOffset	The size, in bytes, and the offset, in bytes, from the beginning of this data structure of an array of LINEPROVIDERENTRY elements, which provide the information on each service provider.

LINEREQMAKECALL

The LINEREQMAKECALL structure describes a request that a call initiated to the lineGetRequest function.

Note You cannot extend this structure.

Structure Details

```
typedef struct linereqmakecall_tag {
  char  szDestAddress[TAPIMAXDESTADDRESSSIZE];
  char  szAppName[TAPIMAXAPPNAMESIZE];
  char  szCalledParty[TAPIMAXCALLEDPARTYSIZE];
  char  szComment[TAPIMAXCOMMENTSIZE];
} LINEREQMAKECALL, FAR *LPLINEREQMAKECALL;
```

Members	Values
szDestAddress [TAPIMAXADDRESSSIZE]	The null-terminated destination address of the make-call request. The address uses the canonical address format or the dialable address format. The maximum length of the address specifies TAPIMAXDESTADDRESSSIZE characters, which include the NULL terminator. Longer strings get truncated.
szAppName [TAPIMAXAPPNAMESIZE]	The null-terminated, user-friendly application name or filename of the application that originated the request. The maximum length of the address specifies TAPIMAXAPPNAMESIZE characters, which include the NULL terminator.
szCalledParty [TAPIMAXCALLEDPARTYSIZE]	The null-terminated, user-friendly called-party name. The maximum length of the called-party information specifies TAPIMAXCALLEDPARTYSIZE characters, which include the NULL terminator.
szComment [TAPIMAXCOMMENTSIZE]	The null-terminated comment about the call request. The maximum length of the comment string specifies TAPIMAXCOMMENTSIZE characters, which include the NULL terminator.

LINETRANSLATECAPS

The LINETRANSLATECAPS structure describes the address translation capabilities. This structure can contain an array of LINELOCATIONENTRY structures and an array of LINECARDENTRY structures. the lineGetTranslateCaps function returns the LINETRANSLATECAPS structure.

Note You must not extend this structure.

Structure Details

```
typedef struct linetranslatecaps_tag {
```

```
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;
  DWORD  dwNumLocations;
  DWORD  dwLocationListSize;
  DWORD  dwLocationListOffset;
  DWORD  dwCurrentLocationID;
  DWORD  dwNumCards;
  DWORD  dwCardListSize;
  DWORD  dwCardListOffset;
  DWORD  dwCurrentPreferredCardID;
} LINETRANSLATECAPS, FAR *LPLINETRANSLATECAPS;
```

Members	Values
dwTotalSize	The total size, in bytes, that is allocated to this data structure.
dwNeededSize	The size, in bytes, for this data structure that is needed to hold all the returned information.
dwUsedSize	The size, in bytes, of the portion of this data structure that contains useful information.
dwNumLocations	The number of entries in the location list. It includes all locations that are defined, including zero (default).
dwLocationListSize dwLocationListOffset	List of locations that are known to the address translation. The list comprises a sequence of LINELOCATIONENTRY structures. The dwLocationListOffset member points to the first byte of the first LINELOCATIONENTRY structure, and the dwLocationListSize member indicates the total number of bytes in the entire list.
dwCurrentLocationID	The dwPermanentLocationID member from the LINELOCATIONENTRY structure for the CurrentLocation.
dwNumCards	The number of entries in the CardList.
dwCardListSize dwCardListOffset	List of calling cards that are known to the address translation. It includes only non-hidden card entries and always includes card 0 (direct dial). The list comprises a sequence of LINECARDENTRY structures. The dwCardListOffset member points to the first byte of the first LINECARDENTRY structure, and the dwCardListSize member indicates the total number of bytes in the entire list.
dwCurrentPreferredCardID	The dwPreferredCardID member from the LINELOCATIONENTRY structure for the CurrentLocation.

LINETRANSLATEOUTPUT

The LINETRANSLATEOUTPUT structure describes the result of an address translation. The lineTranslateAddress function uses this structure.

Note You must not extend this structure.

Structure Details

```
typedef struct linetranslateoutput_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;
  DWORD  dwDialableStringSize;
  DWORD  dwDialableStringOffset;
  DWORD  dwDisplayableStringSize;
  DWORD  dwDisplayableStringOffset;
  DWORD  dwCurrentCountry;
  DWORD  dwDestCountry;
  DWORD  dwTranslateResults;
} LINETRANSLATEOUTPUT, FAR *LPLINETRANSLATEOUTPUT;
```

Members	Values
dwTotalSize	The total size, in bytes, that is allocated to this data structure.
dwNeededSize	The size, in bytes, for this data structure that is needed to hold all the returned information.
dwUsedSize	The size, in bytes, of the portion of this data structure that contains useful information.
dwDialableStringSize dwDialableStringOffset	Contains the translated output that can be passed to the lineMakeCall, lineDial, or other function that requires a dialable string. The output always comprises a null-terminated string (NULL gets included in the count in dwDialableStringSize). This output string includes ancillary fields such as name and subaddress if they were in the input string. This string may contain private information such as calling card numbers. To prevent inadvertent visibility to unauthorized persons, it should not display to the user.
dwDisplayableStringSize dwDisplayableStringOffset	Contains the translated output that can display to the user for confirmation. Identical to DialableString, except the “friendly name” of the card enclosed within bracket characters (for example, “[AT&T Card]”) replaces calling card digits. The ancillary fields, such as name and subaddress, get removed. You can display this string in call-status dialog boxes without exposing private information to unauthorized persons. You can also include this information in call logs.
dwCurrentCountry	Contains the country code that is configured in CurrentLocation. Use this value to control the display by the application of certain user interface elements for local call progress tone detection and for other purposes.

Members	Values
dwDestCountry	Contains the destination country code of the translated address. This value may pass to the dwCountryCode parameter of lineMakeCall and other dialing functions (so the call progress tones of the destination country or region such as a busy signal are properly detected). This field gets set to zero if the destination address that is passed to lineTranslateAddress is not in canonical format.
dwTranslateResults	Indicates the information that is derived from the translation process, which may assist the application in presenting user-interface elements. This field uses one LINETRANSLATERESULT_.

TAPI Phone Functions

TAPI phone functions enable an application to control physical aspects of a phone

Table 5-4 TAPI Phone Functions

TAPI Phone Functions
phoneCallbackFunc
phoneClose
phoneDevSpecific
phoneGetDevCaps
phoneGetDisplay
phoneGetLamp
phoneGetMessage
phoneGetRing
phoneGetStatus
phoneGetStatusMessages
phoneInitialize
phoneInitializeEx
phoneNegotiateAPIVersion
phoneOpen
phoneSetDisplay
phoneSetLamp
phoneSetStatusMessages
phoneShutdown

phoneCallbackFunc

The phoneCallbackFunc function provides a placeholder for the application-supplied function name.

All callbacks occur in the application context. The callback function must reside in a dynamic-link library (DLL) or application module and be exported in the module-definition file.

Function Details

```
VOID FAR PASCAL phoneCallbackFunc(
  HANDLE hDevice,
  DWORD dwMsg,
  DWORD dwCallbackInstance,
  DWORD dwParam1,
  DWORD dwParam2,
  DWORD dwParam3
);
```

Parameters

hDevice

A handle to a phone device that is associated with the callback.

dwMsg

A line or call device message.

dwCallbackInstance

Callback instance data that is passed to the application in the callback. TAPI does not interpret this DWORD.

dwParam1

A parameter for the message.

dwParam2

A parameter for the message.

dwParam3

A parameter for the message.

Further Details

For more information about the parameters that are passed to this callback function, see "TAPI Line Messages" and "TAPI Phone Messages."

phoneClose

The phoneClose function closes the specified open phone device.

Function Details

```
LONG phoneClose(
  HPHONE hPhone
);
```

Parameter

hPhone

A handle to the open phone device that is to be closed. If the function succeeds, this means that the handle is no longer valid.

phoneDevSpecific

The phoneDevSpecific function gets used as a general extension mechanism to enable a telephony API implementation to provide features that are not described in the other TAPI functions. The meanings of these extensions are device specific.

When used with the Cisco Unified TSP, you can use phoneDevSpecific to send device-specific data to a phone device.

Function Details

```
LONG WINAPI phoneDevSpecific (
    HPHONE hPhone,
    LPVOID lpParams,
    DWORD dwSize
);
```

Parameter

hPhone

A handle to a phone device.

lpParams

A pointer to a memory area used to hold a parameter block. Its interpretation is device specific. TAPI passes the contents of the parameter block unchanged to or from the service provider.

dwSize

The size in bytes of the parameter block area.

phoneGetDevCaps

The phoneGetDevCaps function queries a specified phone device to determine its telephony capabilities.

Function Details

```
LONG phoneGetDevCaps(
  HPHONEAPP hPhoneApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  DWORD dwExtVersion,
  LPPHONECAPS lpPhoneCaps
);
```

Parameters

hPhoneApp

The handle to the registration with TAPI for this application.

dwDeviceID

The phone device that is to be queried.

dwAPIVersion

The version number of the telephony API that is to be used. The high-order word contains the major version number; the low-order word contains the minor version number. You can obtain this number with the function phoneNegotiateAPIVersion.

dwExtVersion

The version number of the service provider-specific extensions to be used. This number is obtained with the function phoneNegotiateExtVersion. It can be left as zero if no device-specific extensions are to be used. Otherwise, the high-order word contains the major version number, the low-order word contains the minor version number.

lpPhoneCaps

A pointer to a variably sized structure of type PHONECAPS. Upon successful completion of the request, this structure is filled with phone device capabilities information.

phoneGetDisplay

The phoneGetDisplay function returns the current contents of the specified phone display.

Function Details

```
LONG phoneGetDisplay(
  HPHONE hPhone,
  LPVARSTRING lpDisplay
);
```

Parameters

hPhone

A handle to the open phone device.

lpDisplay

A pointer to the memory location where the display content is to be stored, of type VARSTRING.

phoneGetLamp

The phoneGetLamp function returns the current lamp mode of the specified lamp.

Note Cisco Unified IP Phones 79xx series do not support this function.

Function Details

```
LONG phoneGetLamp(
  HPHONE hPhone,
  DWORD dwButtonLampID,
  LPDWORD lpdwLampMode
);
```

Parameters

hPhone

A handle to the open phone device.

dwButtonLampID

The identifier of the lamp that is to be queried. See Table 5-7, "Phone Button Values" for lamp IDs.

lpdwLampMode

> **Note** Cisco Unified IP Phones 79xx series do not support this function.

A pointer to a memory location that holds the lamp mode status of the given lamp. The lpdwLampMode parameter can have at most one bit set. This parameter uses the following PHONELAMPMODE_ constants:

- PHONELAMPMODE_FLASH - Flash means slow on and off.
- PHONELAMPMODE_FLUTTER - Flutter means fast on and off.
- PHONELAMPMODE_OFF - The lamp is off.
- PHONELAMPMODE_STEADY - The lamp is continuously lit.
- PHONELAMPMODE_WINK - The lamp winks.
- PHONELAMPMODE_UNKNOWN - The lamp mode is currently unknown.
- PHONELAMPMODE_DUMMY - Use this value to describe a button/lamp position that has no corresponding lamp.

phoneGetMessage

The phoneGetMessage function returns the next TAPI message that is queued for delivery to an application that is using the Event Handle notification mechanism (see phoneInitializeEx for further details).

Function Details

```
LONG WINAPI phoneGetMessage(
  HPHONEAPP hPhoneApp,
  LPPHONEMESSAGE lpMessage,
  DWORD dwTimeout
);
```

Parameters

hPhoneApp

The handle that phoneInitializeEx returns. The application must have set the PHONEINITIALIZEEXOPTION_USEEVENT option in the dwOptions member of the PHONEINITIALIZEEXPARAMS structure.

lpMessage

A pointer to a PHONEMESSAGE structure. Upon successful return from this function, the structure contains the next message that had been queued for delivery to the application.

dwTimeout

The time-out interval, in milliseconds. The function returns if the interval elapses, even if no message can be returned. If dwTimeout is zero, the function checks for a queued message and returns immediately. If dwTimeout is INFINITE, the time-out interval never elapses.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

PHONEERR_INVALAPPHANDLE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPOINTER, PHONEERR_NOMEM.

phoneGetRing

The phoneGetRing function enables an application to query the specified open phone device as to its current ring mode.

Function Details

```
LONG phoneGetRing(
  HPHONE hPhone,
  LPDWORD lpdwRingMode,
  LPDWORD lpdwVolume
);
```

Parameters

hPhone

A handle to the open phone device.

lpdwRingMode

The ringing pattern with which the phone is ringing. Zero indicates that the phone is not ringing.

The system supports four ring modes.

Table 5-5 lists the valid ring modes.

Table 5-5 Ring Modes

Ring Modes	Definition
0	Off
1	Inside Ring
2	Outside Ring
3	Feature Ring

lpdwVolume

The volume level with which the phone is ringing. This parameter has no meaning; the value 0x8000 always gets returned.

phoneGetStatus

The phoneGetStatus function enables an application to query the specified open phone device for its overall status.

Function Details

```
LONG WINAPI phoneGetStatusMessages(
    HPHONE hPhone,
    LPPHONESTATUS lpPhoneStatus
    ) ;
```

Parameters

hPhone

A handle to the open phone device to be queried.

lpPhoneStatus

A pointer to a variably sized data structure of type PHONESTATUS, which is loaded with the returned information about the phone status.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Return values include the following:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL PHONEERR_OPERATIONFAILED, PHONEERR_STRUCTURETOOSMALL PHONEERR_OPERATIONUNAVAIL, PHONEERR_UNINITIALIZED

phoneGetStatusMessages

The phoneGetStatusMessages function returns information about which phone-state changes on the specified phone device generate a callback to the application.

An application can use phoneGetStatusMessages to query the generation of the corresponding messages. The phoneSetStatusMessages can control Message generation. All phone status messages remain disabled by default.

Function Details

```
LONG WINAPI phoneGetStatusMessages(
  HPHONE hPhone,
  LPDWORD lpdwPhoneStates,
  LPDWORD lpdwButtonModes,
  LPDWORD lpdwButtonStates
);
```

Parameters

hPhone

A handle to the open phone device that is to be monitored.

lpdwPhoneStates

A pointer to a DWORD holding zero, one or more of the PHONESTATE_ Constants. These flags specify the set of phone status changes and events for which the application can receive notification messages. You can enable or disable monitoring individually for the following states:

- PHONESTATE_OTHER
- PHONESTATE_CONNECTED
- PHONESTATE_DISCONNECTED
- PHONESTATE_OWNER
- PHONESTATE_MONITORS
- PHONESTATE_DISPLAY
- PHONESTATE_LAMP
- PHONESTATE_RINGMODE
- PHONESTATE_RINGVOLUME
- PHONESTATE_HANDSETHOOKSWITCH
- PHONESTATE_HANDSETVOLUME
- PHONESTATE_HANDSETGAIN
- PHONESTATE_SPEAKERHOOKSWITCH
- PHONESTATE_SPEAKERVOLUME
- PHONESTATE_SPEAKERGAIN
- PHONESTATE_HEADSETHOOKSWITCH
- PHONESTATE_HEADSETVOLUME
- PHONESTATE_HEADSETGAIN
- PHONESTATE_SUSPEND
- PHONESTATE_RESUMEF
- PHONESTATE_DEVSPECIFIC
- PHONESTATE_REINIT
- PHONESTATE_CAPSCHANGE
- PHONESTATE_REMOVED

lpdwButtonModes

A pointer to a DWORD that contains flags that specify the set of phone-button modes for which the application can receive notification messages. This parameter uses zero, one, or more of the PHONEBUTTONMODE_ Constants.

lpdwButtonStates

A pointer to a DWORD that contains flags that specify the set of phone button state changes for which the application can receive notification messages. This parameter uses zero, one, or more of the PHONEBUTTONSTATE_ Constants.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

PHONEERR_INVALPHONEHANDLE

PHONEERR_NOMEM

PHONEERR_INVALPOINTER

PHONEERR_RESOURCEUNAVAIL

PHONEERR_OPERATIONFAILED

PHONEERR_UNINITIALIZED.

phoneInitialize

Although the phoneInitialize function is obsolete, tapi.dll and tapi32.dll continue to export it for backward compatibility with applications that are using TAPI versions 1.3 and 1.4.

Function Details

```
LONG WINAPI phoneInitialize(
  LPHPHONEAPP lphPhoneApp,
  HINSTANCE hInstance,
  PHONECALLBACK lpfnCallback,
  LPCSTR lpszAppName,
  LPDWORD lpdwNumDevs
);
```

Parameters

lphPhoneApp

A pointer to a location that is filled with the application usage handle for TAPI.

hInstance

The instance handle of the client application or DLL.

lpfnCallback

The address of a callback function that is invoked to determine status and events on the phone device.

lpszAppName

A pointer to a null-terminated string that contains displayable characters. If this parameter is non-NULL, it contains an application-supplied name of the application. This name, which is provided in the PHONESTATUS structure, indicates, in a user-friendly way, which application is the current owner of the phone device. You can use this information for logging and status reporting purposes. If lpszAppName is NULL, the application filename gets used instead.

lpdwNumDevs

A pointer to DWORD. This location gets loaded with the number of phone devices that are available to the application.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

PHONEERR_INVALAPPNAME

PHONEERR_INIFILECORRUPT

PHONEERR_INVALPOINTER

PHONEERR_NOMEM

PHONEERR_OPERATIONFAILED

PHONEERR_REINIT

PHONEERR_RESOURCEUNAVAIL

PHONEERR_NODEVICE

PHONEERR_NODRIVER

PHONEERR_INVALPARAM

phoneInitializeEx

The phoneInitializeEx function initializes the application use of TAPI for subsequent use of the phone abstraction. It registers the application specified notification mechanism and returns the number of phone devices that are available to the application. A phone device represents any device that provides an implementation for the phone-prefixed functions in the telephony API.

Function Details

```
LONG WINAPI phoneInitializeEx(
  LPHPHONEAPP lphPhoneApp,
  HINSTANCE hInstance,
  PHONECALLBACK lpfnCallback,
  LPCSTR lpszFriendlyAppName,
  LPDWORD lpdwNumDevs,
  LPDWORD lpdwAPIVersion,
  LPPHONEINITIALIZEEXPARAMS lpPhoneInitializeExParams
);
```

Parameters

lphPhoneApp

A pointer to a location that is filled with the application usage handle for TAPI.

hInstance

The instance handle of the client application or DLL. The application or DLL can pass NULL for this parameter, in which case TAPI uses the module handle of the root executable of the process.

lpfnCallback

The address of a callback function that is invoked to determine status and events on the line device, addresses, or calls, when the application is using the "hidden window" method of event notification (for more information, see phoneCallbackFunc). When the application chooses to use the event handle or completion port event notification mechanisms, this parameter gets ignored and should be set to NULL.

lpszFriendlyAppName

A pointer to a null-terminated string that contains only displayable characters. If this parameter is not NULL, it contains an application-supplied name for the application. This name, which is provided in the PHONESTATUS structure, indicates, in a user-friendly way, which application has ownership of the phone device. If lpszFriendlyAppName is NULL, the application module filename gets used instead (as returned by the Windows function GetModuleFileName).

lpdwNumDevs

A pointer to a DWORD. Upon successful completion of this request, the number of phone devices that are available to the application fills this location.

lpdwAPIVersion

A pointer to a DWORD. The application must initialize this DWORD, before calling this function, to the highest API version that it is designed to support (for example, the same value that it would pass into dwAPIHighVersion parameter of phoneNegotiateAPIVersion). Do no use artificially high values; ensure the values are accurately set. TAPI translates any newer messages or structures into values or formats that the application version supports. Upon successful completion of this request, the highest API version that TAPI supports fills this location, which allows the application to detect and adapt to being installed on a system with an older version of TAPI.

lpPhoneInitializeExParams

A pointer to a structure of type PHONEINITIALIZEEXPARAMS that contains additional parameters that are used to establish the association between the application and TAPI (specifically, the application-selected event notification mechanism and associated parameters).

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

PHONEERR_INVALAPPNAME

PHONEERR_OPERATIONFAILED

PHONEERR_INIFILECORRUPT

PHONEERR_INVALPOINTER

PHONEERR_REINIT

PHONEERR_NOMEM

PHONEERR_INVALPARAM

phoneNegotiateAPIVersion

Use the phoneNegotiateAPIVersion function to negotiate the API version number to be used with the specified phone device. It returns the extension identifier that the phone device supports, or zeros if no extensions are provided.

Function Details

```
LONG WINAPI phoneNegotiateAPIVersion(
  HPHONEAPP hPhoneApp,
  DWORD dwDeviceID,
  DWORD dwAPILowVersion,
  DWORD dwAPIHighVersion,
  LPDWORD lpdwAPIVersion,
  LPPHONEEXTENSIONID lpExtensionID
);
```

Parameters

hPhoneApp

The handle to the application registration with TAPI.

dwDeviceID

The phone device to be queried.

dwAPILowVersion

The least recent API version with which the application is compliant. The high-order word represents the major version number, and the low-order word represents the minor version number.

dwAPIHighVersion

The most recent API version with which the application is compliant. The high-order word represents the major version number, and the low-order word represents the minor version number.

lpdwAPIVersion

A pointer to a DWORD in which the API version number that was negotiated will be returned. If negotiation succeeds, this number ranges from dwAPILowVersion to dwAPIHighVersion.

lpExtensionID

A pointer to a structure of type PHONEEXTENSIONID. If the service provider for the specified dwDeviceID parameter supports provider-specific extensions, this structure gets filled with the extension identifier of these extensions when negotiation succeeds. This structure contains all zeros if the line provides no extensions. An application can ignore the returned parameter if it does not use extensions.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

PHONEERR_INVALAPPHANDLE

PHONEERR_OPERATIONFAILED

PHONEERR_BADDEVICEID

PHONEERR_OPERATIONUNAVAIL

PHONEERR_NODRIVER

PHONEERR_NOMEM

PHONEERR_INVALPOINTER

PHONEERR_RESOURCEUNAVAIL,PHONEERR_INCOMPATIBLEAPIVERSION

PHONEERR_UNINITIALIZED

PHONEERR_NODEVICE

phoneOpen

The phoneOpen function opens the specified phone device. Open the device by using either owner privilege or monitor privilege. An application that opens the phone with owner privilege can control the lamps, display, ringer, and hookswitch or hookswitches that belong to the phone. An application that opens the phone device with monitor privilege receives notification only about events that occur at the phone, such as hookswitch changes or button presses. Because ownership of a phone device is exclusive, only one application at a time can have a phone device opened with owner privilege. The phone device can, however, be opened multiple times with monitor privilege.

Note To open a phone device on a CTI port, first ensure a corresponding line device is open.

Function Details

```
LONG phoneOpen(
  HPHONEAPP hPhoneApp,
  DWORD dwDeviceID,
  LPHPHONE lphPhone,
  DWORD dwAPIVersion,
  DWORD dwExtVersion,
  DWORD dwCallbackInstance,
  DWORD dwPrivilege
);
```

Parameters

hPhoneApp

A handle by which the application is registered with TAPI.

dwDeviceID

The phone device to be opened.

lphPhone

A pointer to an HPHONE handle that identifies the open phone device. Use this handle to identify the device when invoking other phone control functions.

dwAPIVersion

The API version number under which the application and telephony API agreed to operate. Obtain this number from phoneNegotiateAPIVersion.

dwExtVersion

The extension version number under which the application and the service provider agree to operate. This number is zero if the application does not use any extensions. Obtain this number from phoneNegotiateExtVersion.

Note The Cisco Unified TSP does not support any phone extensions.

dwCallbackInstance

User instance data that is passed back to the application with each message. The telephony API does not interpret this parameter.

dwPrivilege

The privilege requested. The dwPrivilege parameter can have only one bit set. This parameter uses the following PHONEPRIVILEGE_ constants:

- PHONEPRIVILEGE_MONITOR - An application that opens a phone device with this privilege gets informed about events and state changes that occur on the phone. The application cannot invoke any operations on the phone device that would change its state.
- PHONEPRIVILEGE_OWNER - An application that opens a phone device in this mode can change the state of the lamps, ringer, display, and hookswitch devices of the phone. Having owner privilege to a phone device automatically includes monitor privilege as well.

phoneSetDisplay

The phoneSetDisplay function causes the specified string to display on the specified open phone device.

Note Prior to Release 4.0, Cisco Unified Communications Manager messages that were passed to the phone would automatically overwrite any messages sent to the phone by using phoneSetDisplay(). In Cisco Unified Communications Manager 4.0, the message sent to the phone in the phoneSetDisplay() API remains on the phone until the phone is rebooted. If the application wants to clear the text from the display and see the Cisco Unified Communications Manager messages again, a NULL string, not spaces, should be passed in the phoneSetDisplay() API. In other words, the lpsDisplay parameter should be NULL and the dwSize should be set to 0.

Function Details

```
LONG phoneSetDisplay(
  HPHONE hPhone,
  DWORD dwRow,
  DWORD dwColumn,
  LPCSTR lpsDisplay,
  DWORD dwSize
);
```

Parameters

hPhone

A handle to the open phone device. The application must be the owner of the phone.

dwRow

The row position on the display where the new text displays.

dwColumn

The column position on the display where the new text displays.

lpsDisplay

A pointer to the memory location where the display content is stored. The display information must follow the format that is specified in the dwStringFormat member of the device capabilities for this phone.

dwSize

The size in bytes of the information to which lpsDisplay points.

phoneSetLamp

The phoneSetLamp function causes the specified lamp to glow on the open phone device in the specified lamp mode.

Function Details

```
LONG phoneSetLamp(
  HPHONE hPhone,
  DWORD dwButtonLampID,
  DWORD dwLampMode
);
```

Parameters

hPhone

A handle to the open phone device. Ensure that the application is the owner of the phone.

dwButtonLampID

The button that glows. See "Phone Button Values" Table 5-7 for lamp IDs.

dwLampMode

Note Cisco Unified IP Phones 79xx series does not support this function.

Indicates how the lamp must glow. The dwLampMode parameter can have only a single bit set. This parameter uses the following PHONELAMPMODE_ constants:

- PHONELAMPMODE_FLASH - Flash means slow on and off.
- PHONELAMPMODE_FLUTTER - Flutter means fast on and off.
- PHONELAMPMODE_OFF - The lamp is off.
- PHONELAMPMODE_STEADY - The lamp is continuously on.
- PHONELAMPMODE_WINK - The lamp blinks.
- PHONELAMPMODE_DUMMY - This value describes a button/lamp position that has no corresponding lamp.

phoneSetStatusMessages

The phoneSetStatusMessages function enables an application to monitor the specified phone device for selected status events.

See “TAPI Phone Messages” for supported messages.

Function Details

```
LONG phoneSetStatusMessages(
  HPHONE hPhone,
  DWORD dwPhoneStates,
  DWORD dwButtonModes,
  DWORD dwButtonStates
);
```

Parameters

hPhone

A handle to the open phone device to be monitored.

dwPhoneStates

These flags specify the set of phone status changes and events for which the application can receive notification messages. This parameter can have zero, one, or more bits set. This parameter uses the following PHONESTATE_ constants:

- PHONESTATE_OTHER - Phone status items other than those in the following list changed. The application should check the current phone status to determine which items changed.
- PHONESTATE_OWNER - The number of owners for the phone device changed.
- PHONESTATE_MONITORS - The number of monitors for the phone device changed.
- PHONESTATE_DISPLAY - The display of the phone changed.
- PHONESTATE_LAMP - A lamp of the phone changed.
- PHONESTATE_RINGMODE - The ring mode of the phone changed.
- PHONESTATE_SPEAKERHOOKSWITCH - The hookswitch state changed for this speakerphone.
- PHONESTATE_REINIT - Items changed in the configuration of phone devices. To become aware of these changes (as with the appearance of new phone devices), the application should reinitialize its use of TAPI. New phoneInitialize, phoneInitializeEx, and phoneOpen requests get denied until applications have shut down their usage of TAPI. The hDevice parameter of the PHONE_STATE message stays NULL for this state change because it applies to any line in the system. Because of the critical nature of PHONESTATE_REINIT, you cannot mask such messages, so the setting of this bit gets ignored, and the messages always get delivered to the application.
- PHONESTATE_REMOVED - Indicates that the service provider is removing the device from the system (most likely through user action, through a control panel or similar utility). A PHONE_CLOSE message on the device immediately follows a PHONE_STATE message with this value. Subsequent attempts to access the device prior to TAPI being reinitialized result in PHONEERR_NODEVICE being returned to the application. If a service provider sends a

PHONE_STATE message that contains this value to TAPI, TAPI passes it along to applications that negotiated TAPI version 1.4 or later; applications that negotiated a previous TAPI version do not receive any notification.

dwButtonModes

The set of phone-button modes for which the application can receive notification messages. This parameter can have zero, one, or more bits set. This parameter uses the following PHONEBUTTONMODE_ constants:

- PHONEBUTTONMODE_CALL - The button is assigned to a call appearance.
- PHONEBUTTONMODE_FEATURE - The button is assigned to requesting features from the switch, such as hold, conference, and transfer.
- PHONEBUTTONMODE_KEYPAD - The button is one of the 12 keypad buttons, ‘0’ through ‘9’, ‘*’, and ‘#’.
- PHONEBUTTONMODE_DISPLAY - The button is a “soft” button that is associated with the phone display. A phone set can have zero or more display buttons.

dwButtonStates

The set of phone-button state changes for which the application can receive notification messages. If the dwButtonModes parameter is zero, the system ignores dwButtonStates. If dwButtonModes has one or more bits set, this parameter also must have at least one bit set. This parameter uses the following PHONEBUTTONSTATE_ constants:

- PHONEBUTTONSTATE_UP - The button is in the “up” state.
- PHONEBUTTONSTATE_DOWN - The button is in the “down” state (pressed down).
- PHONEBUTTONSTATE_UNKNOWN - The up or down state of the button is unknown at this time but may become known later.
- PHONEBUTTONSTATE_UNAVAIL - The service provider does not know the up or down state of the button, and the state will not become known.

phoneShutdown

The phoneShutdown function shuts down the application usage of the TAPI phone abstraction.

Note If this function is called when the application has open phone devices, these devices are closed.

Function Details

```
LONG WINAPI phoneShutdown(
  HPHONEAPP hPhoneApp
);
```

Parameter

hPhoneApp

The application usage handle for TAPI.

Return Values

Returns zero if the request succeeds or a negative number if an error occurs. Possible return values follow:

PHONEERR_INVALAPPHANDLE, PHONEERR_NOMEM, PHONEERR_UNINITIALIZED, PHONEERR_RESOURCEUNAVAIL.

TAPI Phone Messages

Messages notify the application of asynchronous events. All messages get sent to the application through the message notification mechanism that the application specified in lineInitializeEx. The message always contains a handle to the relevant object (phone, line, or call), of which the application can determine the type from the message type. Table 5-6 describes TAPI Phone messages.

Table 5-6 TAPI Phone Messages

TAPI Phone Messages
PHONE_BUTTON
PHONE_CLOSE
PHONE_CREATE
PHONE_REMOVE
PHONE_REPLY
PHONE_STATE

PHONE_BUTTON

The PHONE_BUTTON message notifies the application that button press monitoring is enabled if it has detected a button press on the local phone.

Function Details

```
PHONE_BUTTON
hPhone = (HPHONE) hPhoneDevice;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) idButtonOrLamp;
dwParam2 = (DWORD) ButtonMode;
dwParam3 = (DWORD) ButtonState;
```

Parameters

hPhone

A handle to the phone device.

dwCallbackInstance

The callback instance that is provided when the phone device for this application is opened.

dwParam1

The button/lamp identifier of the button that was pressed. Button identifiers zero through 11 always represent the KEYPAD buttons, with ‘0’ being button identifier zero, ‘1’ being button identifier 1 (and so on through button identifier 9), and with ‘*’ being button identifier 10, and ‘#’ being button identifier 11. Find additional information about a button identifier with phoneGetDevCaps.

dwParam2

The button mode of the button. The button mode for each button ID gets listed as “Phone Button Values”.

The TAPI service provider cannot detect button down or button up state changes. To conform to the TAPI specification, two messages are sent to simulate a down state followed by an up state in dwparam3.

This parameter uses the following PHONEBUTTONMODE_ constants:

- PHONEBUTTONMODE_CALL - The button is assigned to a call appearance.
- PHONEBUTTONMODE_FEATURE - The button is assigned to requesting features from the switch, such as hold, conference, and transfer.
- PHONEBUTTONMODE_KEYPAD - The button is one of the 12 keypad buttons, ‘0’ through ‘9’, ‘*’, and ‘#’.
- PHONEBUTTONMODE_DISPLAY - The button is a soft button that is associated with the phone display. A phone set can have zero or more display buttons.

dwParam3

Specifies whether this is a button-down event or a button-up event. This parameter uses the following PHONEBUTTONSTATE_ constants:

- PHONEBUTTONSTATE_UP - The button is in the up state.
- PHONEBUTTONSTATE_DOWN - The button is in the down state (pressed down).

- PHONEBUTTONSTATE_UNKNOWN - The up or down state of the button is not known at this time and may be known later.
- PHONEBUTTONSTATE_UNAVAIL - The service provider does not know the up or down state of the button, and the state cannot become known at a future time.

Button ID values of zero through 11 map to the keypad buttons as defined by TAPI. Values above 11 map to line and feature buttons. The low-order part of the DWORD specifies the feature. The high-order part of the DWORD specifies the instance number of that feature. Table 5-7 lists all possible values for the low-order part of the DWORD that corresponds to the feature.

Use the following expression to make the button ID:

ButtonID = (instance << 16) | featureID

Table 5-7 lists the valid phone button values.

Table 5-7 Phone Button Values

Value	Feature	Has Instance	Button Mode
0	Keypad button 0	No	Keypad
1	Keypad button 1	No	Keypad
2	Keypad button 2	No	Keypad
3	Keypad button 3	No	Keypad
4	Keypad button 4	No	Keypad
5	Keypad button 5	No	Keypad
6	Keypad button 6	No	Keypad
7	Keypad button 7	No	Keypad
8	Keypad button 8	No	Keypad
9	Keypad button 9	No	Keypad
10	Keypad button ‘*’	No	Keypad
11	Keypad button ‘#’	No	Keypad
12	Last Number Redial	No	Feature
13	Speed Dial	Yes	Feature
14	Hold	No	Feature
15	Transfer	No	Feature
16	Forward All (for line one)	No	Feature
17	Forward Busy (for line one)	No	Feature
18	Forward No Answer (for line one)	No	Feature
19	Display	No	Feature
20	Line	Yes	Call
21	Chat (for line one)	No	Feature
22	Whiteboard (for line one)	No	Feature
23	Application Sharing (for line one)	No	Feature
24	T120 File Transfer (for line one)	No	Feature

Table 5-7 Phone Button Values (continued)

Value	Feature	Has Instance	Button Mode
25	Video (for line one)	No	Feature
26	Voice Mail (for line one)	No	Feature
27	Answer Release	No	Feature
28	Auto-answer	No	Feature
44	Generic Custom Button 1	Yes	Feature
45	Generic Custom Button 2	Yes	Feature
46	Generic Custom Button 3	Yes	Feature
47	Generic Custom Button 4	Yes	Feature
48	Generic Custom Button 5	Yes	Feature

PHONE_CLOSE

The PHONE_CLOSE message gets sent when an open phone device is forcibly closed as part of resource reclamation. The device handle is no longer valid after this message is sent.

Function Details

```
PHONE_CLOSE
hPhone = (HPHONE) hPhoneDevice;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) 0;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Parameters

hPhone

A handle to the open phone device that was closed. The handle is no longer valid after this message is sent.

dwCallbackInstance

The callback instance of the application that is provided on an open phone device.

dwParam1 is not used.

dwParam2 is not used.

dwParam3 is not used.

PHONE_CREATE

The PHONE_CREATE message gets sent to inform applications of the creation of a new phone device.

Note CTI Manager cluster support, extension mobility, change notification, and user addition to the directory can generate PHONE_CREATE events.

Function Details

```
PHONE_CREATE
hPhone = (HPHONE) hPhoneDevice;
dwCallbackInstance = (DWORD) 0;
dwParam1 = (DWORD) idDevice;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Parameters

hPhone is not used.

dwCallbackInstance is not used.

dwParam1

The dwDeviceID of the newly created device.

dwParam2 is not used.

dwParam3 is not used.

PHONE_REMOVE

The PHONE_REMOVE message gets sent to inform an application of the removal (deletion from the system) of a phone device. Generally, this method is not used for temporary removals, such as extraction of PCMCIA devices, but only for permanent removals in which the service provider would no longer report the device, if TAPI were reinitialized.

Note CTI Manager cluster support, extension mobility, change notification, and user deletion from the directory can generate PHONE_REMOVE events.

Function Details

```
PHONE_REMOVE
dwDevice = (DWORD) 0;
dwCallbackInstance = (DWORD) 0;
dwParam1 = (DWORD) dwDeviceID;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Parameters

dwDevice is reserved. Set to zero.

dwCallbackInstance is reserved. Set to zero.

dwParam1

Identifier of the phone device that was removed.

dwParam2 is reserved. Set to zero.

dwParam3 is reserved. Set to zero.

PHONE_REPLY

The TAPI PHONE_REPLY message gets sent to an application to report the results of function call that completed asynchronously.

Function Details

```
PHONE_REPLY
hPhone = (HPHONE) 0;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) idRequest;
dwParam2 = (DWORD) Status;
dwParam3 = (DWORD) 0;
```

Parameters

hPhone is not used.

dwCallbackInstance

Returns the application callback instance.

dwParam1

The request identifier for which this is the reply.

dwParam2

The success or error indication. The application should cast this parameter into a LONG. Zero indicates success; a negative number indicates an error.

dwParam3 is not used.

PHONE_STATE

TAPI sends the PHONE_STATE message to an application whenever the status of a phone device changes.

Function Details

```
PHONE_STATE
hPhone = (HPHONE) hPhoneDevice;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) PhoneState;
dwParam2 = (DWORD) PhoneStateDetails;
dwParam3 = (DWORD) 0;
```

Parameters

hPhone

A handle to the phone device.

dwCallbackInstance

The callback instance that is provided when the phone device is opened for this application.

dwParam1

The phone state that changed. This parameter uses the following PHONESTATE_ constants:

- PHONESTATE_OTHER - Phone-status items other than the following ones changed. The application should check the current phone status to determine which items changed.
- PHONESTATE_CONNECTED - The connection between the phone device and TAPI was just made. This happens when TAPI is first invoked or when the wire that connects the phone to the computer is plugged in while TAPI is active.
- PHONESTATE_DISCONNECTED - The connection between the phone device and TAPI just broke. This happens when the wire that connects the phone set to the computer is unplugged while TAPI is active.
- PHONESTATE_OWNER - The number of owners for the phone device changed.
- PHONESTATE_MONITORS - The number of monitors for the phone device changed.
- PHONESTATE_DISPLAY - The display of the phone changed.
- PHONESTATE_LAMP - A lamp of the phone changed.
- PHONESTATE_RINGMODE - The ring mode of the phone changed.
- PHONESTATE_ HANDSETHOOKSWITCH - The hookswitch state changed for this speakerphone.
- PHONESTATE_REINIT - Items changed in the configuration of phone devices. To become aware of these changes (as with the appearance of new phone devices), the application should reinitialize its use of TAPI. The hDevice parameter of the PHONE_STATE message stays NULL for this state change as it applies to any of the phones in the system.
- PHONESTATE_REMOVED - Indicates that the device is being removed from the system by the service provider (most likely through user action, through a control panel or similar utility). Normally, a PHONE_CLOSE message on the device immediately follows a PHONE_STATE message with this value. Subsequent attempts to access the device prior to TAPI being reinitialized result in PHONEERR_NODEVICE being returned to the application. If a service provider sends a PHONE_STATE message that contains this value to TAPI, TAPI passes it along to applications that negotiated TAPI version 1.4 or later; applications that negotiated a previous API version do not receive any notification.

dwParam2

Phone state-dependent information that details the status change. This parameter is not used if multiple flags are set in dwParam1 because multiple status items get changed. The application should invoke phoneGetStatus to obtain a complete set of information.

Parameter dwparam2 can comprise one of PHONESTATE_LAMP, PHONESTATE_DISPLAY, PHONESTATE_HANDSETHOOKSWITCH, or PHONESTATE_RINGMODE. Because the Cisco Unified TSP cannot differentiate among hook switches for handsets, headsets, or speaker, the PHONESTATE_HANDSETHOOKSWITCH value always gets used for hook switches.

If dwparam2 is PHONESTATE_LAMP, dwparam2 is the button ID that the PHONE_BUTTON message defines.

If dwParam1 is PHONESTATE_OWNER, dwParam2 contains the new number of owners.

If dwParam1 is PHONESTATE_MONITORS, dwParam2 contains the new number of monitors.

If dwParam1 is PHONESTATE_LAMP, dwParam2 contains the button/lamp identifier of the lamp that changed.

If dwParam1 is PHONESTATE_RINGMODE, dwParam2 contains the new ring mode.

If dwParam1 is PHONESTATE_HANDSET, SPEAKER, or HEADSET, dwParam2 contains the new hookswitch mode of that hookswitch device. This parameter uses the following PHONEHOOKSWITCHMODE_ constants:

- PHONEHOOKSWITCHMODE_ONHOOK - The microphone and speaker both remain on hook for this device.
- PHONEHOOKSWITCHMODE_MICSPEAKER - The microphone and speaker both remain active for this device. The Cisco Unified TSP cannot distinguish among handsets, headsets, or speakers, so this value gets sent when the device is off hook.

dwParam3

The TAPI specification specifies that dwparam3 is zero; however, the Cisco Unified TSP will send the new lamp state to the application in dwparam3 to avoid the call to phoneGetLamp to obtain the state when dwparam2 is PHONESTATE_LAMP.

TAPI Phone Structures

This section describes the TAPI phone structures that Cisco Unified TSP supports:

Table 5-8 TAPI Phone Structures

TAPI Phone Structure
PHONECAPS Structure
PHONEINITIALIZEEXPARAMS
PHONEMESSAGE
PHONESTATUS
VARSTRING

PHONECAPS Structure

This section lists the Cisco-set attributes for each member of the PHONECAPS structure. If the value of a structure member is device, line, or call specific, the list gives the value for each condition.

Members

dwProviderInfoSize

dwProviderInfoOffset

"Cisco Unified TSPxxx.TSP: Cisco IP PBX Service Provider Ver. X.X(x.x)" where the text before the colon specifies the file name of the TSP, and the text after "Ver. " specifies the version of the TSP.

dwPhoneInfoSize

dwPhoneInfoOffset

"DeviceType:[type]" where type specifies the device type that is specified in the Cisco Unified Communications Manager database.

dwPermanentPhoneID

dwPhoneNameSize

dwPhoneNameOffset

"Cisco Phone: [deviceName]" where deviceName specifies the name of the device in the Cisco Unified Communications Manager database.

dwStringFormat

STRINGFORMAT_ASCII

dwPhoneStates

PHONESTATE_OWNER |

PHONESTATE_MONITORS |

PHONESTATE_DISPLAY | (Not set for CTI Route Points)

PHONESTATE_LAMP | (Not set for CTI Route Points)

PHONESTATE_RESUME |

PHONESTATE_REINIT |

PHONESTATE_SUSPEND

dwHookSwitchDevs

PHONEHOOKSWITCHDEV_HANDSET (Not set for CTI Route Points)

dwHandsetHookSwitchModes

PHONEHOOKSWITCHMODE_ONHOOK | (Not set for CTI Route Points)

PHONEHOOKSWITCHMODE_MICSPEAKER | (Not set for CTI Route Points)

PHONEHOOKSWITCHMODE_UNKNOWN (Not set for CTI Route Points)

dwDisplayNumRows (Not set for CTI Route Points)

1

dwDisplayNumColumns

20 (Not set for CTI Route Points)

dwNumRingModes

3 (Not set for CTI Route Points)

dwPhoneFeatures (Not set for CTI Route Points)

PHONEFEATURE_GETDISPLAY |

PHONEFEATURE_GETLAMP |

PHONEFEATURE_GETRING |

PHONEFEATURE_SETDISPLAY |

PHONEFEATURE_SETLAMP

dwMonitoredHandsetHookSwitchModes

PHONEHOOKSWITCHMODE_ONHOOK | (Not set for CTI Route Points)

PHONEHOOKSWITCHMODE_MICSPEAKER (Not set for CTI Route Points)

PHONEINITIALIZEEXPARAMS

The PHONEINITIALIZEEXPARAMS structure contains parameters that are used to establish the association between an application and TAPI; for example, the application selected event notification mechanism. The phoneInitializeEx function uses this structure.

Structure Details

```
typedef struct phoneinitializeexparams_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;
  DWORD  dwOptions;
  union
  {
    HANDLE hEvent;
    HANDLE hCompletionPort;
  } Handles;
  DWORD dwCompletionKey;
} PHONEINITIALIZEEXPARAMS, FAR *LPPHONEINITIALIZEEXPARAMS;
```

Members

dwTotalSize

The total size, in bytes, that is allocated to this data structure.

dwNeededSize

The size, in bytes, for this data structure that is needed to hold all the returned information.

dwUsedSize

The size, in bytes, of the portion of this data structure that contains useful information.

dwOptions

One of the PHONEINITIALIZEEXOPTION_ Constants. Specifies the event notification mechanism that the application wants to use.

hEvent

If dwOptions specifies PHONEINITIALIZEEXOPTION_USEEVENT, TAPI returns the event handle in this member.

hCompletionPort

If dwOptions specifies PHONEINITIALIZEEXOPTION_USECOMPLETIONPORT, the application must specify, in this member, the handle of an existing completion port that is opened by using CreateIoCompletionPort.

dwCompletionKey

If dwOptions specifies PHONEINITIALIZEEXOPTION_USECOMPLETIONPORT, the application must specify in this field a value that is returned through the lpCompletionKey parameter of GetQueuedCompletionStatus to identify the completion message as a telephony message.

PHONEMESSAGE

The PHONEMESSAGE structure contains the next message that is queued for delivery to the application. The phoneGetMessage function returns the following structure.

Structure Details

```
typedef struct phonemessage_tag {
  DWORD  hDevice;
  DWORD  dwMessageID;
  DWORD_PTR  dwCallbackInstance;
  DWORD_PTR  dwParam1;
  DWORD_PTR  dwParam2;
  DWORD_PTR  dwParam3;
} PHONEMESSAGE, FAR *LPPHONEMESSAGE;
```

Members

hDevice

A handle to a phone device.

dwMessageID

A phone message.

dwCallbackInstance

Instance data that is passed back to the application, which the application specified in phoneInitializeEx. TAPI does not interpret DWORD.

dwParam1

A parameter for the message.

dwParam2

A parameter for the message.

dwParam3

A parameter for the message.

Further Details

For details on the parameter values that are passed in this structure, see "TAPI Phone Messages."

PHONESTATUS

The PHONESTATUS structure describes the current status of a phone device. The phoneGetStatus and TSPI_phoneGetStatus functions return this structure.

Device-specific extensions should use the DevSpecific (dwDevSpecificSize and dwDevSpecificOffset) variably sized area of this data structure.

Note The dwPhoneFeatures member is available only to applications that open the phone device with an API version of 2.0 or later.

Structure Details

```
typedef struct phonestatus_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;
    DWORD  dwStatusFlags;
    DWORD  dwNumOwners;
    DWORD  dwNumMonitors;
    DWORD  dwRingMode;
    DWORD  dwRingVolume;
    DWORD  dwHandsetHookSwitchMode;
    DWORD  dwHandsetVolume;
    DWORD  dwHandsetGain;
    DWORD  dwSpeakerHookSwitchMode;
    DWORD  dwSpeakerVolume;
    DWORD  dwSpeakerGain;
    DWORD  dwHeadsetHookSwitchMode;
    DWORD  dwHeadsetVolume;
    DWORD  dwHeadsetGain;
    DWORD  dwDisplaySize;
    DWORD  dwDisplayOffset;
    DWORD  dwLampModesSize;
    DWORD  dwLampModesOffset;
    DWORD  dwOwnerNameSize;
    DWORD  dwOwnerNameOffset;
    DWORD  dwDevSpecificSize;
    DWORD  dwDevSpecificOffset;
    DWORD  dwPhoneFeatures;
}   PHONESTATUS, FAR *LPPHONESTATUS;
```

Members

dwTotalSize

The total size, in bytes, that is allocated to this data structure.

dwNeededSize

The size, in bytes, for this data structure that is needed to hold all the returned information.

dwUsedSize

The size, in bytes, of the portion of this data structure that contains useful information.

dwStatusFlags

Provides a set of status flags for this phone device. This member uses one of the PHONESTATUSFLAGS_ Constants.

dwNumOwners

The number of application modules with owner privilege for the phone.

dwNumMonitors

The number of application modules with monitor privilege for the phone.

dwRingMode

The current ring mode of a phone device.

dwRingVolume

0x8000

dwHandsetHookSwitchMode

The current hookswitch mode of the phone handset. PHONEHOOKSWITCHMODE_UNKNOWN

dwHandsetVolume

0

dwHandsetGain

0

dwSpeakerHookSwitchMode

The current hookswitch mode of the phone speakerphone. PHONEHOOKSWITCHMODE_UNKNOWN

dwSpeakerVolume

0

dwSpeakerGain

0

dwHeadsetHookSwitchMode

The current hookswitch mode of the phone's headset. PHONEHOOKSWITCHMODE_UNKNOWN

dwHeadsetVolume

0

dwHeadsetGain

0

dwDisplaySize

dwDisplayOffset

0

dwLampModesSize

dwLampModesOffset

0

dwOwnerNameSize

dwOwnerNameOffset

The size, in bytes, of the variably sized field that contains the name of the application that is the current owner of the phone device and the offset, in bytes, from the beginning of this data structure. The name is the application name that the application provides when it invokes with phoneInitialize or phoneInitializeEx. If no application name was supplied, the application's filename is used instead. If the phone currently has no owner, dwOwnerNameSize is zero.

dwDevSpecificSize

dwDevSpecificOffset

Application can send XSI data to phone by using DeviceDataPassThrough device-specific extension. Phone can pass back data to Application. The data is returned as part of this field. The format of the data is as follows:

struct PhoneDevSpecificData

```
{
    DWORD m_DeviceDataSize ; // size of device data
```

```
        DWORD m_DeviceDataOffset ; // offset from PHONESTATUS
        structure
        // this will follow the actual variable length device data.
    }
```

dwPhoneFeatures

The application negotiates an extension version >= 0x00020000. The following features are supported:

- PHONEFEATURE_GETDISPLAY
- PHONEFEATURE_GETLAMP
- PHONEFEATURE_GETRING
- PHONEFEATURE_SETDISPLAY
- PHONEFEATURE_SETLAMP

VARSTRING

The VARSTRING structure returns variably sized strings. The line device class and the phone device class both use it.

Note No extensibility exists with VARSTRING.

Structure Details

```
typedef struct varstring_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;
  DWORD  dwStringFormat;
  DWORD  dwStringSize;
  DWORD  dwStringOffset;
} VARSTRING, FAR *LPVARSTRING;
```

Members

dwTotalSize

The total size, in bytes, that is allocated to this data structure.

dwNeededSize

The size, in bytes, for this data structure that is needed to hold all the returned information.

dwUsedSize

The size, in bytes, of the portion of this data structure that contains useful information.

dwStringFormat

The format of the string. This member uses one of the STRINGFORMAT_ Constants.

dwStringSize

dwStringOffset

The size, in bytes, of the variably sized device field that contains the string information and the offset, in bytes, from the beginning of this data structure.

If a string cannot be returned in a variable structure, the dwStringSize and dwStringOffset members get set in one of the following ways:

dwStringSize and dwStringOffset members both get set to zero.

dwStringOffset gets set to nonzero and dwStringSize gets set to zero.

dwStringOffset gets set to nonzero, dwStringSize gets set to 1, and the byte at the given offset gets set to zero.]

Wave Functions

The AVAudio32.dll implements the wave interfaces to the Cisco wave drivers. The system supports all APIs for input and output waveform devices.

.

Table 5-9 Wave Functions

Wave Functions
waveInAddBuffer
waveInClose
waveInGetID
waveInGetPosition
waveInOpen
waveInPrepareHeader
waveInReset
waveInStart
waveInUnprepareHeader
waveOutClose
waveOutPrepareHeader
waveOutGetDevCaps
waveOutGetID
waveOutGetPosition
waveOutOpen
waveOutPrepareHeader
waveOutReset
waveOutUnprepareHeader
waveOutWrite

waveInAddBuffer

The waveInAddBuffer function sends an input buffer to the given waveform-audio input device. When the buffer is filled, the application receives notification.

Function Details

```
MMRESULT waveInAddBuffer(
  HWAVEIN hwi,
  LPWAVEHDR pwh,
  UINT cbwh
);
```

Parameters

hwi

Handle of the waveform-audio input device.

pwh

Address of a WAVEHDR structure that identifies the buffer.

cbwh

Size, in bytes, of the WAVEHDR structure.

waveInClose

The waveInClose function closes the given waveform-audio input device.

Function Details

```
MMRESULT waveInClose(
  HWAVEIN hwi
);
```

Parameter

hwi

Handle of the waveform-audio input device. If the function succeeds, the handle no longer remains valid after this call.

waveInGetID

The waveInGetID function gets the device identifier for the given waveform-audio input device.

This function gets supported for backward compatibility. New applications can cast a handle of the device rather than retrieving the device identifier.

Function Details

```
MMRESULT waveInGetID(
  HWAVEIN hwi,
  LPUINT puDeviceID
);
```

Parameters

hwi

Handle of the waveform-audio input device.

puDeviceID

Address of a variable to be filled with the device identifier.

waveInGetPosition

The waveInGetPosition function retrieves the current input position of the given waveform-audio input device.

Function Details

```
MMRESULT waveInGetPosition(
  HWAVEIN hwi,
  LPMMTIME pmmt,
  UINT cbmmt
);
```

Parameters

hwi

Handle of the waveform-audio input device.

pmmt

Address of the MMTIME structure.

cbmmt

Size, in bytes, of the MMTIME structure.

waveInOpen

The waveInOpen function opens the given waveform-audio input device for recording.

Function Details

```
MMRESULT waveInOpen(
  LPHWAVEIN phwi,
  UINT uDeviceID,
  LPWAVEFORMATEX pwfx,
  DWORD dwCallback,
  DWORD dwCallbackInstance,
  DWORD fdwOpen
);
```

Parameters

phwi

Address that is filled with a handle that identifies the open waveform-audio input device. Use this handle to identify the device when calling other waveform-audio input functions. This parameter can be NULL if WAVE_FORMAT_QUERY is specified for fdwOpen.HDR structure.

uDeviceID

Identifier of the waveform-audio input device to open. It can be either a device identifier or a handle of an open waveform-audio input device. You can use the following flag instead of a device identifier:

WAVE_MAPPER - The function selects a waveform-audio input device that is capable of recording in the specified format.

pwfx

Address of a WAVEFORMATEX structure that identifies the desired format for recording waveform-audio data. You can free this structure immediately after waveInOpen returns.

Note The formats that the TAPI Wave Driver supports include a 16-bit PCM at 8000 Hz, 8-bit mulaw at 8000 Hz, and 8-bit alaw at 8000 Hz.

dwCallback

Address of a fixed callback function, an event handle, a handle to a window, or the identifier of a thread to be called during waveform-audio recording to process messages that are related to the progress of recording. If no callback function is required, this value can specify zero. For more information on the callback function, see waveInProc in the TAPI API.

dwCallbackInstance

User-instance data that is passed to the callback mechanism. This parameter is not used with the window callback mechanism.

fdwOpen

Flags for opening the device. The following values definitions apply:

- CALLBACK_EVENT - The dwCallback parameter specifies an event handle.
- CALLBACK_FUNCTION - The dwCallback parameter specifies a callback procedure address.
- CALLBACK_NULL - No callback mechanism. This represents the default setting.
- CALLBACK_THREAD - The dwCallback parameter specifies a thread identifier.
- CALLBACK_WINDOW - The dwCallback parameter specifies a window handle.
- WAVE_FORMAT_DIRECT - If this flag is specified, the A driver does not perform conversions on the audio data.
- WAVE_FORMAT_QUERY - The function queries the device to determine whether it supports the given format, but it does not open the device.
- WAVE_MAPPED - The uDeviceID parameter specifies a waveform-audio device to which the wave mapper maps.

waveInPrepareHeader

The waveInPrepareHeader function prepares a buffer for waveform-audio input.

Function Details

```
MMRESULT waveInPrepareHeader(
  HWAVEIN hwi,
  LPWAVEHDR pwh,
  UINT cbwh
);
```

Parameters

hwi

Handle of the waveform-audio input device.

pwh

Address of a WAVEHDR structure that identifies the buffer to be prepared.

cbwh

Size, in bytes, of the WAVEHDR structure.

waveInReset

The waveInReset function stops input on the given waveform-audio input device and resets the current position to zero. All pending buffers get marked as done and get returned to the application.

Function Details

```
MMRESULT waveInReset(
  HWAVEIN hwi
);
```

Parameter

hwi

Handle of the waveform-audio input device.

waveInStart

The waveInStart function starts input on the given waveform-audio input device.

Function Details

```
MMRESULT waveInStart(
  HWAVEIN hwi
);
```

Parameter

hwi

Handle of the waveform-audio input device.

waveInUnprepareHeader

The waveInUnprepareHeader function cleans up the preparation that the waveInPrepareHeader function performs. This function must be called after the device driver fills a buffer and returns it to the application. You must call this function before freeing the buffer.

Function Details

```
MMRESULT waveInUnprepareHeader(
  HWAVEIN hwi,
  LPWAVEHDR pwh,
  UINT cbwh
);
```

Parameters

hwi

Handle of the waveform-audio input device.

pwh

Address of a WAVEHDR structure that identifies the buffer to be cleaned up.

cbwh

Size, in bytes, of the WAVEHDR structure.

waveOutClose

The waveOutClose function closes the given waveform-audio output device.

Function Details

```
MMRESULT waveOutClose(
  HWAVEOUT hwo
);
```

Parameter

hwo

Handle of the waveform-audio output device. If the function succeeds, the handle no longer remains valid after this call.

waveOutGetDevCaps

The waveOutGetDevCaps function retrieves the capabilities of a given waveform-audio output device.

Function Details

```
MMRESULT waveOutGetDevCaps(
```

```
  UINT uDeviceID,
  LPWAVEOUTCAPS pwoc,
  UINT cbwoc
);
```

Parameters

uDeviceID

Identifier of the waveform-audio output device. It can be either a device identifier or a handle of an open waveform-audio output device.

pwoc

Address of a WAVEOUTCAPS structure that is to be filled with information about the capabilities of the device.

cbwoc

Size, in bytes, of the WAVEOUTCAPS structure.

waveOutGetID

The waveOutGetID function retrieves the device identifier for the given waveform-audio output device.

This function gets supported for backward compatibility. New applications can cast a handle of the device rather than retrieving the device identifier.

Function Details

```
MMRESULT waveOutGetID(
  HWAVEOUT hwo,
  LPUINT puDeviceID
);
```

Parameters

hwo

Handle of the waveform-audio output device.

puDeviceID

Address of a variable to be filled with the device identifier.

waveOutGetPosition

The waveOutGetPosition function retrieves the current playback position of the given waveform-audio output device.

Function Details

```
MMRESULT waveOutGetPosition(
  HWAVEOUT hwo,
  LPMMTIME pmmt,
  UINT cbmmt
```

```
);
```

Parameters

hwo

Handle of the waveform-audio output device.

pmmt

Address of an MMTIME structure.

cbmmt

Size, in bytes, of the MMTIME structure.

waveOutOpen

The waveOutOpen function opens the given waveform-audio output device for playback.

Function Details

```
MMRESULT waveOutOpen(
  LPHWAVEOUT phwo,
  UINT uDeviceID,
  LPWAVEFORMATEX pwfx,
  DWORD dwCallback,
  DWORD dwCallbackInstance,
  DWORD fdwOpen
);
```

Parameters

phwo

Address that is filled with a handle that identifies the open waveform-audio output device. Use the handle to identify the device when other waveform-audio output functions are called. This parameter might be NULL if the WAVE_FORMAT_QUERY flag is specified for fdwOpen.

uDeviceID

Identifier of the waveform-audio output device to open. It can be either a device identifier or a handle of an open waveform-audio input device. You can use the following flag instead of a device identifier:

WAVE_MAPPER - The function selects a waveform-audio output device that is capable of playing the given format.

pwfx

Address of a WAVEFORMATEX structure that identifies the format of the waveform-audio data to be sent to the device. You can free this structure immediately after passing it to waveOutOpen.

Note The formats that the TAPI Wave Driver supports include 16-bit PCM at 8000 Hz, 8-bit mulaw at 8000 Hz, and 8-bit alaw at 8000 Hz.

dwCallback

Address of a fixed callback function, an event handle, a handle to a window, or the identifier of a thread to be called during waveform-audio playback to process messages that are related to the progress of the playback. If no callback function is required, this value can specify zero. For more information on the callback function, see waveOutProc in the TAPI API.

dwCallbackInstance

User-instance data that is passed to the callback mechanism. This parameter is not used with the window callback mechanism.

fdwOpen

Flags for opening the device. The following value definitions apply:

- CALLBACK_EVENT - The dwCallback parameter represents an event handle.
- CALLBACK_FUNCTION - The dwCallback parameter specifies a callback procedure address.
- CALLBACK_NULL - No callback mechanism. This value specifies the default setting.
- CALLBACK_THREAD - The dwCallback parameter represents a thread identifier.
- CALLBACK_WINDOW - The dwCallback parameter specifies a window handle.
- WAVE_ALLOWSYNC - If this flag is specified, a synchronous waveform-audio device can be opened. If this flag is not specified while a synchronous driver is opened, the device will fail to open.
- WAVE_FORMAT_DIRECT - If this flag is specified, the ACM driver does not perform conversions on the audio data.
- WAVE_FORMAT_QUERY - If this flag is specified, waveOutOpen queries the device to determine whether it supports the given format, but the device does not actually open.
- WAVE_MAPPED - If this flag is specified, the uDeviceID parameter specifies a waveform-audio device to which the wave mapper maps.

waveOutPrepareHeader

The waveOutPrepareHeader function prepares a waveform-audio data block for playback.

Function Details

```
MMRESULT waveOutPrepareHeader(
  HWAVEOUT hwo,
  LPWAVEHDR pwh,
  UINT cbwh
);
```

Parameters

hwo

Handle of the waveform-audio output device.

pwh

Address of a WAVEHDR structure that identifies the data block to be prepared.

cbwh

Size, in bytes, of the WAVEHDR structure.

waveOutReset

The waveOutReset function stops playback on the given waveform-audio output device and resets the current position to zero. All pending playback buffers get marked as done and get returned to the application.

Function Details

```
MMRESULT waveOutReset(
  HWAVEOUT hwo
);
```

Parameter

hwo

Handle of the waveform-audio output device.

waveOutUnprepareHeader

The waveOutUnprepareHeader function cleans up the preparation that the waveOUtPrepareHeader function performs. Ensure this function is called after the device driver is finished with a data block. You must call this function before freeing the buffer.

Function Details

```
MMRESULT waveOutUnprepareHeader(
  HWAVEOUT hwo,
  LPWAVEHDR pwh,
  UINT cbwh
);
```

Parameters

hwo

Handle of the waveform-audio output device.

pwh

Address of a WAVEHDR structure that identifies the data block to be cleaned up.

cbwh

Size, in bytes, of the WAVEHDR structure.

waveOutWrite

The waveOutWrite function sends a data block to the given waveform-audio output device.

Function Details

```
MMRESULT waveOutWrite(
  HWAVEOUT hwo,
  LPWAVEHDR pwh,
  UINT cbwh
);
```

Parameters

hwo

Handle of the waveform-audio output device.

pwh

Address of a WAVEHDR structure that contains information about the data block.

cbwh

Size, in bytes, of the WAVEHDR structure.

CHAPTER 6

Cisco Device-Specific Extensions

This chapter describes the Cisco device-specific TAPI extensions. CiscoLineDevSpecific and the CCiscoPhoneDevSpecific class represent the parent class. This chapter describes how to invoke the Cisco device-specific TAPI extensions with the lineDevSpecific function. It also describes a set of classes that you can use when you call phoneDevSpecific. It contains the following sections:

- Cisco Line Device Specific Extensions, page 6-1
- Cisco Line Device Feature Extensions, page 6-49
- Cisco Phone Device-Specific Extensions, page 6-53
- Messages, page 6-58

Cisco Line Device Specific Extensions

Table 6-1 lists the subclasses of Cisco Line Device-Specific Extensions. This section contains all of the extensions in the table and descriptions of the following data structures:

- LINEDEVCAPS, page 6-3
- LINECALLINFO, page 6-6
- LINEDEVSTATUS, page 6-14

Table 6-1 Cisco Line Device-Specific Extensions

Cisco Functions	Synopsis
CCiscoLineDevSpecific	The CCiscoLineDevSpecific class specifies the parent class to the following classes.
Message Waiting	The CCiscoLineDevSpecificMsgWaiting class turns the message waiting lamp on or off for the line that the hLine parameter specifies.
Message Waiting Dirn	The CCiscoLineDevSpecificMsgWaiting class turns the message waiting lamp on or off for the line that a parameter specifies and remains independent of the hLine parameter.
Audio Stream Control	The CCiscoLineDevSpecificUserControlRTPStream class controls the audio stream for a line.
Set Status Messages	The CCiscoLineDevSpecificSetStatusMsgs class controls the reporting of certain line device specific messages for a line. The application receives LINE_DEVSPECIFIC messages to signal when to stop and start streaming RTP audio.

Table 6-1 Cisco Line Device-Specific Extensions (continued)

Cisco Functions	Synopsis
Swap-Hold/SetupTransfer	Cisco Unified TSP 4.0 and later do not support this function. The CCiscoLineDevSpecificSwapHoldSetupTransfer class performs a setupTransfer between a call that is in CONNECTED state and a call that is in ONHOLD state. This function will change the state of the connected call to ONHOLDPENDTRANSFER state and the ONHOLD call to CONNECTED state. This action will then allow a completeTransfer to be performed on the two calls.
Redirect Reset Original Called ID	The CCiscoLineDevSpecificRedirectResetOrigCalled class gets used to redirect a call to another party while resetting the original called ID of the call to the destination of the redirect.
Port Registration per Call	The CciscoLineDevSpecificPortRegistrationPerCall class gets used to register a CTI port or route point for the Dynamic Port Registration feature, which allows applications to specify the IP address and UDP port number on a call-by-call basis.
Setting RTP Parameters for Call	The CciscoLineDevSpecificSetRTPParamsForCall class sets the IP address and UDP port number for the specified call.
Redirect Set Original Called ID	Use the CciscoLineDevSpecificSetOrigCalled class to redirect a call to another party while setting the original called ID of the call to any other party.
Join	Use the CciscoLineDevSpecificJoin class to join two or more calls into one conference call.
Set User SRTP Algorithm IDs	Use the CciscoLineDevSpecificUserSetSRTPAlgorithmID class to allow application to set SRTP algorithm IDs. You should use this class after lineopen and before CCiscoLineDevSpecificSetRTPParamsForCall or CCiscoLineDevSpecificUserControlRTPStream
Explicit Acquire	Use the CciscoLineDevSpecificAcquire class to explicitly acquire any CTI Controllable device in the Cisco Unified Communications Manager system, which needs to be opened in Super Provider mode.
Explicit De-Acquire	Use the CciscoLineDevSpecificDeacquire class to explicitly de-acquire any CTI controllable device in the Cisco Unified Communications Manager system.
Redirect FAC CMC	Use the CCiscoLineDevSpecificRedirectFACCMC class to redirect a call to another party while including a FAC, CMC, or both.
Blind Transfer FAC CMC	Use the CCiscoLineDevSpecificBlindTransferFACCMC class to blind transfer a call to another party while including a FAC, CMC, or both.
CTI Port Third Party Monitor	Use the CCiscoLineDevSpecificCTIPortThirdPartyMonitor class to open a CTI port in third-party mode.
Send Line Open	Use the CciscoLineDevSpecificSendLineOpen class to trigger actual line open from TSP side. Use this for delayed open mechanism.
Start Call Monitoring	Use CCiscoLineDevSpecificStartCallMonitoringReq to allow applications to send a start monitoring request for the active call on a line.

Table 6-1 Cisco Line Device-Specific Extensions (continued)

Cisco Functions	Synopsis
Start Call Recording	Use CCiscoLineDevSpecificStartCallRecordingReq to allow applications to send a recording request for the active call on that line.
StopCall Recording	Use CCiscoLineDevSpecificStopCallRecordingReq to allow applications to stop recording a call on that line.
Set Intercom SpeedDial	Use the CciscoLineDevSpecificSetIntercomSpeedDial class to allow application to set or reset SpeedDial/Label on an intercom line.
Intercom Talk Back	Use the CCiscoLineDevSpecificTalkBack class to allow application to initiate talk back on a incoming Intercom call on an Intercom line.
Redirect with Feature Priority	Use the CciscoLineRedirectWithFeaturePriority class to enable an application to redirect calls with specified priority.

LINEDEVCAPS

Cisco TSP implements several line device-specific extensions and uses the DevSpecific (dwDevSpecificSize and dwDevSpecificOffset) variably sized area of the LINEDEVCAPS data structure for those extensions. The the Cisco_LineDevCaps_Ext structure in the CiscoLineDevSpecificMsg.h header file defines the DevSpecific area layout. Cisco TSP organizes the data in that structure based on the extension version in which the data was introduced:

```
//   LINEDEVCAPS Dev Specific extention   //
typedef struct Cisco_LineDevCaps_Ext
{
    Cisco_LineDevCaps_Ext00030000  ext30;
    Cisco_LineDevCaps_Ext00060000  ext60;
    Cisco_LineDevCaps_Ext00070000  ext70;
    Cisco_LineDevCaps_Ext00080000  ext80;
} CISCO_LINEDEVCAPS_EXT;
```

For a specific line device, the extension area will include a portion of this structure starting from the beginning and up to the extension version that an application negotiated.

The individual extension version substructure definitions follow:

```
//    LINEDEVCAPS 00030000  extention    //
typedef struct Cisco_LineDevCaps_Ext00030000
{
    DWORD dwLineTypeFlags;
} CISCO_LINEDEVCAPS_EXT00030000;
//    LINEDEVCAPS 00060000  extention    //
typedef struct Cisco_LineDevCaps_Ext00060000
{
    DWORD dwLocale;
} CISCO_LINEDEVCAPS_EXT00060000;
//    LINEDEVCAPS 00070000  extention    //
typedef struct Cisco_LineDevCaps_Ext00070000
{
    DWORD dwPartitionOffset;
    DWORD dwPartitionSize;
} CISCO_LINEDEVCAPS_EXT00070000;
//    LINEDEVCAPS  00080000  extention    //
typedef struct Cisco_LineDevCaps_Ext00080000
{
    DWORD                   dwLineDevCaps_DevSpecificFlags;        //
LINEFEATURE_DEVSPECIFIC
```

```
    DWORD                  dwLineDevCaps_DevSpecificFeatureFlags; //
LINEFEATURE_DEVSPECIFICFEAT
    RECORD_TYPE_INFO       recordTypeInfo;
    INTERCOM_SPEEDDIAL_INFO intercomSpeedDialInfo;
} CISCO_LINEDEVCAPS_EXT00080000;
```

See the CiscoLineDevSpecificMsg.h header file for additional information on the DevSpecific structure layout and data.

Detail

A

```
typedef struct LineDevCaps_DevSpecificData
{
    DWORD m_DevSpecificFlags;
}LINEDEVCAPS_DEV_SPECIFIC_DATA;
```

Note Be aware that this extension is only available if extension version 3.0 (0x00030000) or higher is negotiated.

B

```
typedef  struct LocaleInfo
{
    DWORD Locale; //This will have the locale info of the device
    DWORD PartitionOffset;
DWORD PartitionSize; //This will have the partition info of the line.
} LOCALE_INFO;
```

Note Be aware that the Locale info is only available along with LINEDEVCAPS_DEV_SPECIFIC_DATA if extension version 6.0 (0x00060000) or higher is negotiated.

C

```
typedef  struct PartitionInfo
{
    DWORD PartitionOffset;
DWORD PartitionSize; //This will have the partition info of the line.
} PARTITION_INFO;
```

Note Be aware that both the Locale and Partition Info is available along with LINEDEVCAPS_DEV_SPECIFIC_DATA if extension version 6.1 (0x00060001) or higher is negotiated.

Parameters

DWORD m_DevSpecificFlags

A bit array that identifies device-specific properties for the line. The bits definition follows:

LINEDEVCAPSDEVSPECIFIC_PARKDN (0x00000001)—Indicates whether this line is a Call Park DN.

Note Be aware that this extension is only available if extension version 3.0 (0x00030000) or higher is negotiated.

DWORD Locale

This entity identifies the locale information for the device. The typical values could be:

```
enum
{
ENGLISH_UNITED_STATES= 1,
FRENCH_FRANCE= 2,
GERMAN_GERMANY= 3,
RUSSIAN_RUSSIAN_FEDERATION= 5,
SPANISH_SPAIN= 6,
ITALIAN_ITALY= 7,
DUTCH_NETHERLANDS= 8,
NORWEGIAN_NORWAY= 9,
PORGUGUESE_PORTUGAL= 10,
SWEDISH_SWEDEN= 11,
DANISH_DENMARK= 12,
JAPANESE_JAPAN= 13,
HUNGARAIN_HUNGARY= 14,
POLISH_POLAND= 15,
GREEK_GREECE= 16,
CHINESE_TAIWAN = 19,
CHINESE_CHINA= 20,
KOREAN_KOREA_REPUBLIC= 21,
FINNISH_FINLAND= 22,
PORTUGUESE_BRAZIL= 23,
CHINESE_HONG_KONG= 24,
SLOVAK_SLOVAKIA= 25,
CZECH_CZECH_REPUBLIC= 26,
BULGARIAN_BULGARIA= 27,
CROATIAN_CROATIA= 28,
SLOVENIAN_SLOVENIA= 29,
ROMANIAN_ROMANIA= 30,
CATALAN_SPAIN= 32,
ENGLISH_UNITED_KINGDOM= 33,
ARABIC_UNITED_ARAB_EMIRATES= 35,
ARABIC_OMAN= 36,
ARABIC_SAUDI_ARABIA= 37,
ARABIC_KUWAIT= 38,
HEBREW_ISRAEL= 39,
SERBIAN_REPUBLIC_OF_SERBIA= 40,
SERBIAN_REPUBLIC_OF_MONTENEGRO= 41,
THAI_THAILAND= 42,
ARABIC_ALGERIA= 47,
ARABIC_BAHRAIN= 48,
ARABIC_EGYPT= 49,
ARABIC_IRAQ= 50,
ARABIC_JORDAN= 51,
ARABIC_LEBANON= 52,
ARABIC_MOROCCO= 53,
ARABIC_QATAR= 54,
ARABIC_TUNISIA= 55,
}
```

LINECALLINFO

Cisco TSP implements several line device-specific extensions and uses the DevSpecific (dwDevSpecificSize and dwDevSpecificOffset) variably sized area of the LINECALLINFO data structure for those extensions. The Cisco_LineCallInfo_Ext structure in the CiscoLineDevSpecificMsg.h header file defines DevSpecific area layout. Cisco TSP organizes the data in that structure based on the extension version in which the data was introduced:

```
//   LINECALLINFO Dev Specific extention   //
typedef struct Cisco_LineCallInfo_Ext
{
    Cisco_LineCallInfo_Ext00060000  ext60;
    Cisco_LineCallInfo_Ext00070000  ext70;
    Cisco_LineCallInfo_Ext00080000  ext80;
} CISCO_LINECALLINFO_EXT;
```

For a specific line device, the extension area includes a portion of this structure starting from the beginning and up to the extension version that an application negotiated.

The individual extension version substructure definitions follow:

```
//    LINECALLINFO 00060000  extention    //
typedef struct Cisco_LineCallInfo_Ext00060000
{
    TSP_UNICODE_PARTY_NAMES  unicodePartyNames;
} CISCO_LINECALLINFO_EXT00060000;
//    LINECALLINFO 00070000  extention    //
typedef struct Cisco_LineCallInfo_Ext00070000
{
    DWORD SRTPKeyInfoStructureOffset;   // offset from base of LINECALLINFO
    DWORD SRTPKeyInfoStructureSize;     // includes variable length data total size
    DWORD SRTPKeyInfoStructureElementCount;
    DWORD SRTPKeyInfoStructureElementFixedSize;
    DWORD DSCPInformationOffset;        // offset from base of LINECALLINFO
    DWORD DSCPInformationSize;          // fixed size of the DSCPInformation structure
    DWORD DSCPInformationElementCount;
    DWORD DSCPInformationElementFixedSize;
    DWORD CallPartitionInfoOffset;      // offset from base of LINECALLINFO
    DWORD CallPartitionInfoSize;        // fixed size of the CallPartitionInformation
structure
    DWORD CallPartitionInfoElementCount;
    DWORD CallPartitionInfoElementFixedSize;
    DWORD ExtendedCallInfoOffset;       //  ===> ExtendedCallInfo { }
    DWORD ExtendedCallInfoSize;         //
    DWORD ExtendedCallInfoElementCount; //
    DWORD ExtendedCallInfoElementSize;  //
} CISCO_LINECALLINFO_EXT00070000;
//    LINEDEVCAPS  00080000  extention    //
typedef struct Cisco_LineDevCaps_Ext00080000
{
    DWORD                   dwLineDevCaps_DevSpecificFlags;        //
LINEFEATURE_DEVSPECIFIC
    DWORD                   dwLineDevCaps_DevSpecificFeatureFlags; //
LINEFEATURE_DEVSPECIFICFEAT
    RECORD_TYPE_INFO        recordTypeInfo;
    INTERCOM_SPEEDDIAL_INFO intercomSpeedDialInfo;
} CISCO_LINEDEVCAPS_EXT00080000;
//    LINECALLINFO 00080001  extension    //
//    --------------------------------
typedef struct Cisco_LineCallInfo_Ext00080001
{
    DWORD CPNInfoOffset;         //array of structure of CPNInfo structure
    DWORD CPNInfoSize;
```

```
    DWORD CPNInfoElementCount;
    DWORD CPNInfoElementFixedSize;
};
```

See the CiscoLineDevSpecificMsg.h header file for additional information on the DevSpecific structure layout and data.

Details

The TSP_Unicode_Party_names structure and SRTP info structure describe the device-specific extensions that the Cisco Unified TSP made to the LINECALLINFO structure. DSCPValueForAudioCalls will contain the DSCP value that CTI sent in the StartTransmissionEvent.

ExtendedCallInfo structure has extra call information. For Cisco Unified Communications Manager Release 7.0(1), the ExtendedCallReason field belongs to the ExtendedCallInfo structure.

CallAttributeInfo contains the information about attributeType (Monitoring, Monitored, Recorder,securityStatus) and PartyInfo (Dn,Partition,DeviceName)

CCMCallID contains CCM Call identifier value.

CallingPartyIPAddress contains the IP address of the calling party if the calling party device supports it.

CallSecurityStatus structure contains the overall security status of the call for two-party call as well as conference call.

```
DWORD TapiCallerPartyUnicodeNameOffset;
DWORD TapiCallerPartyUnicodeNameSize;
DWORDTapiCallerPartyLocale;

DWORD TapiCalledPartyUnicodeNameOffset;
DWORD TapiCalledPartyUnicodeNameSize;
DWORDTapiCalledPartyLocale;

DWORD TapiConnectedPartyUnicodeNameOffset;
DWORD TapiConnectedPartyUnicodeNameSize;
DWORDTapiConnectedPartyLocale;

DWORD TapiRedirectionPartyUnicodeNameOffset;
DWORD TapiRedirectionPartyUnicodeNameSize;
DWORDTapiRedirectionPartyLocale;

DWORD TapiRedirectingPartyUnicodeNameOffset;
DWORD TapiRedirectingPartyUnicodeNameSize;
DWORDTapiRedirectingPartyLocale;

DWORD SRTPKeyInfoStructureOffset; // offset from base of LINECALLINFO
DWORD SRTPKeyInfoStructureSize;// includes variable length data total size
DWORD SRTPKeyInfoStructureElementCount;
DWORD SRTPKeyInfoStructureElementFixedSize;
DWORD DSCPValueInformationOffset;
DWORD DSCPValueInformationSize;
DWORD DSCPValueInformationElementCount;
DWORD DSCPValueInformationElementFixedSize;
DWORD PartitionInformationOffset; // offset from base of LINECALLINFO
DWORD PartitionInformationSize;  // includes variable length data total size
DWORD PartitionInformationElementCount;
DWORD PartitionInformationElementFixedSize;
DWORD ExtendedCallInfoOffset;
DWORD ExtendedCallInfoSize;
DWORD ExtendedCallInfoElementCount;
DWORD ExtendedCallInfoElementSize;
DWORD CallAttrtibuteInfoOffset;
```

```
DWORD CallAttrtibuteInfoSize;
DWORD CallAttrtibuteInfoElementCount;
DWORD CallAttrtibuteInfoElementSize;
DWORD CallingPartyIPAddress;
DWORD CCMCallIDInfoOffset;
DWORD CCMCallIDInfoSize;
DWORD CCMCallIDInfoElementCount;
DWORD CCMCallIDInfoElementFixedSize;
DWORD CallSecurityStatusOffset;
DWORD CallSecurityStatusSize;
DWORD CallSecurityStatusElementCount;
DWORD CallSecurityStatusElementFixedSize;

typedef struct SRTPKeyInfoStructure
{
    SRTPKeyInformation TransmissionSRTPInfo;
    SRTPKeyInformation ReceptionSRTPInfo;
} SRTPKeyInfoStructure;

typedef struct SRTPKeyInformation
{
    DWORDIsSRTPDataAvailable;
    DWORDSecureMediaIndicator;// CiscoSecurityIndicator
    DWORDMasterKeyOffset;
    DWORDMasterKeySize;
    DWORDMasterSaltOffset;
    DWORDMasterSaltSize;
    DWORDAlgorithmID;// CiscoSRTPAlgorithmIDs
    DWORDIsMKIPresent;
    DWORDKeyDerivationRate;
} SRTPKeyInformation;

enum CiscoSRTPAlgorithmIDs
{
    SRTP_NO_ENCRYPTION=0,
    SRTP_AES_128_COUNTER=1
};

enum CiscoSecurityIndicator
{
    SRTP_MEDIA_ENCRYPT_KEYS_AVAILABLE,
    SRTP_MEDIA_ENCRYPT_USER_NOT_AUTH,
    SRTP_MEDIA_ENCRYPT_KEYS_UNAVAILABLE,
    SRTP_MEDIA_NOT_ENCRYPTED
};
```

If isSRTPInfoavailable is set to false, applications should ignore the rest of the information from SRTPKeyInformation.

If MasterKeySize or MasterSlatSize is set to 0, applications should ignore the corresponding MasterKeyOffset or MasterSaltOffset.

```
typedef struct DSCPValueInformation
{
DWORD DSCPValueForAudioCalls;
}

typedef struct  PartitionInformation
{
    DWORD CallerIDPartitionOffset;
    DWORD CallerIDPartitionSize;
    DWORD CalledIDPartitionOffset;
    DWORD CalledIDPartitionSize;
    DWORD ConnecetedIDPartitionOffset;
```

```
    DWORD ConnecetedIDPartitionSize;
    DWORD RedirectionIDPartitionOffset;
    DWORD RedirectionIDPartitionSize;
    DWORD RedirectingIDPartitionOffset;
    DWORD RedirectingIDPartitionSize;
} PartitionInformation;


Struct ExtendedCallInfo
{
    DWORD ExtendedCallReason ;
    DWORD CallerIDURLOffset;// CallPartySipURLInfo
    DWORD CallerIDURISize;
    DWORD CalledIDURLOffset;// CallPartySipURLInfo
    DWORD CalledIDURISize;
    DWORD ConnectedIDURIOffset;// CallPartySipURLInfo
    DWORD ConnectedIDURISize;
    DWORD RedirectionIDURIOffset;// CallPartySipURLInfo
    DWORD RedirectionIDURISize;
    DWORD RedirectingIDURIOffset;// CallPartySipURLInfo
    DWORD RedirectingIDURISize;
}

Struct CallPartySipURLInfo
{
    DWORDdwUserOffset;  //sip user string
    DWORDdwUserSize;
    DWORDdwHostOffset; //host name string
    DWORDdwHostSize;
    DWORDdwPort;// integer port number
    DWORDdwTransportType; // SIP_TRANS_TYPE
    DWORDdwURLType;// SIP_URL_TYPE
}

enum {
        CTI_SIP_TRANSPORT_TCP=1,
        CTI_SIP_TRANSPORT_UDP,
        CTI_SIP_TRANSPORT_TLS
} SIP_TRANS_TYPE;

enum {
    CTI_NO_URL = 0,
    CTI_SIP_URL,
    CTI_TEL_URL
} SIP_URL_TYPE;

typedef struct CallAttributeInfo
{
    DWORD CallAttributeType,
    DWORD PartyDNOffset,
    DWORD PartyDNSize,
    DWORD PartyPartitionOffset,
    DWORD PartyPartitionSize,
    DWORD DeviceNameOffset,
    DWORD DeviceNameSize,
    DWORD OverallCallSecurityStatus
}
typedef struct CCMCallHandleInformation
{
    DWORD CCMCallID;
}

enum
{
```

```
enum
{
 CallAttribute_Regular                    = 0,
 CallAttribute_SilentMonitorCall   .
 CallAttribute_SilentMonitorCall_Target   ,
 CallAttribute_RecordedCall_Automatic     ,
 CallAttribute_RecordedCall_AppControlled
} CallAttributeType
typedef struct CallSecurityStausInfo
{
DWORD CallSecurityStaus
} CallSecurityStausInfo
enum CallSecurityStausValue
{
CallSecurityStatus_Unknown
CallSecurityStatus_NonSecure = 0,
CallSecurityStatus_Secure
}

enum OverallCallSecurityStausValue
{
CallSecurityStatus_Unknown
CallSecurityStatus_NonSecure = 0,
CallSecurityStatus_Secure
}

};
typedef struct CPNInfo
{
    DWORD CallerPartyNumberType;//refer to CiscoNumberType
    DWORD CalledPartyNumberType;
    DWORD ConnectedIdNumberType;
    DWORD RedirectingPartyNumberType;
    DWORD RedirectionPartyNumberType;
            DWORD ModifiedCallingPartySize;
    DWORD ModifiedCallingPartyOffset;
    DWORD ModifiedCalledPartySize;
    DWORD ModifiedCalledPartyOffset;
    DWORD ModifiedConnectedIdSize;
    DWORD ModifiedConnectedIdOffset;
    DWORD ModifiedRedirectingPartySize;
    DWORD ModifiedRedirectingPartyOffset;
    DWORD ModifiedRedirectionPartySize;
    DWORD ModifiedRedirectionPartyOffset;
    DWORD GlobalizedCallingPartySize;
    DWORD GlobalizedCallingPartyOffset;
} CPNInfo;

enum CiscoNumberType {
    NumberType_Unknown  = 0,          // UNKNOWN_NUMBER
    NumberType_International  = 1,    // INTERNATIONAL_NUMBER
    NumberType_National  = 2,         // NATIONAL_NUMBER
    NumberType_NetSpecificNum  = 3,   // NET_SPECIFIC_NUMBER
    NumberType_Subscriber  = 4,       // SUBSCRIBER_NUMBER
    NumberType_Abbreviated  = 6       // ABBREVIATED_NUMBER
};
```

Parameters

Parameter	Value
TapiCallerPartyUnicodeNameOffset TapiCallerPartyUnicodeNameSize	The size, in bytes, of the variably sized field that contains the Unicode Caller party identifier name information, and the offset, in bytes, from the beginning of the LINECALLINFO data structure
TapiCallerPartyLocale	The Unicode Caller party identifier name Locale information
TapiCalledPartyUnicodeNameOffset TapiCalledPartyUnicodeNameSize	The size, in bytes, of the variably sized field that contains the Unicode Called party identifier name information and the offset, in bytes, from the beginning of the LINECALLINFO data structure
TapiCalledPartyLocale	The Unicode Called party identifier name locale information
TapiConnectedPartyUnicodeNameOffset TapiConnectedPartyUnicodeNameSize	The size, in bytes, of the variably sized field that contains the Unicode Connected party identifier name information and the offset, in bytes, from the beginning of the LINECALLINFO data structure
TapiConnectedPartyLocale	The Unicode Connected party identifier name locale information
TapiRedirectionPartyUnicodeNameOffset TapiRedirectionPartyUnicodeNameSize	The size, in bytes, of the variably sized field that contains the Unicode Redirection party identifier name information and the offset, in bytes, from the beginning of the LINECALLINFO data structure
TapiRedirectionPartyLocale	The Unicode Redirection party identifier name locale information
TapiRedirectingPartyUnicodeNameOffset TapiRedirectingPartyUnicodeNameSize	The size, in bytes, of the variably sized field that contains the Unicode Redirecting party identifier name information and the offset, in bytes, from the beginning of the LINECALLINFO data structure
TapiRedirectingPartyLocale	The Unicode Redirecting party identifier name locale information
SRTPKeyInfoStructureOffset	Point to SRTPKeyInfoStructure
SRTPKeyInfoStructureSize	Total size of SRTP info
SRTPKeyInfoStructureElementCount	Number of SRTPKeyInfoStructure element
SRTPKeyInfoStructureElementFixedSize	Fixed size of SRTPKeyInfoStructure
SecureMediaIndicator	Indicates whether media is secure and whether application is authorized for key information
MasterKeyOffset MasterKeySize	The offset and size of SRTP MasterKey information
MasterSaltOffset MasterSaltSize	The offset and size of SRTP MasterSaltKey information
AlgorithmID	Specifies negotiated SRTP algorithm ID
IsMKIPresent	Indicates whether MKI is present

Parameter	Value
KeyDerivationRate	Provides the KeyDerivationRate
DSCPValueForAudioCalls	The DSCP value for Audio Calls
CallerIDPartitionOffset CallerIDPartitionSize	The size, in bytes, of the variably sized field that contains the Caller party identifier partition information and the offset, in bytes, from the beginning of LINECALLINFO data structure
CalledIDPartitionOffset CalledIDPartitionSize	The size, in bytes, of the variably sized field that contains the Called party identifier partition information and the offset, in bytes, from the beginning of LINECALLINFO data structure
ConnectedIDPartitionOffset ConnecetedIDPartitionSize	The size, in bytes, of the variably sized field that contains the Connected party identifier partition information and the offset, in bytes, from the beginning of LINECALLINFO data structure
RedirectionIDPartitionOffset RedirectionIDPartitionSize	The size, in bytes, of the variably sized field that contains the Redirection party identifier partition information, and the offset, in bytes, from the beginning of LINECALLINFO data structure
RedirectingIDPartitionOffset RedirectingIDPartitionSize	The size, in bytes, of the variably sized field that contains the Redirecting party identifier partition information and the offset, in bytes, from the beginning of LINECALLINFO data structure
ExtendedCallReason	Presents all the last feature-related CTI Call reason code to the application as an extension to the standard reason codes that TAPI supports. This provides the feature-specific information per call. As phones that are running SIP are supported through CTI, new features can get introduced for phones that are running on SIP during releases. **Note** Be aware that this field is not backward compatible and can change as changes or additions are made in the SIP phone support for a feature. Applications should implement some default behavior to handle any unknown reason codes that might be provided through this field. For Refer, the reason code specified is CtiCallReason_Refer. For Replaces, the reason code specified is CtiCallReason_Replaces.
CallerIDURLOffset CallerIDURLSize	The size, in bytes, of the variably sized field that contains the Caller party identifier URL information and the offset, in bytes, from the beginning of LINECALLINFO data structure

Parameter	Value
CalledIDURLOffset CalledIDURLSize	The size, in bytes, of the variably sized field that contains the Called party identifier URL information and the offset, in bytes, from the beginning of LINECALLINFO data structure
ConnectedIDURLOffset ConnecetedIDURLSize	The size, in bytes, of the variably sized field that contains the Connected party identifier URL information and the offset, in bytes, from the beginning of LINECALLINFO data structure
RedirectionIDURLOffset RedirectionIDURLSize	The size, in bytes, of the variably sized field that contains the Redirection party identifier URL information and the offset, in bytes, from the beginning of LINECALLINFO data structure
RedirectingIDURLOffset RedirectingIDURLSize	The size, in bytes, of the variably sized field that contains the Redirecting party identifier URL information and the offset, in bytes, from the beginning of LINECALLINFO data structure
CallAttributeType	Identifies whether the following info(DN.Partition.DeviceName) is for a regular call, a monitoring call, a monitored call, or a recording call
PartyDNOffset, PartyDNSize,	The size, in bytes, of the variably sized field that contains the Monitoring/Monitored/Recorder party DN information and the offset, in bytes, from the beginning of the LINECALLINFO data structure
PartyPartitionOffset PartyPartitionSize	The size, in bytes, of the variably sized field that contains the Monitoring/Monitored/Recorder party partition information and the offset, in bytes, from the beginning of the LINECALLINFO data structure
DeviceNameOffset DeviceNameSize	The size, in bytes, of the variably sized field that contains the Monitoring/Monitored/Recorder party device name and the offset, in bytes, from the beginning of the LINECALLINFO data structure
OverallCallSecurityStatus	The security status of the call for two-party calls and conference calls
CCMCallID	The Cisco Unified Communications Manager caller ID for each call leg

To indicate that partition information exists in the LINECALLINFO structure, the system fires a LINECALLINFOSTATE_DEVSPECIFIC event. The bit map indicating the change is defined as the following:

SLDST_NUMBER_TYPE_CHANGED	0x00000080
SLDST_GLOBALIZED_CALLING_PARTY_CHANGED	0x00000100

All available bitmap values of dwParam3 for LINECALLINFOSTATE_DEVSPECIFIC event are:

SLDST_SRTP_INFO	0x00000001
SLDST_QOS_INFO	0x00000002
SLDST_PARTITION_INFO	0x00000004

SLDST_EXTENDED_CALL_INFO	0x00000008
SLDST_CALL_ATTRIBUTE_INFO	0x00000010 //M&R
SLDST_CCM_CALL_ID	0x00000020 //M&R
SLDST_SECURITY_STATUS_INFO	0x00000040 //SecureConf
SLDST_NUMBER_TYPE_CHANGED	0x00000080 //CPN
SLDST_GLOBALIZED_CALLING_PARTY_CHANGED	0x00000100 //CPN
SLDST_FAR_END_IP_ADDRESS_CHANGED	0x00000200//IPv6 new

Also, whenever a change occurs in the partition information, the system fires a LINEDEVSPECIFIC event that indicates which exact field in the devSpecific portion of the LINECALLINFO changed as shown below. This event fires only if the application has negotiated 7.0 extension version or higher.

```
LINEDEVSPECIFIC
{
  hDevice = hcall //call handle for which the info has changed.
  dwParam1 = SLDSMT_LINECALLINFO_DEVSPECIFICDATA //indicates DevSpecific portion's changed
  dwParam2 = SLDST_SRTP_INFO | SLDST_QOS_INFO |SLDST_PARTITION_INFO |
SLDST_EXTENDED_CALL_INFO | SLDST_CALL_ATTRIBUTE_INFO|SLDST_CCM_CALLID|
SLDST_CALL_SECURITY_STATUS
  dwParam3 = …
  dwParam3 will be security indicator if dwParam2 has bit set for SLDST_SRTP_INFO
}
  SLDST_SRTP_INFO = 0x00000001
  SLDST_QOS_INFO = 0x00000002
  SLDST_PARTITION_INFO = 0x00000004
  SLDST_EXTENDED_CALL_INFO= 0x00000008
  SLDST_CALL_ATTRIBUTE_INFO = 0x00000010
  SLDST_CCM_CALLID = 0x00000020
  SLDST_CALL_SECURITY_STATUS=0x00000040
```

LINEDEVSTATUS

Cisco TSP implements several line device-specific extensions and uses the DevSpecific (dwDevSpecificSize and dwDevSpecificOffset) variably sized area of the LINEDEVSTATUS data structure for those extensions. Cisco TSP defines the DevSpecific area layout in the Cisco_LineDevStatus_Ext structure in the CiscoLineDevSpecificMsg.h header file. The extension version in which the data was introduced provides basis for how the data in that structure is organized.

```
//   LINEDEVSTATUS Dev Specific extention   //
typedef struct Cisco_LineDevStatus_Ext
{
    Cisco_LineDevStatus_Ext00060000  ext60;
    Cisco_LineDevStatus_Ext00070000  ext70;
    Cisco_LineDevStatus_Ext00080000  ext80;
} CISCO_LINEDEVSTATUS_EXT;
```

For a specific line device, the extension area will include a portion of this structure, starting from the beginning and up to the extension version that an application negotiated.

Detail

The individual extension version substructure definitions follow:

```
//    LINEDEVSTATUS 00060000  extention    //
typedef struct Cisco_LineDevStatus_Ext00060000
{
```

```
    DWORD dwSupportedEncoding;
} CISCO_LINEDEVSTATUS_EXT00060000;
//    LINEDEVSTATUS 00070000  extention    //
typedef struct Cisco_LineDevStatus_Ext00070000
{
    char lpszAlternateScript[MAX_ALTERNATE_SCRIPT_SIZE];
    // An empty string means there  is no alternate script configured
    // or the phone does not support alternate scripts
} CISCO_LINEDEVSTATUS_EXT00070000;
//    LINEDEVSTATUS 00080000  extention    //
typedef struct CiscoLineDevStatus_DoNotDisturb
{
    DWORD m_LineDevStatus_DoNotDisturbOption;
    DWORD m_LineDevStatus_DoNotDisturbStatus;
} CISCOLINEDEVSTATUS_DONOTDISTURB;
```

You can find additional information on the DevSpecific structure layout and data in the CiscoLineDevSpecificMsg.h header file.

The CiscoLineDevStatus_DoNotDisturb structure belongs to the LINEDEVSTATUS_DEV_SPECIFIC_DATA structure and gets used to reflect the current state of the Do Not Disturb feature.

Parameters

DWORD dwSupportEncoding

This parameter indicates the Support Encoding for the Unicode Party names that are being sent in device-specific extension of the LINECALLINFO structure.

The typical values could be

```
enum {
UnknownEncoding = 0,// Unknown encoding
NotApplicableEncoding = 1,// Encoding not applicable to this device
AsciiEncoding = 2,          // ASCII encoding
Ucs2UnicodeEncoding = 3     // UCS-2 Unicode encoding
}
```

Note Be aware that the dwSupportedEncoding extension is only available if extension version 0x00060000 or higher is negotiated.

LPCSTR lpszAlternateScript

This parameter specifies the alternate script that the device supports. An empty string indicates the device does not support or is not configured with an alternate script.

The only supported script in this release is "Kanji" for the Japanese locale.

m_LineDevStatus_DoNotDisturbOption

This field contains DND option that is configured for the device and can comprise one of the following enum values:

```
enum CiscoDoNotDisturbOption {
    DoNotDisturbOption_NONE      = 0,
    DoNotDisturbOption_RINGEROFF = 1,
    DoNotDisturbOption_REJECT    = 2
};
```

```
m_LineDevStatus_ DoNotDisturbStatus field contains current DND status on the device
and can be one of the following enum values:

enum CiscoDoNotDisturbStatus {
    DoNotDisturbStatus_UNKNOWN  = 0,
    DoNotDisturbStatus_ENABLED  = 1,
    DoNotDisturbStatus_DISABLED = 2
};
```

Note Be aware that this extension is only available if extension version 8.0 (0x00080000) or higher is negotiated.

CCiscoLineDevSpecific

This section provides information on how to perform Cisco Unified TAPI specific functions with the CCiscoLineDevSpecific class, which represents the parent class to all the following classes. It comprises a virtual class and is provided here for informational purposes.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificMsgWaiting
|
+-- CCiscoLineDevSpecificMsgWaitingDirn
|
+-- CCiscoLineDevSpecificUserControlRTPStream
|
+--CciscoLineDevSpecificSetStatusMsgs
|
+--CCiscoLineDevSpecificRedirectResetOrigCalled
|
+--CCiscoLineDevSpecificPortRegistrationPerCall
|
+--CciscoLineDevSpecificSetRTPParamsForCall
|
+--CCiscoLineDevSpecificRedirectSetOrigCalled
|
+--CCiscoLineDevSpecificJoin
|
+--CciscoLineDevSpecificUserSetSRTPAlgorithmID
|
+--CCiscoLineDevSpecificAcquire
|
+--CciscoLineDevSpecificDeacquire
|
+-- CciscoLineDevSpecificSendLineOpen
|
+-- CCiscoLineDevSpecificSetIntercomSpeedDial
|
+-- CCiscoLineDevSpecificTalkBack
|
+-- CciscoLineRedirectWithFeaturePriority
|
+--CCiscoLineDevSpecificStartCallMonitoringReq
|
+--CCiscoLineDevSpecificStartCallRecordingReq
|
+--CCiscoLineDevSpecificStopCallRecordingReq
|
+-- CciscoLineDevSpecificDirectTransfer
|
```

```
+-- CCiscoLineDevSpecificMsgSummary
|
+-- CCiscoLineDevSpecificMsgSummaryDirn
```

Header File

The file CiscoLineDevSpecific.h contains the constant, structure, and class definition for the Cisco line device-specific classes.

Class Detail

```
class CCiscoLineDevSpecific
  {
  public:
    CCicsoLineDevSpecific(DWORD msgType);
    virtual ~CCiscoLineDevSpecific();
    DWORD GetMsgType(void) const {return m_MsgType;}
    void* lpParams() {return &m_MsgType;}
    virtual DWORD dwSize() = 0;
  private:
    DWORD m_MsgType;
  };
```

Functions

lpParms()

You can use function to obtain the pointer to the parameter block.

dwSize()

Function will give the size of the parameter block area.

Parameter

m_MsgType

Specifies the type of message.

Subclasses

Each subclass of CCiscoLineDevSpecific includes a different value that is assigned to the parameter m_MsgType. If you are using C instead of C++, this represents the first parameter in the structure.

Enumeration

The CiscoLineDevSpecificType enumeration provides valid message identifiers.

```
enum CiscoLineDevSpecificType  {
    SLDST_MSG_WAITING = 1,
    SLDST_MSG_WAITING_DIRN,
    SLDST_USER_CRTL_OF_RTP_STREAM,
    SLDST_SET_STATUS_MESSAGES,
    SLDST_NUM_TYPE,
    SLDST_SWAP_HOLD_SETUP_TRANSFER, // Not Supported in Cisco TSP 3.4 and Beyond
    SLDST_REDIRECT_RESET_ORIG_CALLED,
```

```
    SLDST_USER_RECEIVE_RTP_INFO,
    SLDST_USER_SET_RTP_INFO,
    SLDST_JOIN,
    SLDST_USER_SET_SRTP_ALGORITHM_ID,
    SLDST_SEND_LINE_OPEN,
};
```

Message Waiting

The CCiscoLineDevSpecificMsgWaiting class turns the message waiting lamp on or off for the line that the hLine parameter specifies.

Note This extension does not require an extension version to be negotiated.

```
CCiscoLineDevSpecific
  |
  +-- CCiscoLineDevSpecificMsgWaiting
```

Class Detail

```
class CCiscoLineDevSpecificMsgWaiting : public CCiscoLineDevSpecific
{
 public:
  CCiscoLineDevSpecificMsgWaiting() : CCiscoLineDevSpecific(SLDST_MSG_WAITING){}
  virtual ~CCiscoLineDevSpecificMsgWaiting() {}
  virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
  DWORD m_BlinkRate;
};
```

Parameters

DWORD m_MsgType

Equals SLDST_MSG_WAITING.

DWORD m_BlinkRate

Any supported PHONELAMPMODE_ constants that are specified in the phoneSetLamp() function.

Note Cisco Unified IP Phone 7900 Series supports only PHONELAMPMODE_OFF and PHONELAMPMODE_STEADY

Message Waiting Dirn

The CCiscoLineDevSpecificMsgWaitingDirn class turns the message waiting lamp on or off for the line that a parameter specifies and remains independent of the hLine parameter.

Note This extension does not require an extension version to be negotiated.

```
CCiscoLineDevSpecific
  |
  +-- CCiscoLineDevSpecificMsgWaitingDirn
```

Class Detail

```
class CCiscoLineDevSpecificMsgWaitingDirn : public CCiscoLineDevSpecific
{
 public:
  CCiscoLineDevSpecificMsgWaitingDirn() :
    CCiscoLineDevSpecific(SLDST_MSG_WAITING_DIRN) {}
  virtual ~CCiscoLineDevSpecificMsgWaitingDirn() {}
  virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
  DWORD m_BlinkRate;
  char m_Dirn[25];
};
```

Parameters

DWORD m_MsgType

Specifies SLDST_MSG_WAITING_DIRN.

DWORD m_BlinkRate

As in the CCiscoLineDevSpecificMsgWaiting message.

Note Cisco Unified IP Phone 7900 Series supports only PHONELAMPMODE_OFF and PHONELAMPMODE_STEADY

char m_Dirn[25]

The directory number for which the message waiting lamp should be set.

Message Summary

Use the CCiscoLineDevSpecificMsgSummary class to turn the message waiting lamp on or off as well as to provide voice and fax message counts for the line specified by the hLine parameter.

Note Be aware that this extension does not require an extension version to be negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificMsgSummary
```

Class Detail

```
class CCiscoLineDevSpecificMsgSummary : public CCiscoLineDevSpecific
{
public:
  CCiscoLineDevSpecificMsgSummary() : CCiscoLineDevSpecific(SLDST_MSG_SUMMARY){}
  virtual ~CCiscoLineDevSpecificMsgSummary() {}
  virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
  DWORD m_BlinkRate;
  MSG_SUMMARY m_MessageSummary;
};
```

Parameters

DWORD m_MsgType

equals SLDST_MSG_SUMMARY.

DWORD m_BlinkRate

is any supported PHONELAMPMODE_ constants specified in the phoneSetLamp() function.

MSG_SUMMARY m_MessageSummary

A data structure with the following format:

```
typedef struct {
DWORD m_voiceCounts; // indicates if new voice counts are
                      //  provided. True=counts will be displayed
                      //  on supported phones.
DWORD m_totalNewVoiceMsgs; // specifies the total number of new
                      // voice messages. This number includes all
                      // the high and normal priority voice
                      // messages that are new.
DWORD m_totalOldVoiceMsgs; // specifies the total number of old
                      // voice messages. This number includes all
                      // high and normal priority voice messages
                      // that are old.
DWORD m_highPriorityVoiceCounts; // indicates if old voice
                      // counts are provided. True=counts will be
                      // displayed on supported phones.
DWORD m_newHighPriorityVoiceMsgs; //specifies the number of new
                      // high priority voice messages.
DWORD m_oldHighPriorityVoiceMsgs; //specifies the number of old
                      // high priority voice messages.
DWORD m_faxCounts; // indicates if new fax counts are
                      //  provided. True=counts will be displayed
                      //  on supported phones.
DWORD m_totalNewFaxMsgs; // specifies the total number of new
```

```
                      // fax messages. This number includes all
                      // the high and normal priority fax
                      // messages that are new.
DWORD m_totalOldFaxMsgs; // specifies the total number of old
                      // fax messages. This number includes all
                      // high and normal priority fax messages
                      // that are old.
DWORD m_highPriorityFaxCounts; // indicates if old fax counts
                      //  are provided. True=counts will be
                      //  displayed on supported phones.
DWORD m_newHighPriorityFaxMsgs; // specifies the number of new
                      // high priority fax messages.
DWORD m_oldHighPriorityFaxMsgs; // specifies the number of old
                      // high priority fax messages.
       } MSG_SUMMARY;
```

Message Summary Dirn

Use the CCiscoLineDevSpecificMsgSummaryDirn class to turn the message waiting lamp on or off and to provide voice and fax message counts for the line specified by a parameter and is independent of the hLine parameter.

Note Be aware that this extension does not require an extension version to be negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificMsgSummaryDirn
```

Class Detail

```
class CCiscoLineDevSpecificMsgSummaryDirn : public CCiscoLineDevSpecific
{
public:
  CCiscoLineDevSpecificMsgSummaryDirn() : CCiscoLineDevSpecific(SLDST_MSG_SUMMARY_DIRN) {}
  virtual ~CCiscoLineDevSpecificMsgSummaryDirn() {}
  virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
  DWORD m_BlinkRate;
  char m_Dirn[25];
  MSG_SUMMARY m_MessageSummary;
};
```

Parameters

DWORD m_MsgType

equals SLDST_MSG_SUMMARY_DIRN.

DWORD m_BlinkRate

is as in the CCiscoLineDevSpecificMsgSummary message.

char m_Dirn[25]

is the directory number for which the message waiting lamp should be set.

MSG_SUMMARY m_MessageSummary

A data structure with the following format:

```
typedef struct {
DWORD m_voiceCounts; // indicates if new voice counts are
                      //  provided. True=counts will be displayed
                      //  on supported phones.
DWORD m_totalNewVoiceMsgs; // specifies the total number of new
                      // voice messages. This number includes all
                      // the high and normal priority voice
                      // messages that are new.
DWORD m_totalOldVoiceMsgs; // specifies the total number of old
                      // voice messages. This number includes all
                      // high and normal priority voice messages
                      // that are old.
DWORD m_highPriorityVoiceCounts; // indicates if old voice
```

```
                       // counts are provided. True=counts will be
                       // displayed on supported phones.
DWORD m_newHighPriorityVoiceMsgs; //specifies the number of new
                       // high priority voice messages.
DWORD m_oldHighPriorityVoiceMsgs; //specifies the number of old
                       // high priority voice messages.
DWORD m_faxCounts; // indicates if new fax counts are
                       //  provided. True=counts will be displayed
                       //  on supported phones.
DWORD m_totalNewFaxMsgs; // specifies the total number of new
                       // fax messages. This number includes all
                       // the high and normal priority fax
                       // messages that are new.
DWORD m_totalOldFaxMsgs; // specifies the total number of old
                       // fax messages. This number includes all
                       // high and normal priority fax messages
                       // that are old.
DWORD m_highPriorityFaxCounts; // indicates if old fax counts
                       //  are provided. True=counts will be
                       //  displayed on supported phones.
DWORD m_newHighPriorityFaxMsgs; // specifies the number of new
                       // high priority fax messages.
DWORD m_oldHighPriorityFaxMsgs; // specifies the number of old
                       // high priority fax messages.
       } MSG_SUMMARY;
```

Audio Stream Control

The CCiscoLineDevSpecificUserControlRTPStream class controls the audio stream of a line. To use this class you must call the lineNegotiateExtVersion API before opening the line. When lineNegotiateExtVersion is called ensure the highest bit is set on both the dwExtLowVersion and dwExtHighVersion parameters. This causes the call to lineOpen to behave differently. The line does not actually open, but waits for a lineDevSpecific call to complete the open with more information. The CCiscoLineDevSpecificUserControlRTPStream class provides the extra information that is required.

```
CCiscoLineDevSpecific
  |
  +-- CCiscoLineDevSpecificUserControlRTPStream
```

Procedure

Step 1 Call lineNegotiateExtVersion for the deviceID of the line that is to be opened (OR 0x80000000 with the dwExtLowVersion and dwExtHighVersion parameters).

Step 2 Call lineOpen for the deviceID of the line that is to be opened.

Step 3 Call lineDevSpecific with a CCiscoLineDevSpecificUserControlRTPStream message in the lpParams parameter.

Class Detail

```
class CCiscoLineDevSpecificUserControlRTPStream : public CCiscoLineDevSpecific
 {
  public:
  CCiscoLineDevSpecificUserControlRTPStream() :
    CCiscoLineDevSpecific(SLDST_USER_CRTL_OF_RTP_STREAM),
    m_ReceiveIP(-1),
    m_ReceivePort(-1),
    m_NumAffectedDevices(0)
     {
     memset(m_AffectedDeviceID, 0, sizeof(m_AffectedDeviceID));
     }
  virtual ~CCiscoLineDevSpecificUserControlRTPStream() {}
  DWORD m_ReceiveIP;   // UDP audio reception IP
  DWORD m_ReceivePort; // UDP audio reception port
  DWORD m_NumAffectedDevices;
  DWORD m_AffectedDeviceID[10];
  DWORD m_MediaCapCount;
  MEDIA_CAPS m_MediaCaps;
  virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
 };
```

Parameters

DWORD m_MsgType

Equals SLDST_USER_CRTL_OF_RTP_STREAM

DWORD m_ReceiveIP:

The RTP audio reception IP address in network byte order

DWORD m_ReceivePort:

The RTP audio reception port in network byte order

DWORD m_NumAffectedDevices:

The TSP returns this value. It contains the number of deviceIDs in the m_AffectedDeviceID array that are valid. Any device with multiple directory numbers that are assigned to it will have multiple TAPI lines, one per directory number.

DWORD m_AffectedDeviceID[10]:

The TSP returns this value. It contains the list of deviceIDs for any device that is affected by this call. Do not call lineDevSpecific for any other device in this list.

DWORD m_mediaCapCount

The number of codecs that are supported for this line.

MEDIA_CAPS m_MediaCaps -

A data structure with the following format:

typedef struct {

DWORD MediaPayload;

DWORD MaxFramesPerPacket;

DWORD G723BitRate;

} MEDIA_CAPS[MAX_MEDIA_CAPS_PER_DEVICE];

This data structure defines each codec that is supported on a line. The limit specifies 18. The following description shows each member in the MEDIA_CAPS data structure:

MediaPayload specifies an enumerated integer that contains one of the following values:

```
enum
    {
Media_Payload_G711Alaw64k = 2,
Media_Payload_G711Alaw56k = 3, // "restricted"
Media_Payload_G711Ulaw64k = 4,
Media_Payload_G711Ulaw56k = 5, // "restricted"
Media_Payload_G722_64k = 6,
Media_Payload_G722_56k = 7,
Media_Payload_G722_48k = 8,
Media_Payload_G7231 = 9,
Media_Payload_G728 = 10,
Media_Payload_G729 = 11,
Media_Payload_G729AnnexA = 12,
Media_Payload_G729AnnexB = 15,
Media_Payload_G729AnnexAwAnnexB = 16,
Media_Payload_GSM_Full_Rate = 18,
Media_Payload_GSM_Half_Rate = 19,
Media_Payload_GSM_Enhanced_Full_Rate = 20,
Media_Payload_Wide_Band_256k = 25,
Media_Payload_Data64 = 32,
Media_Payload_Data56 = 33,
Media_Payload_GSM = 80,
Media_Payload_G726_32K = 82,
Media_Payload_G726_24K = 83,
Media_Payload_G726_16K = 84,
// Media_Payload_G729_B = 85,
// Media_Payload_G729_B_LOW_COMPLEXITY = 86,
}   Media_PayloadType;
```

Read MaxFramesPerPacket as MaxPacketSize. It specifies a 16-bit integer that indicates the maximum desired RTP packet size in milliseconds. Typically, this value gets set to 20.

G723BitRate specifies a 6-byte field that contains either the G.723.1 information bit rate, or it gets ignored. The following list provides values for the G.723.1 field values:

```
enum
    {
    Media_G723BRate_5_3 = 1, //5.3Kbps
    Media_G723BRate_6_4 = 2  //6.4Kbps
    }   Media_G723BitRate;
```

Set Status Messages

Use the CCiscoLineDevSpecificSetStatusMsgs class to turn on or off the status messages for the line that the hLine parameter specifies. The Cisco Unified TSP supports the following flags:

- DEVSPECIFIC_MEDIA_STREAM—Setting this flag on a line turns on the reporting of media streaming messages for that line. Clearing this flag turns off the reporting of media streaming messages for that line.
- DEVSPECIFIC_CALL_TONE_CHANGED—Setting this flag on a line turns on the reporting of call tone changed events for that line. Clearing this flag turns off the reporting of call tone changed events for that line.

Note This extension only applies if extension version 0x00020001 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificSetStatusMsgs
```

Class Detail

```
class CCiscoLineDevSpecificSetStatusMsgs : public CCiscoLineDevSpecific
{
public:
CCiscoLineDevSpecificSetStatusMsgs() :
CCiscoLineDevSpecific(SLDST_SET_STATUS_MESSAGES) {}
virtual ~CCiscoLineDevSpecificSetStatusMsgs() {}
DWORD m_DevSpecificStatusMsgsFlag;
virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
};
```

Parameters

DWORD m_MsgType

Equals SLDST_SET_STATUS_MESSAGES.

DWORD m_DevSpecificStatusMsgsFlag

Identifies which status changes cause a LINE_DEVSPECIFIC message to be sent to the application.

The supported values follow:

```
#define DEVSPECIFIC_MEDIA_STREAM 0x00000001
#define DEVSPECIFIC_CALL_TONE_CHANGED 0x00000002
#define CALL_DEVSPECIFIC_RTP_EVENTS 0x00000003
#define DEVSPECIFIC_IDLE_TRANSFER_REASON0x00000004
#define DEVSPECIFIC_SPEEDDIAL_CHANGED0x00000008
#define DEVSPECIFIC_PARK_STATUS    0x00000080
```

Swap-Hold/SetupTransfer

Note Cisco Unified TSP 4.0 and later do not support this.

The CCiscoLineDevSpecificSwapHoldSetupTransfer class gets used to perform a SetupTransfer between a call that is in CONNECTED state and a call that is in the ONHOLD state. This function changes the state of the connected call to ONHOLDPENDTRANSFER state and the ONHOLD call to CONNECTED state. This allows a CompleteTransfer to be performed on the two calls. In Cisco Unified TSP 4.0 and later, the TSP allows applications to use lineCompleteTransfer() to transfer the calls without having to use the CCiscoLineDevSpecificSwapHoldSetupTransfer function. Therefore, this function returns LINEERR_OPERATIONUNAVAIL in Cisco Unified TSP 4.0 and beyond.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificSwapHoldSetupTransfer
```

Note This extension only applies if extension version 0x00020002 or higher is negotiated.

Class Details

```
class CCiscoLineDevSpecificSwapHoldSetupTransfer : public CCiscoLineDevSpecific
    {
    public:
      CCiscoLineDevSpecificSwapHoldSetupTransfer() :
CCiscoLineDevSpecific(SLDST_SWAP_HOLD_SETUP_TRANSFER) {}
      virtual ~CCiscoLineDevSpecificSwapHoldSetupTransfer() {}
      DWORD heldCallID;
      virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the
virtual function table pointer
    };
```

Parameters

DWORD m_MsgType

Equals SLDST_SWAP_HOLD_SETUP_TRANSFER.

DWORD heldCallID

Equals the callid of the held call that is returned in dwCallID of LPLINECALLINFO.

HCALL hCall (in lineDevSpecific parameter list)

Equals the handle of the connected call.

Redirect Reset Original Called ID

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificRedirectResetOrigCalled
```

Description

The CCiscoLineDevSpecificRedirectResetOrigCalled class redirects a call to another party while it resets the original called ID of the call to the destination of the redirect.

Note This extension only applies if extension version 0x00020003 or higher is negotiated.

Class Details

```
class CCiscoLineDevSpecificRedirectResetOrigCalled: public CCiscoLineDevSpecific
    {
    public:
      CCiscoLineDevSpecificRedirectResetOrigCalled:
CCiscoLineDevSpecific(SLDST_REDIRECT_RESET_ORIG_CALLED) {}
      virtual ~CCiscoLineDevSpecificRedirectResetOrigCalled{}
      char m_DestDirn[25]; //redirect destination address
      virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the
virtual function table pointer
    };
```

Parameters

DWORD m_MsgType

Equals SLDST_REDIRECT_RESET_ORIG_CALLED.

DWORD m_DestDirn

Equals the destination address where the call needs to be redirected.

HCALL hCall (In lineDevSpecific parameter list)

Equals the handle of the connected call.

Port Registration per Call

The CCiscoLineDevSpecificPortRegistrationPerCall class registers the CTI Port for the RTP parameters on a per-call basis. With this request, the application receives the new lineDevSpecific event that requests that it needs to set the RTP parameters for the call.

To use this class, ensure the lineNegotiateExtVersion API is called before opening the line. When calling lineNegotiateExtVersion, ensure the highest bit is set on both the dwExtLowVersion and dwExtHighVersion parameters.

This causes the call to lineOpen to behave differently. The line does not actually open, but waits for a lineDevSpecific call to complete the open with more information. The extra information required is provided in the CciscoLineDevSpecificPortRegistrationPerCall class.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificPortRegistrationPerCall
```

Procedure

Step 1 Call lineNegotiateExtVersion for the deviceID of the line that is to be opened (or 0x80000000 with the dwExtLowVersion and dwExtHighVersion parameters)

Step 2 Call lineOpen for the deviceID of the line that is to be opened.

Step 3 Call lineDevSpecific with a CciscoLineDevSpecificPortRegistrationPerCall message in the lpParams parameter.

Note This extension is only available if the extension version 0x00040000 or higher gets negotiated.

Class Details

```
class CCiscoLineDevSpecificPortRegistrationPerCall: public CCiscoLineDevSpecific
{
public:
    CCiscoLineDevSpecificPortRegistrationPerCall () :
    CCiscoLineDevSpecific(SLDST_USER_RECEIVE_RTP_INFO),
    m_RecieveIP(-1), m_RecievePort(-1), m_NumAffectedDevices(0)
    {
    memset((char*)m_AffectedDeviceID, 0, sizeof(m_AffectedDeviceID));
    }

    virtual ~ CCiscoLineDevSpecificPortRegistrationPerCall () {}
    DWORD m_NumAffectedDevices;
    DWORD m_AffectedDeviceID[10];
    DWORD m_MediaCapCount;
    MEDIA_CAPSm_MediaCaps;
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
//  subtract out the virtual function table pointer
    };
```

Parameters

DWORD m_MsgType

Equals SLDST_USER_RECEIVE_RTP_INFO

DWORD m_NumAffectedDevices:

TSP returns this value. It contains the number of deviceIDs in the m_AffectedDeviceID array that are valid. Any device with multiple directory numbers that are assigned to it will have multiple TAPI lines, one per directory number.

DWORD m_AffectedDeviceID[10]:

TSP returns this value. It contains the list of deviceIDs for any device that is affected by this call. Do not call lineDevSpecific for any other device in this list.

DWORD m_mediaCapCount

The number of codecs that are supported for this line.

MEDIA_CAPS m_MediaCaps -

A data structure with the following format:

```
typedef struct {
DWORD MediaPayload;
DWORD MaxFramesPerPacket;
DWORD G723BitRate;
} MEDIA_CAPS[MAX_MEDIA_CAPS_PER_DEVICE];
```

This data structure defines each codec that is supported on a line. The limit specifies 18. The following description applies for each member in the MEDIA_CAPS data structure:

MediaPayload is an enumerated integer that contains one of the following values.

```
enum
{
Media_Payload_G711Alaw64k = 2,
Media_Payload_G711Alaw56k = 3, // "restricted"
Media_Payload_G711Ulaw64k = 4,
Media_Payload_G711Ulaw56k = 5, // "restricted"
Media_Payload_G722_64k = 6,
Media_Payload_G722_56k = 7,
Media_Payload_G722_48k = 8,
Media_Payload_G7231 = 9,
Media_Payload_G728 = 10,
Media_Payload_G729 = 11,
Media_Payload_G729AnnexA = 12,
Media_Payload_G729AnnexB = 15,
Media_Payload_G729AnnexAwAnnexB = 16,
Media_Payload_GSM_Full_Rate = 18,
Media_Payload_GSM_Half_Rate = 19,
Media_Payload_GSM_Enhanced_Full_Rate = 20,
Media_Payload_Wide_Band_256k = 25,
Media_Payload_Data64 = 32,
Media_Payload_Data56 = 33,
Media_Payload_GSM = 80,
Media_Payload_G726_32K = 82,
Media_Payload_G726_24K = 83,
Media_Payload_G726_16K = 84,
// Media_Payload_G729_B = 85,
// Media_Payload_G729_B_LOW_COMPLEXITY = 86,
} Media_PayloadType;
```

MaxFramesPerPacket should read as MaxPacketSize and comprises a 16 bit integer that is specified in milliseconds. It indicates the RTP packet size. Typically, this value gets set to 20.

G723BitRate comprises a six byte field that contains either the G.723.1 information bit rate, or gets ignored. The values for the G.723.1 field comprises values that are enumerated as follows.

```
enum
{
Media_G723BRate_5_3 = 1, //5.3Kbps
Media_G723BRate_6_4 = 2 //6.4Kbps
} Media_G723BitRate;
```

Setting RTP Parameters for Call

The CCiscoLineDevSpecificSetRTPParamsForCall class sets the RTP parameters for a specific call.

Note This extension only applies if extension version 0x00040000 or higher gets negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificSetRTPParamsForCall
```

Class Details

```
class CciscoLineDevSpecificSetRTPParamsForCall: public CCiscoLineDevSpecific
{
public:
    CciscoLineDevSpecificSetRTPParamsForCall () :
CCiscoLineDevSpecific(SLDST_USER_SET_RTP_INFO) {}
    virtual ~ CciscoLineDevSpecificSetRTPParamsForCall () {}
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
    // subtract out the virtual function table pointer
    DWORD m_RecieveIP;   // UDP audio reception IP
    DWORD m_RecievePort; // UDP audio reception port
    };
```

Parameters

DWORD m_MsgType

Equals SLDST_USER_SET_RTP_INFO

DWORD m_ReceiveIP

This specifies the RTP audio reception IP address in the network byte order to set for the call.

DWORD m_ReceivePort

This specifies the RTP audio reception port in the network byte order to set for the call.

Redirect Set Original Called ID

The CCiscoLineDevSpecificRedirectSetOrigCalled class redirects a call to another party while it sets the original called ID of the call to any other party.

Note This extension only applies if extension version 0x00040000 or higher gets negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificRedirectSetOrigCalled
```

Class Details

```
class CCiscoLineDevSpecificRedirectSetOrigCalled: public CCiscoLineDevSpecific
{
public:
    CCiscoLineDevSpecificRedirectSetOrigCalled () :
CCiscoLineDevSpecific(SLDST_REDIRECT_SET_ORIG_CALLED) {}
    virtual ~ CCiscoLineDevSpecificRedirectSetOrigCalled () {}
    char m_DestDirn[25];
    char m_SetOriginalCalledTo[25];
    // subtract virtual function table pointer
    virtual DWORD dwSize(void) const {return (sizeof (*this) - 4) ;
}
```

Parameters

DWORD m_MsgType

Equals SLDST_REDIRECT_SET_ORIG_CALLED

char m_DestDirn[25]

Indicates the destination of the redirect. If this request is being used to transfer to voice mail, set this field to the voice mail pilot number of the DN of the line for the voice mail, to which you want to transfer.

char m_SetOriginalCalledTo[25]

Indicates the DN to which the OriginalCalledParty needs to be set. If this request is being used to transfer to voice mail, set this field to the DN of the line for the voice mail, to which you want to transfer.

HCALL hCall (in lineDevSpecific parameter list)

Equals the handle of the connected call.

Join

The CCiscoLineDevSpecificJoin class joins two or more calls into one conference call. Each call that is being joined can be in the ONHOLD or the CONNECTED call state.

The Cisco Unified Communications Manager may succeed in joining some calls that are specified in the Join request, but not all. In this case, the Join request will succeed and the Cisco Unified Communications Manager attempts to join as many calls as possible.

Note This extension only applies if extension version 0x00040000 or higher gets negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificJoin
```

Class Details

```
class CCiscoLineDevSpecificJoin : public CCiscoLineDevSpecific
{
    public:
        CCiscoLineDevSpecificJoin () : CCiscoLineDevSpecific(SLDST_JOIN) {}
        virtual ~ CCiscoLineDevSpecificJoin () {}
        DWORD m_CallIDsToJoinCount;
        CALLIDS_TO_JOIN m_CallIDsToJoin;
        virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
        // subtract out the virtual function table pointer
};
```

Parameters

DWORD m_MsgType

Equals SLDST_JOIN

DWORD m_CallIDsToJoinCount

The number of callIDs that are contained in the m_CallIDsToJoin parameter.

CALLIDS_TO_JOIN m_CallIDsToJoin

A data structure that contains an array of dwCallIDs to join with the following format:

```
typedef struct {
    DWORD   CallID; // dwCallID to Join
} CALLIDS_TO_JOIN[MAX_CALLIDS_TO_JOIN];
```

where MAX_CALLIDS_TO_JOIN is defined as:

```
const DWORD MAX_CALLIDS_TO_JOIN = 14;
```

HCALL hCall (in LineDevSpecific parameter list)

Equals the handle of the call that is being joined with callIDsToJoin to create the conference.

Set User SRTP Algorithm IDs

The CciscoLineDevSpecificUserSetSRTPAlgorithmID class gets used to allow applications to set SRTP algorithm IDs. To use this class, ensure the lineNegotiateExtVersion API is called before opening the line. When calling lineNegotiateExtVersion, ensure the highest bit or second highest bit is set on both the dwExtLowVersion and dwExtHighVersion parameters. This causes the call to lineOpen to behave differently. The line does not actually opens, but waits for a lineDevSpecific call to complete the open with more information. Provide the extra information that is required in the CciscoLineDevSpecificUserSetSRTPAlgorithmID class.

Note This extension is only available if extension version 0x80070000, 0x4007000 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+-- CciscoLineDevSpecificUserSetSRTPAlgorithmID
```

Procedure

Step 1 Call lineNegotiateExtVersion for the deviceID of the line that is to be opened. (0x80070000 or 0x4007000 with the dwExtLowVersion and dwExtHighVersion parameters)

Step 2 Call lineOpen for the deviceID of the line that is to be opened.

Step 3 Call lineDevSpecific with a CciscoLineDevSpecificUserSetSRTPAlgorithmID message in the lpParams parameter to specify SRTP algorithm IDs.

Step 4 Call lineDevSpecific with either CciscoLineDevSpecificPortRegistrationPerCall or CCiscoLineDevSpecificUserControlRTPStream message in the lpParams parameter.

Class Detail

```
class CciscoLineDevSpecificUserSetSRTPAlgorithmID: public CCiscoLineDevSpecific
{
  public:
    CciscoLineDevSpecificUserSetSRTPAlgorithmID () :
    CCiscoLineDevSpecific(SLDST_USER_SET_SRTP_ALGORITHM_ID),
    m_SRTPAlgorithmCount(0),
    m_SRTP_Fixed_Element_Size(4)
    {
    }

    virtual ~ CciscoLineDevSpecificUserSetSRTPAlgorithmID () {}
      DWORD m_SRTPAlgorithmCount;     //Maximum is MAX_CISCO_SRTP_ALGORITHM_IDS
    DWORD m_SRTP_Fixed_Element_Size;//Should be size of DWORD, it should be always 4.
      DWORD m_SRTPAlgorithm_Offset;   //offset from beginning of the message buffer
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the virtual
function table pointer
};
```

Supported Algorithm Constants

```
enum CiscoSRTPAlgorithmIDs
{
    SRTP_NO_ENCRYPTION=0,
```

```
    SRTP_AES_128_COUNTER=1
};
```

Parameters

DWORD m_MsgType

Equals SLDST_USER_SET_SRTP_ALGORITHM_ID

DWORD m_SRTPAlgorithmCount

This numbers of algorithm IDs that are specified in this message.

DWORD m_SRTP_Fixed_Element_Size

Should be size of DWORD, it should be always 4.

DWORD m_SRTPAlgorithm_Offset

Offset from the beginning of the message buffer. This is offset where you start put algorithm ID array.

Note Be aware that the dwSize should be recalculated based on size of the structure, m_SRTPAlgorithmCount and m_SRTP_Fixed_Element_Size.

Explicit Acquire

The CCiscoLineDevSpecificAcquire class gets used to explicitly acquire any CTI controllable device.

If a Superprovider application needs to open any CTI Controllable device on the Cisco Unified Communications Manager system, the application should explicitly acquire that device by using the above interface. After successful response, it can open the device as usual.

Note Be aware that this extension is only available if extension version 0x00070000 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+--CCiscoLineDevSpecificAcquire
```

Class Details

```
class CCiscoLineDevSpecificAcquire : public CCiscoLineDevSpecific
{
    public:
        CCiscoLineDevSpecificAcquire () : CCiscoLineDevSpecific(SLDST_ACQUIRE) {}
        virtual ~ CCiscoLineDevSpecificAcquire () {}
        char m_DeviceName[16];
        virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
        // subtract out the virtual function table pointer
};
```

Parameters

DWORD m_MsgType

Equals SLDST_ACQUIRE

m_DeviceName[16]

The DeviceName that needs to be explicitly acquired.

Explicit De-Acquire

The CCiscoLineDevSpecificDeacquire class is used to explicitly de-acquire the explicitly acquired device.

If a Superprovider application has explicitly acquired any CTI Controllable device on the Cisco Unified Communications Manager system, then the application should explicitly De-acquire that device by using the above interface.

Note Be aware that this extension is only available if extension version 0x00070000 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+--CCiscoLineDevSpecificDeacquire
```

Class Details

```
class CCiscoLineDevSpecificDeacquire : public CCiscoLineDevSpecific
{
    public:
CCiscoLineDevSpecificDeacquire () : CCiscoLineDevSpecific(SLDST_ACQUIRE) {}
        virtual ~ CCiscoLineDevSpecificDeacquire () {}
        char m_DeviceName[16];
        virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
        // subtract out the virtual function table pointer
};
```

Parameters

DWORD m_MsgType

Equals SLDST_DEACQUIRE

char m_DeviceName[16]

The DeviceName that needs to be explicitly de-acquired.

Redirect FAC CMC

The CCiscoLineDevSpecificRedirectFACCMC class is used to redirect a call to another party that requires a FAC, CMC, or both.

Note Be aware that this extension is only available if extension version 0x00050000 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+--CCiscoLineDevSpecificRedirectFACCMC
```

If the FAC is invalid, the TSP will return a new device-specific error code LINEERR_INVALIDFAC. If the CMC is invalid, the TSP will return a new device-specific error code LINEERR_INVALIDCMC.

Class Detail

```
class CCiscoLineDevSpecificRedirectFACCMC: public CCiscoLineDevSpecific
{
public:
    CCiscoLineDevSpecificRedirectFACCMC () : CCiscoLineDevSpecific(SLDST_REDIRECT_FAC_CMC)
{}
    virtual ~ CCiscoLineDevSpecificRedirectFACCMC () {}
    char m_DestDirn[49];
    char m_FAC[17];
    char m_CMC[17];
    // subtract virtual function table pointer
    virtual DWORD dwSize(void) const {return (sizeof (*this) - 4) ;
}
```

Parameters

DWORD m_MsgType

Equals SLDST_REDIRECT_FAC_CMC

char m_DestDirn[49]

Indicates the destination of the redirect.

char m_FAC[17]

Indicates the FAC digits. If the application does not want to pass any FAC digits, it must set this parameter to a NULL string.

char m_CMC[17]

Indicates the CMC digits. If the application does not want to pass any CMC digits, it must set this parameter to a NULL string.

HCALL hCall (in lineDevSpecific parameter list)

Equals the handle of the call to be redirected.

Blind Transfer FAC CMC

The CCiscoLineDevSpecificBlindTransferFACCMC class is used to blind transfer a call to another party that requires a FAC, CMC, or both. If the FAC is invalid, the TSP will return a new device specific error code LINEERR_INVALIDFAC. If the CMC is invalid, the TSP will return a new device specific error code LINEERR_INVALIDCMC.

Note Be aware that this extension is only available if extension version 0x00050000 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+--CCiscoLineDevSpecificBlindTransferFACCMC
```

Class Detail

```
class CCiscoLineDevSpecificBlindTransferFACCMC: public CCiscoLineDevSpecific
{
public:
    CCiscoLineDevSpecificBlindTransferFACCMC () :
CCiscoLineDevSpecific(SLDST_BLIND_TRANSFER_FAC_CMC) {}
    virtual ~ CCiscoLineDevSpecificBlindTransferFACCMC () {}
    char m_DestDirn[49];
    char m_FAC[17];
    char m_CMC[17];
    // subtract virtual function table pointer
    virtual DWORD dwSize(void) const {return (sizeof (*this) - 4) ;
}
```

Parameters

DWORD m_MsgType

Equals SLDST_BLIND_TRANSFER_FAC_CMC

char m_DestDirn[49]

Indicates the destination of the blind transfer.

char m_FAC[17]

Indicates the FAC digits. If the application does not want to pass any FAC digits, it must set this parameter to a NULL string.

char m_CMC[17]

Indicates the CMC digits. If the application does not want to pass any CMC digits, it must set this parameter to a NULL string.

HCALL hCall (in lineDevSpecific parameter list)

Equals the handle of the call that is to be blind transferred.

CTI Port Third Party Monitor

The CCiscoLineDevSpecificCTIPortThirdPartyMonitor class is used for opening CTI ports in third-party mode.

To use this class, ensure the lineNegotiateExtVersion API is called before opening the line. When calling lineNegotiateExtVersion, ensure the highest bit is set on both the dwExtLowVersion and dwExtHighVersion parameters. This causes the call to lineOpen to behave differently. The line does not actually open, but waits for a lineDevSpecific call to complete the open with more information. Provide the extra information that is required in the CCiscoLineDevSpecificCTIPortThirdPartyMonitor class.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificCTIPortThirdPartyMonitor
```

Procedure

Step 1 Call lineNegotiateExtVersion for the deviceID of the line that is to be opened. (OR 0x80000000 with the dwExtLowVersion and dwExtHighVersion parameters)

Step 2 Call lineOpen for the deviceID of the line that is to be opened.

Step 3 Call lineDevSpecific with a CCiscoLineDevSpecificCTIPortThirdPartyMonitor message in the lpParams parameter.

Note Be aware that this extension is only available if extension version 0x00050000 or higher is negotiated.

Class Detail

```
class CCiscoLineDevSpecificCTIPortThirdPartyMonitor: public CCiscoLineDevSpecific
{
public:
    CCiscoLineDevSpecificCTIPortThirdPartyMonitor () :
    CCiscoLineDevSpecific(SLDST_CTI_PORT_THIRD_PARTY_MONITOR) {}
    virtual ~ CCiscoLineDevSpecificCTIPortThirdPartyMonitor () {}
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;} //
    subtract out the virtual function table pointer
};
```

Parameters

DWORD m_MsgType

equals SLDST_CTI_PORT_THIRD_PARTY_MONITOR

Send Line Open

The CciscoLineDevSpecificSendLineOpen class is used for general delayed open purpose. To use this class, ensure the lineNegotiateExtVersion API is called before opening the line. When calling lineNegotiateExtVersion, ensure the second highest bit is set on both the dwExtLowVersion and dwExtHighVersion parameters. This causes the call to lineOpen to behave differently. The line does not actually open, but waits for a lineDevSpecific call to complete the open with more information. The extra information required is provided in the CciscoLineDevSpecificUserSetSRTPAlgorithmID class.

```
CCiscoLineDevSpecific
|
+-- CciscoLineDevSpecificSendLineOpen
```

Procedure

Step 1 Call lineNegotiateExtVersion for the deviceID of the line that is to be opened. (0x40070000 with the dwExtLowVersion and dwExtHighVersion parameters).

Step 2 Call lineOpen for the deviceID of the line that is to be opened.

Step 3 Call other lineDevSpecific, like CciscoLineDevSpecificUserSetSRTPAlgorithmID message in the lpParams parameter to specify SRTP algorithm IDs.

Step 4 Call lineDevSpecific with either CciscoLineDevSpecificSendLineOpen to trigger the lineopen from TSP side.

Note Be aware that this extension is only available if extension version 0x40070000 or higher is negotiated.

Class Detail

```
class CciscoLineDevSpecificSendLineOpen: public CCiscoLineDevSpecific
  {
  public:
    CciscoLineDevSpecificSendLineOpen () :
    CCiscoLineDevSpecific(SLDST_SEND_LINE_OPEN) {}

    virtual ~ CciscoLineDevSpecificSendLineOpen () {}
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the virtual
function table pointer
};
```

Set Intercom SpeedDial

Use the CciscoLineSetIntercomSpeeddial class to allow application to set or reset SpeedDial/Label on an intercom line.

Note Be aware that this extension is only available if extension version 0x00080000 or higher is negotiated

```
CCiscoLineDevSpecific
|
+-- CciscoLineSetIntercomSpeeddial
```

Procedure

Step 1 Call lineNegotiateExtVersion for the deviceID of the line that is to be opened (0x00080000 or higher).

Step 2 Call lineOpen for the deviceID of the line that is to be opened.

Step 3 Wait for line in service.

Step 4 Call CciscoLineSetIntercomSpeeddial to set or reset speed dial setting on the intercom line.

Class Detail

```
class CciscoLineSetIntercomSpeeddial: public CCiscoLineDevSpecific
  {
  public:
    CciscoLineSetIntercomSpeeddial () :
    CCiscoLineDevSpecific(SLDST_LINE_SET_INTERCOM_SPEEDDIAL) {}

    virtual ~ CciscoLineSetIntercomSpeeddial () {}
    DWORD SetOption;        //0=clear app value, 1= set App Value
    char Intercom_DN[MAX_DIRN];
    char Intercom_Ascii_Label[MAX_DIRN];
    wchar_t Intercom_Unicode_Label[MAX_DIRN];
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the virtual
function table pointer
};
```

Parameters

DWORD m_MsgType

Equals SLDST_USER_SET_INTERCOM_SPEEDDIAL

DWORD SetOption

Use this parameter to indicate whether the application wants to set a new intercom speed dial value or clear the previous value. 0 = clear, 1 = set.

Char Intercom_DN [MAX_DIRN]

A DN array that indicates the intercom target

Char Intercom_Ascii_Label[MAX_DIRN]

Indicates the ASCII value of the intercom line label

Wchar_tIntercom_Unicode_Label[MAX_DIRN]

Indicates the Unicode value of the intercom line label

MAX_DIRN is defined as 25.

Intercom Talk Back

Use the CCiscoLineDevSpecificTalkBack class to allow application to initiate talk back on an incoming intercom call on an intercom line.

Note Be aware that this extension is only available if extension version 0x00080000 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificTalkBack
```

Class Detail

```
class CCiscoLineDevSpecificTalkBack: public CCiscoLineDevSpecific
{
  public:
    CCiscoLineDevSpecificTalkBack () :
    CCiscoLineDevSpecific(SLDST_INTERCOM_TALKBACK) {}

    virtual ~ CCiscoLineDevSpecificTalkBack () {}
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the virtual
function table pointer
};
```

Redirect with Feature Priority

CciscoLineRedirectWithFeaturePriority enables an application to redirect calls with specified feature priorities. The following is the structure of CciscoLineDevSpecific:

```
CCiscoLineDevSpecific
|
+-- CciscoLineRedirectWithFeaturePriority
```

Note Be aware that this extension is only available if the extension version 0x00080001 or higher is negotiated.

Detail

```
class CciscoLineRedirectWithFeaturePriority: public CCiscoLineDevSpecific
  {
  public:
    CciscoLineRedirectWithFeaturePriority() :
    CCiscoLineDevSpecific(SLDST_REDIRECT_WITH_FEATURE_PRIORITY) {}

    virtual ~ CciscoLineRedirectWithFeaturePriority () {}
    CiscoDoNotDisturbFeaturePriority FeaturePriority;
    char m_DestDirn[25];
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the virtual
function table pointer
};
```

Parameters

DWORD m_MsgType

Equals SLDST_REDIRECT_WITH_FEATURE_PRIORITY

enum CiscoDoNotDisturbFeaturePriority {CallPriority_NORMAL = 1, CallPriority_URGENT = 2, CallPriority_EMERGENCY = 3};

This identifies the priorities.

char m_DestDirn[25];

This is redirect destination.

Start Call Monitoring

Use CCiscoLineDevSpecificStartCallMonitoring to allow application to send a start monitoring request for the active call on a line.

Note Be aware that this extension is only available if extension version 0x00080000 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificStartCallMonitoring
```

Class Detail

```
class CCiscoLineDevSpecificStartCallMonitoring: public CCiscoLineDevSpecific
{
public:
    CCiscoLineDevSpecificStartCallMonitoring () :
CCiscoLineDevSpecific(SLDST_START_CALL_MONITORING) {}
    virtual ~    CCiscoLineDevSpecificStartCallMonitoring () {}
    DWORD m_PermanentLineID ;
  DWORD m_MonitorMode;
```

```
   DWORD m_ToneDirection;
   // subtract out the virtual function table pointer
   virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
} ;
```

Parameters

DWORD m_MsgType

Equals SLDST_START_MONITORING

DWORD m_ PermanentLineID

The permanent lineID of the line whose active call has to be monitored.

DWORD MonitorMode

This can have the following enum value:

```
enum
         {
    MonitorMode_None  = 0,
    MonitorMode_Silent  = 1,
     MonitorMode_Whisper = 2,    //  Not used
         MonitorMode_Active  = 3    //  Not used
} MonitorMode;
```

Note Silent Monitoring mode represents the only mode that is supported in which the supervisor cannot talk to the agent.

DWORD PlayToneDirection

This parameter specifies whether a tone should play at the agent or customer phone when monitoring starts. It can have following enum values:

```
enum
    {
     PlayToneDirection_LocalOnly = 0,
     PlayToneDirection_RemoteOnly,
     PlayToneDirection_BothLocalAndRemote,
     PlayToneDirection_NoLocalOrRemote
    } PlayToneDirection
```

Return Values

```
-   LINERR_OPERATIONFAILED
-   LINEERR_OPERATIONUNAVAIL
-   LINEERR_RESOURCEUNAVAIL
-   LINEERR_BIB_RESOURCE_UNAVAIL
-   LINERR_PENDING_REQUEST
-   LINEERR_OPERATION_ALREADY_INPROGRESS
-   LINEERR_ALREADY_IN_REQUESTED_STATE
-   LINEERR_PRIMARY_CALL_INVALID
-   LINEERR_PRIMARY_CALL_STATE_INVALID
```

Start Call Recording

Use CCiscoLineDevSpecificStartCallRecording to allow applications to send a recording request for the active call on that line.

Note Be aware that this extension is only available if extension version 0x00080000 or higher is negotiated

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificStartCallRecording
```

Class Detail

```
class CCiscoLineDevSpecificStartCallRecording: public CCiscoLineDevSpecific
{
public:
CCiscoLineDevSpecificStartCallRecording () :
CCiscoLineDevSpecific(SLDST_START_CALL_RECORDING) {}
    virtual ~    CCiscoLineDevSpecificStartCallRecording () {}

    DWORD m_ToneDirection;
    // subtract out the virtual function table pointer
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
} ;
```

Parameters

DWORD m_MsgType

Equals SLDST_START_RECORDING

DWORD PlayToneDirection

This parameter specifies whether a tone should play at the agent or customer phone when recording starts. It can have following enum values:

```
enum
    {
     PlayToneDirection_NoLocalOrRemote = 0,
     PlayToneDirection_LocalOnly,
     PlayToneDirection_RemoteOnly,
     PlayToneDirection_BothLocalAndRemote
    } PlayToneDirection
```

Return Values

```
-   LINERR_OPERATIONFAILED
-   LINEERR_OPERATIONUNAVAIL
-   LINEERR_INVALCALLHANDLE
-   LINEERR_BIB_RESOURCE_UNAVAIL
-   LINERR_PENDING_REQUEST
-   LINERR_OPERATION_ALREADY_INPROGRESS
```

StopCall Recording

Use CCiscoLineDevSpecificStopCallRecording to allow application to stop recording a call on that line.

Note Be aware that this extension is only available if extension version 0x00080000 or higher is negotiated.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificStopCallRecording
```

Class Detail

```
class CCiscoLineDevSpecificStopCallRecording: public CCiscoLineDevSpecific
{
public:
CCiscoLineDevSpecificStopCallRecording () :
CCiscoLineDevSpecific(SLDST_STOP_CALL_RECORDING) {}
    virtual ~    CCiscoLineDevSpecificStopCallRecording () {}

    // subtract out the virtual function table pointer
    virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
} ;
```

Parameters

DWORD m_MsgType

Equals SLDST_STOP_RECORDING

Return Values

```
-   LINERR_OPERATIONFAILED
-   LINEERR_OPERATIONUNAVAIL
-   LINEERR_INVALCALLHANDLE
-   LINERR_PENDING_REQUEST
```

Set IP Address Mode

Use CCiscoLineDevSpecificSetIPAddressMode to enable the application to set the address mode during registration. To use this class, ensure that lineNegotiateExtVersion API is called before opening the line. When calling lineNegotiateExtVersion, ensure the highest bit or second highest is set on both the dwExtLowVersion and dwExtHighVersion parameters. This causes the call to lineOpen to behave differently. The line is not actually opened, but waits for a lineDevSpecific call to complete the open with more information. Provide the extra information required in the CCiscoLineDevSpecificSetIPAddressMode class.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificSetIPAddressMode
```

Note Be aware that this extension is available only if extension version 0x80090000, 0x40090000 or higher is negotiated.

Procedure

Step 1 Call lineNegotiateExtVersion for the deviceID of the line that is to be opened (0x80090000 or 0x40090000 with the dwExtLowVersion and dwExtHighVersion parameters).

Step 2 Call lineOpen for the deviceID of the line that is to be opened.

Step 3 Call lineDevSpecific with a CCiscoLineDevSpecificSetIPAddressMode message in the lpParams parameter to specify IP Addressing mode.

Supported Address Modes

enum CiscoIPAddressMode

```
{
    IP_ADDRESS_V4=1,
    IP_ADDRESS_V6=2,
    IP_ADDRESS_V4_V6=3
};
```

Class Detail

```
class CCiscoLineDevSpecificSetIPAddressMode: public CCiscoLineDevSpecific
{
  public:
    CCiscoLineDevSpecificSetIPAddressMode () :
    CCiscoLineDevSpecific(SLDST_USER_SET_IP_ADDRESS_MODE),
    m_IPAddressMode(0)
    {
    }

    virtual ~ CCiscoLineDevSpecificSetIPAddressMode () {}
      int m_ IPAddressMode;     //Addressing Mode to be specified

    virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the virtual
function table pointer
};
```

Parameters

DWORD m_MsgType

Equals SLDST_USER_SET_IP_ADDRESS_MODE

int m_ IPAddressMode

This specifies the Addressing mode with which user wants the CTI Port/RP registered.

Set IPv6 Address

Use CCiscoLineDevSpecificSetIPv6Address class to allow the application to set IPv6 address during static registration. To use this class, ensure the lineNegotiateExtVersion API must be called before opening the line. When calling lineNegotiateExtVersion, ensure the highest bit or second highest must be set on both the dwExtLowVersion and dwExtHighVersion parameters. This causes the call to lineOpen to behave differently. The line does not actually open, but waits for a lineDevSpecific call to complete the open with more information. The extra information required is provided in the CCiscoLineDevSpecificSetIPv6Address class.

```
CCiscoLineDevSpecific
|
+-- CCiscoLineDevSpecificSetIPv6Address
```

Note Be aware that this extension is available only if extension version 0x80090000, 0x40090000 or higher is negotiated.

Procedure

Step 1 Open Call lineNegotiateExtVersion for the deviceID of the line (0x90070000 or 0x40090000 with the dwExtLowVersion and dwExtHighVersion parameters)

Step 2 Open Call lineOpen for the deviceID of the line.

Step 3 Call lineDevSpecific with a CCiscoLineDevSpecificSetIPAddressMode message in the lpParams parameter to specify IP Addressing mode as IPv6.

Step 4 Call lineDevSpecific with a CCiscoLineDevSpecificSetIPv6Address message in the lpParams parameter to specify IPv6 address for registration.

Class Detail

```
class CCiscoLineDevSpecificSetIPv6Address: public CCiscoLineDevSpecific
{
  public:
    CCiscoLineDevSpecificSetIPv6Address () :
    CCiscoLineDevSpecific(SLDST_USER_SET_IPv6_ADDRESS),
    m_ReceiveIPv6Address(-1), m_ReceivePort(-1)
    {
    }

    virtual ~ CCiscoLineDevSpecificSetIPv6Address () {}
      char m_ReceiveIPv6Address[16];     //Ipv6 address that user wants to specify
    DWORD m_ReceivePort;

    virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the virtual
function table pointer
};
```

Parameters

DWORD m_MsgType

Equals SLDST_USER_SET_IPv6_ADDRESS

char m_ReceiveIPv6Address[16]

User has to specify the IPv6 address to register the CTI Port with

DWORD m_ReceivePort

This specifies the port number for the user to register the CTI Port.

Set RTP Parameters for IPv6 Calls

Use CciscoLineDevSpecificSetRTPParamsForCallIPv6 class to set the RTP parameters for calls for which you must specify IPv6 address.

Note Be aware that this extension is available only if extension version 0x00090000 or higher is negotiated.

Class Detail

```
class CciscoLineDevSpecificSetRTPParamsForCallIPv6: public CCiscoLineDevSpecific
{
public:
CciscoLineDevSpecificSetRTPParamsForCallIPv6 () :
CCiscoLineDevSpecific(SLDST_USER_SET_RTP_INFO_IPv6) {}
virtual ~ CciscoLineDevSpecificSetRTPParamsForCallIPv6 () {}
virtual DWORD dwSize(void) const {return sizeof(*this)-4;} // subtract out the virtual
function table pointer
char m_RecieveIPv6[16];   // UDP audio reception IPv6
DWORD m_RecievePort // UDP audio reception port
  };
```

Parameters

DWORD m_MsgType

Equals SLDST_USER_SET_RTP_INFO_IPv6

DWORD m_ReceiveIPv6

This is the RTP audio reception IPv6 address to set for the call

DWORD m_RecievePort

This is the RTP audio reception port to set for the call.

Direct Transfer

Use the CciscoLineDevSpecificDirectTransfer to transfer calls across lines or on the same line.

```
CCiscoLineDevSpecific
|
+-- CciscoLineDevSpecificDirectTransfer
```

Note Be aware that this extension is available only if extension version 0x00090001 or higher is negotiated.

Class Detail

```
class CciscoLineDevSpecificDirectTransfer: public CCiscoLineDevSpecific
{
    public:
        CciscoLineDevSpecificDirectTransfer () :
CCiscoLineDevSpecific(SLDST_DIRECT_TRANSFER) {}
        virtual ~ CciscoLineDevSpecificDirectTransfer () {}
        DWORD m_CallIDsToTransfer;
        virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
```

```
        // subtract out the virtual function table pointer
};
```

Parameters

DWORD m_MsgType

equals SLDST_ DIRECT_TRANSFER

DWORD m_CallIDsToTransfer

Consult dwCallID to be transferred

HCALL hCall (in LineDevSpecific parameter list)

Equals the handle of the call that is being transferred.

Cisco Line Device Feature Extensions

CCiscoLineDevSpecificFeature represents the parent class. Currently, it consist of only one subclass: CCiscoLineDevSpecificFeature_DoNotDisturb, which allows applications to enable and disable the Do-Not-Disturb feature on a device.

This section describes line device feature-specific extensions to the TAPI structures that Cisco TSP supports, and it contains the following structure:

- LINEDEVCAPS, page 6-49
- LINEDEVSTATUS, page 6-50
- CCiscoLineDevSpecificFeature, page 6-50
- Do-Not-Disturb, page 6-51
- Do-Not-Disturb Change Notification Event, page 6-52

LINEDEVCAPS

The CiscoLineDevCaps_DevSpecificFlags structure contains line device capability extension flags that describe the Cisco line device specific extensions for device capabilities. The m_LineDevCaps_DevSpecificFeatureFlags field in that structure reflects extended device feature capabilities. Currently, Cisco TSP uses only the LINEDEVCAPS_DEVSPECIFICFEATURE_DONOTDISTURB (0x00000001) bit in that field.

```
//   Line device capability extention flags
typedef struct CiscoLineDevCaps_DevSpecificFlags
{
    DWORD m_LineDevCaps_DevSpecificFlags;           // LINEFEATURE_DEVSPECIFIC
    DWORD m_LineDevCaps_DevSpecificFeatureFlags;    // LINEFEATURE_DEVSPECIFICFEAT
} CISCOLINEDEVCAPS_DEVSPECIFICFLAGS;

// Bit assignments
#define LINEDEVCAPS_DEVSPECIFICFEATURE_DONOTDISTURB   0x00000001  // Ext 00080000
```

LINEDEVSTATUS

The LINEDEVSTATUS_DEV_SPECIFIC_DATA structure contains data for all device-specific extensions that Cisco TSP added to the TAPI LINEDEVSTATUS structure. The CiscoLineDevStatus_DoNotDisturb structure belongs to the LINEDEVSTATUS_DEV_SPECIFIC_DATA structure and reflects the current state of the Do-Not-Disturb feature.

Note Be aware that this extension is only available if extension version 8.0 (0x00080000) or higher is negotiated.

```
//    LINEDEVSTATUS 00080000  extention    //
//    --------------------------------
typedef struct CiscoLineDevStatus_DoNotDisturb
{
    DWORD m_LineDevStatus_DoNotDisturbOption;
    DWORD m_LineDevStatus_DoNotDisturbStatus;
} CISCOLINEDEVSTATUS_DONOTDISTURB;
```

The m_LineDevStatus_DoNotDisturbOption field contains DND option that is configured for the device and can comprise one of the following enum values:

```
enum CiscoDoNotDisturbOption {
    DoNotDisturbOption_NONE      = 0,
    DoNotDisturbOption_RINGEROFF = 1,
    DoNotDisturbOption_REJECT    = 2
};
```

The m_LineDevStatus_ DoNotDisturbStatus field contains current DND status on the device and can comprise one of the following enum values:

```
enum CiscoDoNotDisturbStatus {
    DoNotDisturbStatus_UNKNOWN  = 0,
    DoNotDisturbStatus_ENABLED  = 1,
    DoNotDisturbStatus_DISABLED = 2
};
```

CCiscoLineDevSpecificFeature

This section provides information on how to invoke Cisco-specific TAPI extensions with the CCiscoLineDevSpecificFeature class, which represents the parent class to all the following classes.

Note Be aware that this virtual class is provided for informational purposes only.

```
CCiscoLineDevSpecificFeature
```

Header File

The file CiscoLineDevSpecific.h contains the corresponding constant, structure, and class definitions for the Cisco lineDevSpecificFeature extension classes.

Class Detail

```
class CCiscoLineDevSpecificFeature
{
public:
  CCicsoLineDevSpecificFeature(const DWORD msgType): m_MsgType(msgType) {;}
  virtual ~ CCicsoLineDevSpecificFeature() {;}
  DWORD GetMsgType(void) const {return m_MsgType;}
  void* lpParams(void) const {return (void*)&m_MsgType;}
  virtual DWORD dwSize(void) const = 0;
private:
  DWORD m_MsgType;
};
```

Functions

lpParms()

Function that can be used to obtain a pointer to the parameter block

dwSize()

Function that returns size of the parameter block area

Parameter

m_MsgType

Specifies the type of message. The parameter value uniquely identifies the feature to invoke on the device. The PHONEBUTTONFUNCTION_ TAPI_Constants definition lists the valid feature identifiers. Currently, the only recognized value specifies PHONEBUTTONFUNCTION_DONOTDISTURB (0x0000001A).

Each subclass of CCiscoLineDevSpecificFeature includes a unique value that is assigned to the m_MsgType parameter.

Subclasses

Each subclass of CCiscoLineDevSpecificFeature carries a unique value that is assigned to the m_MsgType parameter. If you are using C instead of C++, this represents the first parameter in the structure.

Do-Not-Disturb

Use the CCiscoLineDevSpecificFeature_DoNotDisturb class in conjunction with the request to enable or disable the DND feature on a device.

The Do-Not-Disturb feature gives phone users the ability to go into a Do Not Disturb (DND) state on the phone when they are away from their phones or simply do not want to answer the incoming calls. A phone softkey, DND, allows users to enable or disable this feature.

```
CCiscoLineDevSpecificFeature
|
+-- CCiscoLineDevSpecificFeature_DoNotDisturb
```

Class Detail

```
class CCiscoLineDevSpecificFeature_DoNotDisturb : public CCiscoLineDevSpecificFeature
{
public:
  CCiscoLineDevSpecificFeature_DoNotDisturb()
: CCiscoLineDevSpecificFeature(PHONEBUTTONFUNCTION_DONOTDISTURB),
     m_Operation((CiscoDoNotDisturbOperation)0) {}
virtual ~CCiscoLineDevSpecificFeature_DoNotDisturb() {}
virtual DWORD dwSize(void) const {return sizeof(*this)-4;}

CiscoDoNotDisturbOperation  m_Operation;
};
```

Parameters

DWORD m_MsgType

Equals PHONEBUTTONFUNCTION_DONOTDISTURB.

CiscoDoNotDisturbOperation m_Operation

Specifies a requested operation and can comprise one of the following enum values:

```
enum CiscoDoNotDisturbOperation {
    DoNotDisturbOperation_ENABLE    = 1,
    DoNotDisturbOperation_DISABLE   = 2
};
```

Do-Not-Disturb Change Notification Event

Cisco TSP notifies applications via the LINE_DEVSPECIFICFEATURE message about changes in the DND configuration or status. To receive change notifications, an application needs to enable the DEVSPECIFIC_DONOTDISTURB_CHANGED message flag with a lineDevSpecific SLDST_SET_STATUS_MESSAGES request.

The LINE_DEVSPECIFICFEATURE message notifies the application about device-specific events that occur on a line device. In the case of a Do-Not-Disturb Change Notification, the message includes information about the type of change that occurred on a device and the resulting feature status or configured option.

Message Details

```
LINE_DEVSPECIFICFEATURE
dwDevice = (DWORD) hLine;
dwCallbackInstance = (DWORD) hCallback;
dwParam1 = (DWORD) PHONEBUTTONFUNCTION_DONOTDISTURB;
dwParam2 = (DWORD) typeOfChange;
dwParam3 = (DWORD) currentValue;

enum CiscoDoNotDisturbOption {
    DoNotDisturbOption_NONE      = 0,
    DoNotDisturbOption_RINGEROFF = 1,
    DoNotDisturbOption_REJECT    = 2
};

enum CiscoDoNotDisturbStatus {
    DoNotDisturbStatus_UNKNOWN  = 0,
    DoNotDisturbStatus_ENABLED  = 1,
```

```
    DoNotDisturbStatus_DISABLED = 2
};

enum CiscoDoNotDisturbNotification {
    DoNotDisturb_STATUS_CHANGED  = 1,
    DoNotDisturb_OPTION_CHANGED  = 2
};
```

Parameters

dwDevice

A handle to a line device

dwCallbackInstance

The callback instance that is supplied when the line is opened

dwParam1

Always equal to PHONEBUTTONFUNCTION_DONOTDISTURB for the Do-Not-Disturb change notification

dwParam2

Indicates type of change and can comprise one of the following enum values:

```
enum CiscoDoNotDisturbNotification {
    DoNotDisturb_STATUS_CHANGED  = 1,
    DoNotDisturb_OPTION_CHANGED  = 2
};
```

dwParam3

If the dwParm2 indicates status change with the value DoNotDisturb_STATUS_CHANGED, this parameter can comprise one of the following enum values:

```
enum CiscoDoNotDisturbStatus {
    DoNotDisturbStatus_UNKNOWN  = 0,
    DoNotDisturbStatus_ENABLED  = 1,
    DoNotDisturbStatus_DISABLED = 2
};
```

If the dwParm2 indicates option change with the value DoNotDisturb_OPTION_CHANGED, this parameter can comprise one of the following enum values:

```
enum CiscoDoNotDisturbOption {
    DoNotDisturbOption_NONE      = 0,
    DoNotDisturbOption_RINGEROFF = 1,
    DoNotDisturbOption_REJECT    = 2
};
```

Cisco Phone Device-Specific Extensions

Table 6-2 lists the subclasses of CiscoPhoneDevSpecific.

Table 6-2 Cisco Phone Device-Specific TAPI functions

Cisco Functions	Synopsis
CCiscoPhoneDevSpecific	The CCiscoPhoneDevSpecific class represents the parent class to the following classes.
CCiscoPhoneDevSpecificDataPassThrough	This function allows application to send the Device Specific XSI data through CTI.
CCiscoPhoneDevSpecificAcquire	This function allows application to acquire any CTI-controllable device that can get opened later in superprovider mode.
CCiscoPhoneDevSpecificDeacquire	This function allows application to deacquire a CTI-controllable device that was explicitly acquired.
CCiscoPhoneDevSpecificGetRTPSnapshot	This function allows application to request secure RTP indicator for calls on the device.

CCiscoPhoneDevSpecific

This section provides information on how to perform Cisco TAPI-specific functions with the CCiscoPhoneDevSpecific class, which represents the parent class to all the following classes.

Note Be aware that this virtual class is provided for informational purposes only.

```
CCiscoPhoneDevSpecific
|
+-- CCiscoPhoneDevSpecificDataPassThrough
```

Header File

The file CiscoLineDevSpecific.h contains the constant, structure, and class definition for the Cisco phone device-specific classes.

Class Detail

```
class CCiscoPhoneDevSpecific
{
    public :
        CCiscoPhoneDevSpecific(DWORD msgType):m_MsgType(msgType) {;}
        virtual ~CCiscoPhoneDevSpecific() {;}
        DWORD GetMsgType (void) const { return m_MsgType;}
        void *lpParams(void) const {return (void*)&m_MsgType;}
        virtual DWORD dwSize(void) const = 0;
    private :
        DWORD m_MsgType ;
}
```

Functions

lpParms()

Function that can be used to obtain the pointer to the parameter block

dwSize()

Function that will give the size of the parameter block area

Parameter

m_MsgType

Specifies the type of message.

Subclasses

Each subclass of CCiscoPhoneDevSpecific represents a different value that is assigned to the parameter m_MsgType. If you are using C instead of C++, this represents the first parameter in the structure.

Enumeration

The CiscoPhoneDevSpecificType enumeration includes valid message identifiers.

```
enum CiscoLineDevSpecificType {
CPDST_DEVICE_DATA_PASSTHROUGH_REQUEST = 1
};
```

CCiscoPhoneDevSpecificDataPassThrough

XSI-enabled IP phones allow applications to directly communicate with the phone and access XSI features (for example, manipulate display, get user input, play tone, and so on). To allow TAPI applications to have access to some of these XSI capabilities without having to setup and maintain an independent connection directly to the phone, TAPI will provide the ability to send device data through the CTI interface. This feature gets exposed as a Cisco Unified TSP device-specific extension.

PhoneDevSpecificDataPassthrough request only gets supported for the IP phone devices. Application must open a TAPI phone device with minimum extension version 0x00030000 to make use of this feature.

The CCiscoPhoneDevSpecificDataPassThrough class is used to send the device-specific data to CTI-controlled IP phone devices.

Note Be aware that this extension requires applications to negotiate extension version as 0x00030000.

```
CCiscoPhoneDevSpecific
|
+-- CCiscoPhoneDevSpecificDataPassThrough
```

Class Detail

```
class CCiscoPhoneDevSpecificDataPassThrough : public CCiscoPhoneDevSpecific
{
public:
    CCiscoPhoneDevSpecificDataPassThrough () :
    CCiscoPhoneDevSpecific(CPDST_DEVICE_DATA_PASSTHROUGH_REQUEST)
```

```
    {
      memset((char*)m_DeviceData, 0, sizeof(m_DeviceData)) ;
    }
    virtual ~CCiscoPhoneDevSpecificDataPassThrough() {;}
    // data size determined by MAX_DEVICE_DATA_PASSTHROUGH_SIZE
    TCHAR m_DeviceData[MAX_DEVICE_DATA_PASSTHROUGH_SIZE] ;
    // subtract out the virtual funciton table pointer size
    virtual DWORD dwSize (void) const {return (sizeof (*this)-4) ;}
}
```

Parameters

DWORD m_MsgType

Equals CPDST_DEVICE_DATA_PASSTHROUGH_REQUEST.

DWORD m_DeviceData

This character buffer contains the XML data that is to be sent to phone device.

Note Be aware that MAX_DEVICE_DATA_PASSTHROUGH_SIZE = 2000.

A phone can pass data to an application and it can get retrieved by using PhoneGetStatus (PHONESTATUS:devSpecificData). See PHONESTATUS description for further details.

CCiscoPhoneDevSpecificAcquire

The CCiscoPhoneDevSpecificAcquire class gets used to explicitly acquire any CTI controllable device.

If a Super-provider application needs to open any CTI-controllable device on the Cisco Unified Communications Manager system, the application should explicitly acquire that device by using the preceding interface. After successful response, it can open the device as usual.

Note Be aware that this extension is only available if extension version 0x00070000 or higher is negotiated.

```
CCiscoPhoneDevSpecific
|
+-- CCiscoPhoneDevSpecificAcquire
```

Class Details

```
class CCiscoPhoneDevSpecific Acquire : public CCiscoPhoneDevSpecific
{
    public:
CCiscoPhoneDevSpecificAcquire () : CCiscoPhoneDevSpecific (CPDST_ACQUIRE) {}
        virtual ~ CCiscoPhoneDevSpecificAcquire () {}
        char m_DeviceName[16];
        virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
        // subtract out the virtual function table pointer
};
```

Parameters

DWORD m_MsgType

Equals CPDST_ACQUIRE

char m_DeviceName[16]

The DeviceName that needs to be explicitly acquired.

CCiscoPhoneDevSpecificDeacquire

The CCiscoPhoneDevSpecificDeacquire class gets used to explicitly de-acquire an explicitly acquired device.

If a SuperProvider application explicitly acquired any CTI-controllable device on the Unified Communications Manager system, the application should explicitly de-acquire that device by using this interface.

Note Be aware that this extension is only available if extension version 0x00070000 or higher is negotiated.

```
CCiscoPhoneDevSpecific
|
+-- CCiscoPhoneDevSpecificDeacquire
```

Class Details

```
class CCiscoPhoneDevSpecificDeacquire : public CCiscoPhoneDevSpecific
{
    public:
CCiscoPhoneDevSpecificDeacquire () : CCiscoPhoneDevSpecific (CPDST_ACQUIRE) {}
        virtual ~ CCiscoPhoneDevSpecificDeacquire () {}
        char m_DeviceName[16];
        virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
        // subtract out the virtual function table pointer
};
```

Parameters

DWORD m_MsgType

Equals CPDST_DEACQUIRE

char m_DeviceName[16]

The DeviceName that needs to be explicitly de-acquired.

CCiscoPhoneDevSpecificGetRTPSnapshot

The CCiscoPhoneDevSpecificGetRTPSnapshot class gets used to request Call RTP snapshot event from the device. There will be LineCallDevSpecific event for each call on the device.

Note Be aware that this extension is only available if extension version 0x00070000 or higher is negotiated.

```
CCiscoPhoneDevSpecific
|
+-- CCiscoPhoneDevSpecificGetRTPSnapshot
```

Class Details

```
class CCiscoPhoneDevSpecificGetRTPSnapshot: public CCiscoPhoneDevSpecific
{
    public:
CCiscoPhoneDevSpecificGetRTPSnapshot () : CCiscoPhoneDevSpecific
(CPDST_REQUEST_RTP_SNAPSHOT_INFO) {}
        virtual ~ CCiscoPhoneDevSpecificGetRTPSnapshot () {}
        char m_DeviceName[16];
        virtual DWORD dwSize(void) const {return sizeof(*this)-4;}
        // subtract out the virtual function table pointer
};
```

Parameters

DWORD m_MsgType

Equals CPDST_DEACQUIRE

char m_DeviceName[16]

The DeviceName that needs to be explicitly de-acquired.

Messages

This section describes the line device specific messages that the Cisco Unified TSP supports. An application receives nonstandard TAPI messages in the following LINE_DEVSPECIFIC messages:

- A message to signal when to stop and start streaming RTP audio.
- A message that contains the call handle of active calls when the application starts up.
- A message that indicates to set the RTP parameters based on the data of the message.
- A message that indicates secure media status.

The message type represents an enumerated integer with the following values:

```
enum CiscoLineDevSpecificMsgType
{
    SLDSMT_START_TRANSMISION = 1,
    SLDSMT_STOP_TRANSMISION,
    SLDSMT_START_RECEPTION,
    SLDSMT_STOP_RECEPTION,
    SLDSMT_LINE_EXISTING_CALL,
    SLDSMT_OPEN_LOGICAL_CHANNEL,
    SLDSMT_CALL_TONE_CHANGED,
```

```
    SLDSMT_LINECALLINFO_DEVSPECIFICDATA,
    SLDSMT_NUM_TYPE,
    SLDSMT_LINE_PROPERTY_CHANGED,
    SLDSMT_MONITORING_STARTED,
    SLDSMT_MONITORING_ENDED,
    SLDSMT_RECORDING_STARTED,
    SLDSMT_RECORDING_ENDED
};
```

Start Transmission Events

SLDSMT_START_TRANSMISION

When a message is received, the RTP stream transmission starts and:

- dwParam2 specifies the network byte order IP address of the remote machine to which the RTP stream should be directed.
- dwParam3, specifies the high-order word that is the network byte order IP port of the remote machine to which the RTP stream should be directed.
- dwParam3, specifies the low-order word that is the packet size, in milliseconds, to use.

The application receives these messages to signal when to start streaming RTP audio. At extension version 1.0 (0x00010000), the parameters have the following format:

- dwParam1 contains the message type.
- dwParam2 contains the IP address of the remote machine.
- dwParam3 contains the network byte order IP port of the remote machine to which the RTP stream should be directed in the high-order word and the packet size in milliseconds in the low-order word.

At extension version 2.0 (0x00020000), start transmission uses the following format:

- dwParam1:from highest order bit to lowest
- First two bits blank
- Precedence value 3 bits
- Maximum frames per packet 8 bits
- G723 bit rate 2 bits
- Silence suppression value 1 bit
- Compression type 8 bits
- Message type 8 bits
- dwParam2 contains the IP address of the remote machine
- dwParam3 contains the network byte order IP port of the remote machine to which the RTP stream should be directed in the high-order word and the packet size in milliseconds in the low-order word.

At extension version 4.0 (0x00040000), start transmission has the following format:

- hCall – The call of the Start Transmission event
- dwParam1:from highest order bit to lowest
  - First two bits blank
  - Precedence value 3 bits
  - Maximum frames per packet 8 bits

   - G723 bit rate 2 bits
   - Silence suppression value 1 bit
   - Compression type 8 bits
   - Message type 8 bits
- dwParam2 contains the IP address of the remote machine
- dwParam3 contains the network byte order IP port of the remote machine to which the RTP stream should be directed in the high-order word and the packet size in milliseconds in the low-order word.

Start Reception Events

SLDSMT_START_RECEPTION

When a message is received, the RTP stream reception starts and:

- dwParam2 specifies the network byte order IP address of the local machine to use.
- dwParam3, specifies the high-order word that is the network byte order IP port to use.
- dwParam3, specifies the low-order high-order word that is the packet size, in milliseconds, to use.

When a message is received, the RTP stream reception should commence.

At extension version 1, the parameters have the following format:

- dwParam1 contains the message type.
- dwParam2 contains the IP address of the remote machine.
- dwParam3 contains the network byte order IP port of the remote machine to which the RTP stream should be directed in the high-order word and the packet size in milliseconds in the low-order word.

At extension version 2 start reception uses the following format:

- dwParam1:from highest order bit to lowest
- First 13 bits blank
- G723 bit rate 2 bits
- Silence suppression value 1 bit
- Compression type 8 bits
- Message type 8 bits
- dwParam2 contains the IP address of the remote machine
- dwParam3 contains the network byte order IP port of the remote machine to which the RTP stream should be directed in the high-order word and the packet size in milliseconds in the low-order word.

At extension version 4.0 (0x00040000), start reception uses the following format:

- hCall – The call of the Start Reception event
- dwParam1:from highest order bit to lowest
   - First 13 bits blank
   - G723 bit rate 2 bits
   - Silence suppression value 1 bit
   - Compression type 8 bits
   - Message type 8 bits

- dwParam2 contains the IP address of the remote machine
- dwParam3 contains the network byte order IP port of the remote machine to which the RTP stream should be directed in the high-order word and the packet size in milliseconds in the low-order word.

Stop Transmission Events

SLDSMT_STOP_TRANSMISION

When a message is received, transmission of the streaming should stop.

At extension version 1.0 (0x00010000), stop transmission uses the following format:

- dwParam1 – Message type

At extension version 4.0 (0x00040000), stop transmission uses the following format:

- hCall – The call for which the Stop Transmission event applies.
- dwParam1 – Message type

Stop Reception Events

SLDSMT_STOP_RECEPTION

When a message is received, reception of the streaming should stop.

At extension version 1.0 (0x00010000), stop reception uses the following format:

- dwParam1 - message type

At extension version 4.0 (0x00040000), stop reception uses the following format:

- hCall – The call for which the Stop Reception event applies.
- dwParam1 – Message type

Existing Call Events

SLDSMT_LINE_EXISTING_CALL

These events inform the application of existing calls in the PBX when it starts up. The format of the parameters follows:

- dwParam1 – Message type
- dwParam2 – Call object
- dwParam3 – TAPI call handle

Open Logical Channel Events

SLDSMT_OPEN_LOGICAL_CHANNEL

When a call has media established at a CTI Port or Route Point that is registered for Dynamic Port Registration, receipt of this message indicates that an IP address and UDP port number need to be set for the call.

Note Be aware that this extension is only available if extension version 0x00040000 or higher gets negotiated.

The following format of the parameters applies:

- hCall - The call for which the Open Logical Channel event applies.
- dwParam1 – Message type
- dwParam2 – Compression Type
- dwParam3 – Packet size in milliseconds

At extension version 9.0 (0x00090000), start transmission has the following format:

- hCall - The call the Open Logical Channel event is for
- dwParam1: from highest order bit to lowest
- First eight bits blank
- Maximum frames per packet 8 bits
- Compression type 8 bits
- Message type 8 bits
- dwParam2 contains the IP addressing mode
- dwParam3 is for future use.

LINECALLINFO_DEVSPECIFICDATA Events

SLDSMT_LINECALLINFO_DEVSPECIFICDATA

This message indicates DEVSPECIFICDATA information changed in the DEVSPECIFIC portion of the LINECALLINFO structure for SRTP, QoS, Partition support, call security status, CallAttributeInfo, and CCM CallID.

Note Be aware that SRTP, QoS, Partition support is available only if extension version 0x00070000 or higher is negotiated, and that call security status, CallAttributeInfo and CCM CallID are available only if extension version 0x00080000 or higher is negotiated.

The following format applies for the parameters:

- hCall - The call handle
- dwParam1 - Message type

```
SLDSMT_LINECALLINFO_DEVSPECIFICDATA\
```

- dwParam2 - This bitMask Indicator field applies for SRTP, QoS and Partition.

```
SLDST_SRTP_INFO | SLDST_QOS_INFO | SLDST_PARTITION_INFO |
SLDST_EXTENDED_CALL_INFO|SLDST_CALL_SECURITY_STATUS|SLDST_CALL_ATTRIBUTE_INFO
|SLDST_CCM_CALLID
```

The bit mask values follow:

```
SLDST_SRTP_INFO = 0x00000001
SLDST_QOS_INFO = 0x00000002
SLDST_PARTITION_INFO = 0x00000004
SLDST_EXTENDED_CALL_INFO = 0x00000008
SLDST_CALL_ATTRIBUTE_INFO = 0x00000010
SLDST_CCM_CALLID                      = 0x00000020
|SLDST_CALL_SECURITY_STATUS=0x00000040
```

For example, if there are changes in SRTP and QoS but not in Partition, then both the SLDST_SRTP_INFO and SLDST_QOS_INFO bits will be set. The value for dwParam2 = SLDST_SRTP_INFO | SLDST_QOS_INFO = 0x00000011.

- dwParam3

If a change occurs in the SRTP Information, this field would contain the CiscoSecurityIndicator.

```
enum CiscoSecurityIndicator
{
    SRTP_MEDIA_ENCRYPT_KEYS_AVAILABLE,
    SRTP_MEDIA_ENCRYPT_USER_NOT_AUTH,
    SRTP_MEDIA_ENCRYPT_KEYS_UNAVAILABLE,
    SRTP_MEDIA_NOT_ENCRYPTED
};
```

Note dwParam3 is used when dwParam2 has the SRTP bit mask set.

Call Tone Changed Events

SLDSMT_CALL_TONE_CHANGED

When a tone change occurs on a call, receipt of this message indicates the tone and the feature that caused the tone change.

Note Be aware that this extension is only available if extension version 0x00050000 or higher is negotiated. In the Cisco Unified TSP 4.1 release and later, this event only gets sent for Call Tone Changed Events where the tone equals CTONE_ZIPZIP and the tone gets generated as a result of the FAC/CMC feature.

The format of the parameters follows:

- hCall—The call for which the Call Tone Changed event applies
- dwParam—Message type
- dwParam2—CTONE_ZIPZIP, 0x31 (Zip Zip tone)
- dwParam3—If dwParam2 is CTONE_ZIPZIP, this parameter contains a bitmask with the following possible values:
  - CZIPZIP_FACREQUIRED—If this bit is set, it indicates that a FAC is required.
  - CZIPZIP_CMCREQUIRED—If this bit is set, it indicates that a CMC is required.

Note For a DN that requires both codes, the first event always applies for the FAC and CMC code. The application optionally can send both codes separated by # in the same request. The second event generation remains optional based on what the application sends in the first request.

CHAPTER 7

Cisco Unified TAPI Examples

This chapter provides examples that illustrate how to use the Cisco Unified TAPI implementation. This chapter includes the following subroutines:

- MakeCall
- OpenLine
- CloseLine

MakeCall

```
STDMETHODIMP CTACtrl::MakeCall(BSTR destNumber, long pMakeCallReqID, long hLine, BSTR user2user, long
translateAddr) {
    AFX_MANAGE_STATE(AfxGetStaticModuleState())

    USES_CONVERSION;
    tracer->tracef(SDI_LEVEL_ENTRY_EXIT, "CTACtrl::Makecall %s %d %d %s %d\n",
        T2A((LPTSTR)destNumber), pMakeCallReqID, hLine, T2A((LPTSTR)user2user),
        translateAddr);

    //CtPhoneNo   m_pno;
    CtTranslateOutput   to;

    //LPCSTR  pszTranslatable;
    CString sDialable;

    CString theDestNumber(destNumber);

    CtCall* pCall;
    CtLine* pLine=CtLine::FromHandle((HLINE)hLine);

    if (pLine==NULL) {
        tracer->tracef(SDI_LEVEL_ERROR, "CTACtrl::MakeCall : pLine == NULL\n");
        return S_FALSE;
    } else {
        pCall=new CtCall(pLine);
        pCall->AddSink(this);

        sDialable = theDestNumber;

        if (translateAddr) {
            //m_pno.SetWholePhoneNo((LPCSTR)theDestNumber);
            //pszTranslatable = m_pno.GetTranslatable();
            if (TSUCCEEDED(to.TranslateAddress(pCall->GetLine()->GetDeviceID(),
                (LPCSTR)theDestNumber)) ) {
```

```
                sDialable = to.GetDialableString();
            }
        }
        TRESULT tr = pCall->MakeCall((LPCSTR)sDialable, 0, this);
        if( TPENDING(tr) || TSUCCEEDED(tr)) {
            //GCGC the correct hCall pointer is not being returned yet
            if (translateAddr)
                Fire_MakecallReply(hLine, (long)tr, (long)pCall->GetHandle(),
                    sDialable.AllocSysString());
            else
                Fire_MakecallReply(hLine, (long)tr, (long)pCall->GetHandle(),destNumber);

            return S_OK;
        } else {
            //GCGC delete the call that was created above.
            tracer->tracef(SDI_LEVEL_ERROR, "CTACtrl::MakeCall : pCall->MakeCall failed\n");
            delete pCall;
            return S_FALSE;
         }
    }
}
```

OpenLine

```
STDMETHODIMP CTACtrl::OpenLine(long lDeviceID, BSTR lineDirNumber, long lPriviledges,
                              long lMediaModes, BSTR receiveIPAddress, long lreceivePort) {
    USES_CONVERSION;
    tracer->tracef(SDI_LEVEL_ENTRY_EXIT, "CTACtrl::OpenLine %d %s %d %d %s %d\n",
        lDeviceID, T2A((LPTSTR)lineDirNumber), lPriviledges, lMediaModes,
        T2A((LPTSTR)receiveIPAddress), lreceivePort);

    int lineID;
    TRESULT tr;
    CString strReceiveIP(receiveIPAddress);
    CString strReqAddress(lineDirNumber);

    //bool bTermMedia=((!strReceiveIP.IsEmpty()) && (lreceivePort!=0));
    bool bTermMedia=(((lMediaModes & LINEMEDIAMODE_AUTOMATEDVOICE) != 0)  &&
        (lreceivePort!=0) && (!strReceiveIP.IsEmpty()));
    CtLine* pLine;

    AFX_MANAGE_STATE(AfxGetStaticModuleState())

    tracer->tracef(SDI_LEVEL_DETAILED, "TAC: --> OpenLine()\n");

    if ((lDeviceID<0) && !strcmp((char *)lineDirNumber, "")) {
        tracer->tracef(SDI_LEVEL_ERROR, "TCD: error - bad device ID and no dirn to open\n");
        return S_FALSE;
    }
    lineID=lDeviceID;

    if (lDeviceID<0) {
        //search for line ID in list of lines.
        CtLineDevCaps   ldc;
        int numLines=::TfxGetNumLines();
        for( DWORD nLineID = 0; (int)nLineID < numLines; nLineID++ ) {
            if( /*ShouldShowLine(nLineID) &&*/ TSUCCEEDED(ldc.GetDevCaps(nLineID)) ) {
                CtAddressCaps ac;
                tracer->tracef(SDI_LEVEL_DETAILED, "CTACtrl::OpenLine :
                    Calling ac.GetAddressCaps %d 0\n", nLineID);
                if ( TSUCCEEDED(ac.GetAddressCaps(nLineID, 0)) ) {
```

```
                // GCGC only one address supported
                 CString strCurrAddress(ac.GetAddress());
                 if (strReqAddress==strCurrAddress) {
                    lineID=nLineID;
                    break;
                 }
            }

        } else {
            tracer->tracef(SDI_LEVEL_ERROR, "TAC: error - GetAddressCaps() failed\n");
        }
    }
}

if (lDeviceID<0) {
    tracer->tracef(SDI_LEVEL_ERROR,
        "TAC: error - could not find dirn %s to open line.\n",(LPCSTR)lineDirNumber);
    return S_FALSE;
}

// if we are to do media termination; negotiate the extensions version

DWORD retExtVersion;
if (bTermMedia) {
    TRESULT tr3;
    tracer->tracef(SDI_LEVEL_DETAILED,
        "TAC: lineNegotiateExtVersion - appHandle = %d, deviceID = %d, API ver = %d,
                HiVer = %d, LoVer = %d\n", CtLine::GetAppHandle(), lineID,
                CtLine::GetApiVersion(lineID),
                0x80000000 | 0x00010000L,
                0x80000000 | 0x00020000L );
    tr3=::lineNegotiateExtVersion(CtLine::GetAppHandle(),
                lineID, CtLine::GetApiVersion(lineID),
                0x80000000 | 0x00010000L,    // TAPI v1.3,
                0x80000000 | 0x00020000L,
                &retExtVersion);
    tracer->tracef(SDI_LEVEL_DETAILED,
        "TAC: lineNegotiateExtVersion returned: %d\n", tr3);
}

pLine=new CtLine();
tr=pLine->Open(lineID, this, lPriviledges, lMediaModes);
if( TSUCCEEDED(tr)) {
    if (bTermMedia) {
        if (retExtVersion==0x10000) {
            CiscoLineDevSpecificUserControlRTPStream dsucr;
            dsucr.m_RecievePort=lreceivePort;
            dsucr.m_RecieveIP=::inet_addr((LPCSTR)strReceiveIP);
            TRESULT tr2;

            tr2=::lineDevSpecific(pLine->GetHandle(),
                0,0, dsucr.lpParams(),dsucr.dwSize());
            tracer->tracef(SDI_LEVEL_DETAILED,
                "TAC: lineDevSpecific returned: %d\n", tr2);
        } else {
            //GCGC here put in the new calls to set the media types!
            CiscoLineDevSpecificUserControlRTPStream2 dsucr;
            dsucr.m_RecievePort=lreceivePort;
            dsucr.m_RecieveIP=::inet_addr((LPCSTR)strReceiveIP);
            dsucr.m_MediaCapCount=4;

            dsucr.m_MediaCaps[0].MediaPayload=4;
            dsucr.m_MediaCaps[0].MaxFramesPerPacket=30;
            dsucr.m_MediaCaps[0].G723BitRate=0;
```

```
        dsucr.m_MediaCaps[1].MediaPayload=9;
        dsucr.m_MediaCaps[1].MaxFramesPerPacket=90;
        dsucr.m_MediaCaps[1].G723BitRate=1;
        dsucr.m_MediaCaps[2].MediaPayload=9;
        dsucr.m_MediaCaps[2].MaxFramesPerPacket=90;
        dsucr.m_MediaCaps[2].G723BitRate=2;
        dsucr.m_MediaCaps[3].MediaPayload=11;
        dsucr.m_MediaCaps[3].MaxFramesPerPacket=90;
        dsucr.m_MediaCaps[3].G723BitRate=0;

        TRESULT tr2;

        tr2=::lineDevSpecific(pLine->GetHandle(),
                                0,0, dsucr.lpParams(),dsucr.dwSize());
        tracer->tracef(SDI_LEVEL_DETAILED,
            "TAC: lineDevSpecific returned: %d\n", tr2);
    }
}

CtAddressCaps ac;
LPCSTR  pszAddressName;
if ( TSUCCEEDED(ac.GetAddressCaps(lineID, 0)) ) {
    // GCGC only one address supported
     pszAddressName = ac.GetAddress();
} else {
    pszAddressName = NULL;
    tracer->tracef(SDI_LEVEL_ERROR, "TAC: error - GetAddressCaps() failed.\n");
}

OpenedLine((long)pLine->GetHandle(), pszAddressName, 0);

// now let's try to open the associated phone device
// Get the phone from the line

DWORDnPhoneID;
bool b_phoneFound=false;
CtDeviceID  did;
 if((m_bUsesPhones) && TSUCCEEDED(did.GetID("tapi/phone", pLine->GetHandle())) ) {
     nPhoneID = did.GetDeviceID();
    tracer->tracef(SDI_LEVEL_DETAILED,
        "TAC: Retrieved phone device %d for line.\n",nPhoneID);

    // check to see if phone device is already open

    long hPhone;
    CtPhone* pPhone;
    if (!m_deviceID2phone.Lookup((long)nPhoneID,hPhone)) {
        tracer->tracef(SDI_LEVEL_SIGNIFICANT,
            "TAC: phone device not found in open list, opening it...\n");
        pPhone=new CtPhone();
        TRESULT tr_phone;
        tr_phone=pPhone->Open(nPhoneID,this);
        if (TSUCCEEDED(tr_phone)) {
            ::phoneSetStatusMessages(pPhone->GetHandle(),
                PHONESTATE_DISPLAY | PHONESTATE_LAMP |
                PHONESTATE_HANDSETHOOKSWITCH | PHONESTATE_HEADSETHOOKSWITCH |
                PHONESTATE_REINIT | PHONESTATE_CAPSCHANGE | PHONESTATE_REMOVED,
                PHONEBUTTONMODE_KEYPAD | PHONEBUTTONMODE_FEATURE |
                PHONEBUTTONMODE_CALL |
                PHONEBUTTONMODE_LOCAL | PHONEBUTTONMODE_DISPLAY,
                PHONEBUTTONSTATE_UP | PHONEBUTTONSTATE_DOWN);
            m_phone2line.SetAt((long)pPhone->GetHandle(), (long)pLine->GetHandle());
            m_line2phone.SetAt((long)pLine->GetHandle(),(long)pPhone->GetHandle());
            m_deviceID2phone.SetAt((long)nPhoneID,(long)pPhone->GetHandle());
```

```
                    m_phoneUseCount.SetAt((long)pPhone->GetHandle(), 1);
                } else {
                    tracer->tracef(SDI_LEVEL_ERROR,
                        "TAC: error - phoneOpen failed with code %d\n", tr_phone);
                }
            } else {
                pPhone=CtPhone::FromHandle((HPHONE)hPhone);
                long theCount;

                if (m_phoneUseCount.Lookup((long)pPhone->GetHandle(),theCount))
                    m_phoneUseCount.SetAt((long)pPhone->GetHandle(), theCount+1);
                else {
                    //GCGC this would be an error condition!
                    tracer->tracef(SDI_LEVEL_ERROR,
                        "TAC: error - m_phoneUseCount does not contain phone entry.\n");
                }
            }
        } else {
            tracer->tracef(SDI_LEVEL_ERROR,
                "TAC: error - could not retrieve PhoneID for line.\n");
        }
        tracer->tracef(SDI_LEVEL_DETAILED, "TAC: <-- OpenLine()\n");
        return S_OK;
    } else {
        tracer->tracef(SDI_LEVEL_ERROR, "TAC: error - lineOpen failed: %d\n", tr);
        tracer->tracef(SDI_LEVEL_DETAILED, "TAC: <-- OpenLine()\n");
        OpenLineFailed(tr,0);
        delete pLine;
        return S_FALSE;
    }
}
```

CloseLine

```
STDMETHODIMP CTACtrl::CloseLine(long hLine) {

    AFX_MANAGE_STATE(AfxGetStaticModuleState())

    tracer->tracef(SDI_LEVEL_ENTRY_EXIT, "CTACtrl::CloseLine %d\n", hLine);

    CtLine* pLine;
    pLine=CtLine::FromHandle((HLINE) hLine);

    if (pLine!=NULL) {
            // close the line
            pLine->Close();
            // remove it from the list
            delete pLine;
            long hPhone;
            long theCount;
            if ((m_bUsesPhones) && (m_line2phone.Lookup(hLine,hPhone))) {
                CtPhone* pPhone=CtPhone::FromHandle((HPHONE)hPhone);
                if (pPhone!=NULL) {
                    if (m_phoneUseCount.Lookup(hPhone,theCount))
                        if (theCount>1) {
                            // decrease the number of lines using this phone
                            m_phoneUseCount.SetAt(hPhone,theCount-1);
                        }
                        else {
                            //nobody else is using this phone, so let's remove it.
                            m_deviceID2phone.RemoveKey((long)pPhone->GetDeviceID());
```

```
                        m_phone2line.RemoveKey(hPhone);
                        m_phoneUseCount.RemoveKey(hPhone);

                        //now let's close the phone
                        pPhone->Close();
                    }
                }
                //either way, remove the map entry from line to phone.
                m_line2phone.RemoveKey(hLine);
            }
            return S_OK;
        }
        else
        return S_FALSE;
}
```

APPENDIX A

Message Sequence Charts

This appendix contains message sequences or call scenarios and illustrates a subset of these scenarios that are supported by the Cisco Unified TSP. Be aware that the event order is not guaranteed in all cases and can vary depending on the scenario and the event.

This appendix contains the following sections:

- Logical Partitioning Support, page A-149
- Support for Cisco IP Phone 6900 Series, page A-153

Abbreviations

The following list gives abbreviations that are used in the CTI events that are shown in each scenario:

- NP—Not Present
- LR—LastRedirectingParty
- CH—CtiCallHandle
- GCH—CtiGlobalCallHandle
- RIU—RemoteInUse flag
- DH—DeviceHandle

Manual Outbound Call

Table A-1 describes the message sequences for Manual Outbound Call when party A is idle.

Table A-1 Message Sequences for Manual Outbound Call

Action	CTI Messages	TAPI Messages	TAPI Structures
1. Party A goes off-hook	NewCallEven CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_APPNEWCALL hDevice=A dwCallbackInstance=0 dwParam1=0 dwParam2=hCall-1 dwParam3=OWNER	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialtone, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALTONE dwParam2=UNAVAIL dwParam3=0	No change

Table A-1 Message Sequences for Manual Outbound Call (continued)

2. Party A dials Party B	CallStateChangedEvent, CH=C1, State=Dialing, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALING dwParam2=0 dwParam3=0	No change
3. Party B accepts call	CallStateChangedEvent, CH=C1, State=Proceeding, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=PROCEEDING dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CALLEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Ringback, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=RINGBACK dwParam2=0 dwParam3=0	No change

Table A-1 Message Sequences for Manual Outbound Call (continued)

4. Party B answers call	CallStateChangedEvent, CH=C1, State=Connected, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=ACTIVE dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=B dwRedirectionID=NP dwRedirectionID=NP
	CallStartReceptionEvent, DH=A, CH=C1	LINE_DEVSPECIFIC[1] hDevice=hCall-1 dwCallBackInstance=0 dwParam1=StartReception dwParam2=IP Address dwParam3=Port	No change
	CallStartTransmissionEvent, DH=A, CH=C1	LINE_DEVSPECIFIC[2] hDevice=hCall-1 dwCallBackInstance=0 dwParam1=StartTransmission dwParam2=IP Address dwParam3=Port	No change

1. LINE_DEVSPECIFIC events are sent only if the application has requested them by using lineDevSpecific()
2. LINE_DEVSPECIFIC events are sent only if the application has requested them by using lineDevSpecific()

Blind Transfer

Table A-2 describes the message sequences for Blind Transfer when A calls B, B answers, and A and B are connected.

Table A-2 Message Sequences for Blind Transfer

Action	CTI Messages	TAPI Messages	TAPI Structures
Party B does a lineBlindTranfser() to blind transfer call from party A to party C	**Party A**		
	CallPartyInfoChangedEvent, CH=C1, CallingChanged=False, Calling=A, CalledChanged=True, Called=C, OriginalCalled=B, LR=B, Cause=BlindTransfer	LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTEDID, REDIRECTINGID, REDIRECTIONID	TSPI LINECALLINFO dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NULL dwRedirectingID=NP dwRedirectionID=NP
	Party B		
	CallStateChangedEvent, CH=C2, State=Idle, Reason=Direct, Calling=A, Called=B, OriginalCalled=B, LR=NULL	TSPI: LINE_CALLSTATE lhDevice=hCall-1 dwCallbackInstance=0 dwParam1=IDLE dwParam2=0 dwParam3=0	TSPI LINECALLINFO dwOrigin=INTERNAL dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NULL dwRedirectingID=NULL dwRedirectionID=NULL
	Party C		
	NewCallEvent, CH=C3, origin=Internal_Inbound, Reason=BlindTransfer, Calling=A, Called=C, OriginalCalled=B, LR=B	TSPI: LINE_APPNEWCALL hDevice=C dwCallbackInstance=0 dwParam1=0 dwParam2=hCall-1 dwParam3=OWNER	TSPI LINECALLINFO dwOrigin=INTERNAL dwReason=TRANSFER dwCallerID=A dwCalledID=C dwConnectedID=NULL dwRedirectingID=B dwRedirectionID=C

Table A-2 Message Sequences for Blind Transfer (continued)

Party C is offering	**Party A**		
	CallStateChangeEvent, CH=C1, State=Ringback, Reason=Direct, Calling=A, Called=C, OriginalCalled=B, LR=B	TSPI: LINE_CALLSTATE, hDevice=hCall-1, dwCallbackInstance=0, dwParam1= RINGBACK dwParam2=0 dwParam3=0	TSPI LINECALLINFO dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NULL dwRedirectingID=B dwRedirectionID=C
	Party C		
	CallStateChangedEvent, CH=C3, State=Offering, Reason=BlindTransfer, Calling=A, Called=C, OriginalCalled=B, LR=B	TSPI: LINE_CALLSTATE, hDevice=hCall-1, dwCallbackInstance=0, dwParam1= OFFERING dwParam2=0 dwParam3=0	TSPI LINECALLINFO dwOrigin=INTERNAL dwCallerID=A dwCalledID=C dwConnectedID=NULL dwRedirectingID=B dwRedirectionID=C

Redirect Set Original Called (TxToVM)

Table A-3 describes the message sequences for Redirece Set Original Called (TxToVM) feature where A calls B, B answers, and A and B are connected.

Table A-3 Message Sequences for Redirect Set Original Called (TxToVM)

Action	CTI Messages	TAPI Messages	TAPI Structures
Party B does lineDevSpecific for REDIRECT_SET_ORIG_CALLED with DestDN = C's VMP and SetOrigCalled = C	**Party A**		
	CallPartyInfoChangedEvent, CH=C1, CallingChanged=False, Calling=A, CalledChanged=True, Called=C, OriginalCalled=NULL, LR=NULL, Cause=Redirect	LINE_CALLINFO, hDevice=hCall-1, dwCallbackInstance=0, dwParam1=CONNECTEDID, REDIRECTINGID, REDIRECTIONID	TSPI LINECALLINFO dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=C dwConnectedID=NULL dwRedirectingID=NP dwRedirectionID=NP
	Party B		
	CallStateChangedEvent, CH=C2, State=Idle, reason=DIRECT, Calling=A, Called=B, OriginalCalled=B, LR=NULL	TSPI: LINE_CALLSTATE, hDevice=hCall-1, dwCallbackInstance=0, dwParam1=IDLE dwParam2=0 dwParam3=0	TSPI LINECALLINFO dwOrigin=INTERNAL dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NULL dwRedirectingID=NULL dwRedirectionID=NULL
	Party C's VMP		
	NewCallEvent, CH=C3, origin=Internal_Inbound, reason=Redirect, Calling=A, Called=C, OriginalCalled=C, LR=B	TSPI: LINE_APPNEWCALL hDevice=C dwCallbackInstance=0 dwParam1=0 dwParam2=hCall-1 dwParam3=OWNER	TSPI LINECALLINFO dwOrigin=INTERNAL dwReason=REDIRECT dwCallerID=A dwCalledID=C dwConnectedID=NULL dwRedirectingID=B dwRedirectionID=C's VMP

Table A-3 Message Sequences for Redirect Set Original Called (TxToVM) (continued)

Action	CTI Messages	TAPI Messages	TAPI Structures
Party C is offering	**Party A**		
	CallStateChangeEvent, CH=C1, State=Ringback, Reason=Direct, Calling=A, Called=C, OriginalCalled=C, LR=B	TSPI: LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1= RINGBACK dwParam2=0 dwParam3=0	TSPI LINECALLINFO dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NULL dwRedirectingID=B dwRedirectionID=C's VMP
	Party C		
	CallStateChangedEvent, CH=C3, State=Offering, Reason=Redirect, Calling=A, Called=C, OriginalCalled=C, LR=B	TSPI: LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1= OFFERING dwParam2=0 dwParam3=0	TSPI LINECALLINFO dwOrigin=INTERNAL dwCallerID=A dwCalledID=C dwConnectedID=NULL dwRedirectingID=B dwRedirectionID=C

Shared Lines-Initiating a New Call Manually

Table A-4 describes the message sequences for Shared Lines-Initiating a new call manually where Party A and Party A' represent shared line appearances. Also, Party A and Party A' are idle.

Table A-4 Message Sequences for Shared Lines-Initiating a New Call Manually

Action	CTI Messages	TAPI Messages	TAPI Structures
1. Party A goes off-hook	NewCallEvent, CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct, RIU=false	LINE_APPNEWCALL hDevice=A dwCallbackInstance=0 dwParam1=0 dwParam2=hCall-1 dwParam3=OWNER	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialtone, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP, RIU=false	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALTONE dwParam2=UNAVAIL dwParam3=0	No change
	Party A'		
	NewCallEvent, CH=C1, GCH=G1, Calling=A', Called=NP, OrigCalled=NP, LR=NP, S tate=Dialtone, Origin=OutBound, Reason=Direct, RIU=true	LINE_APPNEWCALL hDevice=A' dwCallbackInstance=0 dwParam1=0 dwParam2=hCall-2 dwParam3=OWNER	LINECALLINFO (hCall-2) hLine=A' dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A' dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialtone, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP, RIU=true	LINE_CALLSTATE hDevice=hCall-2 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=INACTIVE dwParam3=0	No change

Table A-4 Message Sequences for Shared Lines-Initiating a New Call Manually (continued)

2. Party A dials Party B	**Party A**		
	CallStateChangedEvent, CH=C1, State=Dialing, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP, RIU=false	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALING dwParam2=0 dwParam3=0	No change
	Party A′		
	None	None	None
3. Party B accepts call	**Party A**		
	CallPartyInfoChangedEvent , CH=C1, CallingChanged=False, Calling=A, CalledChanged=true, Called=B, Reason=Direct, RIU=false	Ignored	No change
	CallStateChangedEvent, CH=C1, State=Proceeding, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP, RIU=false	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=PROCEEDING dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1= CALLERID, CALLEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Ringback, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP, RIU=false	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=RINGBACK dwParam2=0 dwParam3=0	No change

Table A-4 Message Sequences for Shared Lines-Initiating a New Call Manually (continued)

3. Party B accepts call (continued)	**Party A'**		
	CallPartyInfoChangedEvent , CH=C1, CallingChanged=False, Calling=A', CalledChanged=true, Called=B, Reason=Direct, RIU=true	Ignored	No change
	CallStateChangedEvent, CH=C1, State=Proceeding, Cause=CauseNoError, Reason=Direct, Calling=A', Called=B, OrigCalled=B, LR=NP, RIU=true	LINE_CALLSTATE hDevice=hCall-2 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=INACTIVE dwParam3=0 LINE_CALLINFO hDevice=hCall-2 dwCallbackInstance=0 dwParam1= CALLERID, CALLEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-2) hLine=A' dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A' dwCalledID=B dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Ringback, Cause=CauseNoError, Reason=Direct, Calling=A', Called=B, OrigCalled=B, LR=NP, RIU=true	LINE_CALLSTATE hDevice=hCall-2 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=INACTIVE dwParam3=0	No change

Table A-4 Message Sequences for Shared Lines-Initiating a New Call Manually (continued)

4. Party B answers call	**Party A**		
	CallStateChangedEvent, CH=C1, State=Connected, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP, RIU=false	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=ACTIVE dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTEDI D dwParam2=0, dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=B dwRedirectionID=NP dwRedirectionID=NP
	Party A'		
	CallStateChangedEvent, CH=C1, State=Connected, Cause=CauseNoError, Reason=Direct, Calling=A', Called=B, OrigCalled=B, LR=NP, RIU=true	LINE_CALLSTATE hDevice=hCall-2 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=INACTIVE dwParam3=0 LINE_CALLINFO hDevice=hCall-2 dwCallbackInstance=0 dwParam1=CONNECTEDI D dwParam2=0, dwParam3=0	LINECALLINFO (hCall-2) hLine=A' dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A' dwCalledID=B dwConnectedID=B dwRedirectionID=NP dwRedirectionID=NP

Presentation Indication

Making a Call Through Translation Pattern

Table A-5 describes the message sequences for the Presentation Indication scenario of making a call through translation pattern. In the Translation Pattern admin pages, both the callerID/Name and ConnectedID/Name get set to "Restricted".

Table A-5 Message Sequences for Making a Call Through Translation Pattern

Action	CTI Messages	TAPI Messages	TAPI Structures
Party A goes off-hook	NewCallEvent, CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_APPNEWCALL hDevice=A dwCallbackInstance=0 dwParam1=0 dwParam2=hCall-1 dwParam3=OWNER	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialtone, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALTONE dwParam2=UNAVAIL dwParam3=0	No change
Party A dials Party B through Translation pattern	CallStateChangedEvent, CH=C1, State=Dialing, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALING dwParam2=0 dwParam3=0	No change
Party B accepts the call	CallStateChangedEvent, CH=C1, State=Proceeding, Cause=CauseNoError, Reason=Direct, Calling=A, CallingPartyPI=Allowed, Called=B, CalledPartyPI= Restricted, OrigCalled=B, OrigCalledPI=restricted, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1= PROCEEDING dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CALLEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCallerIDName=A's Name dwCalledID=B dwCalledIDName=B’s name dwConnectedID=NP dwConnectedIDName=NP dwRedirectionID=NP dwRedirectionIDName=NP dwRedirectionID=NP dwRedirectionIDName=NP

Table A-5 Message Sequences for Making a Call Through Translation Pattern (continued)

Party B accepts the call (continued)	CallStateChangedEvent, CH=C1, State=Ringback, Cause=CauseNoError, Reason=Direct, Calling=A, CallingPI = Allowed, Called=B, CalledPI = Restricted, OrigCalled=B, OrigCalledPI = Restricted, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=RINGBACK dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedIDFlags = LINECALLPARTYID_ BLOCKED dwConnectedID=NP dwRedirectionID=NP dwRedirectionIDFlags = LINECALLPARTYID_ BLOCKED dwRedirectionID=NP
Party B answers the call	CallStateChangedEvent, CH=C1, State=Connected, Cause=CauseNoError, Reason=Direct, Calling=A, CallingPI = Allowed, Called=B, CalledPI = Restricted, OrigCalled=B, OrigCalledPI = Restricted, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=ACTIVE dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCallerIDName=A's Name dwCalledID=B dwCalledIDName=B's Name dwConnectedID=A, dwConnectedIDName= A's Name, dwRedirectingID=NP dwRedirectingIDName=NP dwRedirectionIDFlags = LINECALLPARTYID_ BLOCKED dwRedirectionID=NP dwRedirectionIDName=NP
	CallStartReceptionEvent, DH=A, CH=C1	LINE_DEVSPECIFIC[1] hDevice=hCall-1 dwCallBackInstance=0 dwParam1= StartReception dwParam2=IP Address dwParam3=Port	No change
	CallStartTransmissionEvent, DH=A, CH=C1	LINE_DEVSPECIFIC[1] hDevice=hCall-1 dwCallBackInstance=0 dwParam1= StartTransmission dwParam2=IP Address dwParam3=Port	No change

1. LINE_DEVSPECIFIC events only get sent if the application requested them by using lineDevSpecific().

Blind Transfer Through Translation Pattern

Table A-6 describes the message sequences for the Presentation Indication scenario of Blind Transfer through Translation Pattern. In this scenario, A calls via translation pattern B, B answers, and A and B are connected.

Table A-6 Message Sequences for Blind Transfer Through Translation Pattern

Action	CTI Messages	TAPI Messages	TAPI Structures
Party B does a lineBlindTranfser() to blind transfer call from party A to party C via translation pattern	**Party A**		
	CallPartyInfoChangedEvent, CH=C1, CallingChanged=False, Calling=A, CallingPartyPI=Restricted, CalledChanged=True, Called=C, CalledPartyPI=Restricted, OriginalCalled=NULL, OriginalCalledPI=Restricted, LR=NULL, Cause=BlindTransfer	LINE_CALLINFO, hDevice=hCall-1, dwCallbackInstance=0, dwParam1=CONNECTEDID, REDIRECTINGID, REDIRECTIONID	TSPI LINECALLINFO dwOrigin=OUTBOUND dwReason=DIRECT dwCallerIDFlags = LINECALLPARTYID_BLOCKED dwCallerID=NP dwCallerIDName=NP dwCalledID=B dwCalledIDName=B's name dwConnectedIDFlags = LINECALLPARTYID_BLOCKED dwConnectedID=NP dwConnectedIDName=NP dwRedirectingID=B dwRedirectingIDName=B's name dwRedirectionIDFlags = LINECALLPARTYID_BLOCKED dwRedirectionID=NP dwRedirectionIDName=NP
	Party B		

Table A-6 Message Sequences for Blind Transfer Through Translation Pattern (continued)

<table>
<tr><td></td><td>CallStateChangedEvent,
CH=C2,
State=Idle, Reason=Direct,
Calling=A,
CallingPartyPI=Restricted,
Called=B,
CalledPartyPI=Restricted,
OriginalCalled=B,
OrigCalledPartyPI=Restricted, LR=NULL</td><td>TSPI: LINE_CALLSTATE,
hDevice=hCall-1,
dwCallbackInstance=0,
dwParam1=IDLE
dwParam2=0
dwParam3=0</td><td>TSPI LINECALLINFO
dwOrigin=INTERNAL
dwReason=DIRECT
dwCallerIDFlags = LINECALLPARTYID_BLOCKED
dwCallerID=NP
dwCallerIDName=NP
dwCalledID=B
dwCalledIDName=B’s name
dwConnectedIDFlags = LINECALLPARTYID_BLOCKED
dwConnectedID=NP
dwConnectedIDName=NP
dwRedirectingID=B
dwRedirectingIDName= B’s name
dwRedirectionIDFlags = LINECALLPARTYID_BLOCKED
dwRedirectionID=NP
dwRedirectionIDName=NP</td></tr>
<tr><td rowspan="2">Party B does a lineBlindTranfser() to blind transfer call from party A to party C via translation pattern (continued)</td><td colspan="3">Party C</td></tr>
<tr><td>NewCallEvent,
CH=C3,
origin=Internal_Inbound,
Reason=BlindTransfer,
Calling=A,
CallingPartyPI=Restricted,
Called=C,
CalledPartyPI=Restricted,
OriginalCalled=B,
OrigCalledPartyPI=Restricted,
LR=B,
LastRedirectingPartyPI= Restricted</td><td>TSPI:
LINE_APPNEWCALL
hDevice=C
dwCallbackInstance=0
dwParam1=0
dwParam2=hCall-1
dwParam3=OWNER</td><td>TSPI LINECALLINFO
dwOrigin=INTERNAL
dwReason=TRANSFER
dwCallerIDFlags = LINECALLPARTYID_BLOCKED
dwCallerID=NP
dwCallerIDName=NP
dwCalledID=NP
dwCalledIDName=NP
dwConnectedIDFlags = LINECALLPARTYID_BLOCKED
dwConnectedID=NP
dwConnectedIDName=NP
dwRedirectingID=B
dwRedirectingIDName= B's name
dwRedirectionIDFlags = LINECALLPARTYID_BLOCKED
dwRedirectionID=NP
dwRedirectionIDName=NP</td></tr>
</table>

Table A-6 Message Sequences for Blind Transfer Through Translation Pattern (continued)

Party C is offering	**Party A**		
	CallStateChangeEvent, CH=C1, State=Ringback, Reason=Direct, Calling=A, CallingPartyPI=Restricted, Called=C, CalledPartyPI=Restricted, OriginalCalled=B, OrigCalledPartyPI=Restricted, LR=B, LastRedirectingPartyPI= Restricted	TSPI: LINE_CALLSTATE, hDevice=hCall-1, dwCallbackInstance=0, dwParam1= RINGBACK dwParam2=0 dwParam3=0	TSPI LINECALLINFO dwOrigin=OUTBOUND dwReason=DIRECT dwCallerIDFlags = LINECALLPARTYID_ BLOCKED dwCallerID=NP dwCallerIDName=NP dwCalledID=B dwCalledIDName=B's name dwConnectedIDFlags = LINECALLPARTYID_ BLOCKED dwConnectedID=NP dwConnectedIDName=NP dwRedirectingID=B dwRedirectingIDName= B's name dwRedirectionIDFlags = LINECALLPARTYID_ BLOCKED dwRedirectionID=NP dwRedirectionIDName=NP
Party C is offering (continued)	**Party C**		
	CallStateChangedEvent, CH=C3, State=Offering, Reason=BlindTransfer, Calling=A, CallingPartyPI=Restricted, Called=C, CalledPartyPI=Restricted, OriginalCalled=B, OrigCalledPartyPI=Restricted, LR=B, LastRedirectingPartyPI= Restricted	TSPI: LINE_CALLSTATE, hDevice=hCall-1, dwCallbackInstance=0, dwParam1= OFFERING dwParam2=0 dwParam3=0	TSPI LINECALLINFO dwOrigin=INTERNAL dwCallerIDFlags = LINECALLPARTYID_ BLOCKED dwCallerID=NP dwCallerIDName=NP dwCalledID=NP dwCalledIDName=NP dwConnectedIDFlags = LINECALLPARTYID_ BLOCKED dwConnectedID=NP dwConnectedIDName=NP dwRedirectingID=B dwRedirectingIDName= B's name dwRedirectionIDFlags = LINECALLPARTYID_ BLOCKED dwRedirectionID=NP dwRedirectionIDName=NP

Forced Authorization and Client Matter Code Scenarios

Manual Call to a Destination that Requires an FAC

Table A-7 describes the message sequences for the Forced Authorization and Client Matter Code scenario of Manual Call to a Destination that requires an FAC.

Preconditions

Party A is Idle. Party B requires an FAC.

The scenario remains similar if Party B requires a CMC instead of an FAC.

Table A-7 Message Sequences for Manual Call to a Destination that Requires an FAC

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A goes off-hook	NewCallEvent, CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_APPNEWCALL hDevice=A dwCallbackInstance=0 dwParam1=0 dwParam2=hCall-1 dwParam3=OWNER	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialtone, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALTONE dwParam2=UNAVAIL dwParam3=0	No change

Table A-7 Message Sequences for Manual Call to a Destination that Requires an FAC (continued)

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A dials Party B	CallStateChangedEvent, CH=C1, State=Dialing, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALING dwParam2=0 dwParam3=0	No change
	CallToneChangedEvent, CH=C1, Tone=ZipZip, Feature=FACCMC, FACRequired=True, CMCRequired=False	LINE_DEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1=SLDSMT_CAL L_TONE_CHANGED dwParam2=CTONE_ZIPZI P dwParam3= CZIPZIP_FACREQUIRED	No change
Party A dials the FAC, and Party B accepts the call	CallStateChangedEvent, CH=C1, State=Proceeding, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=PROCEEDING dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CALLEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Ringback, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=RINGBACK dwParam2=0 dwParam3=0	No change

Manual Call to a Destination that Requires both FAC and CMC

Table A-8 describes the message sequences for the Forced Authorization and Client Matter Code scenario of a manual call to a destination that requires both FAC and CMC.

Preconditions

Party A is Idle. Party B requires an FAC and a CMC.

Table A-8 Message Sequences for Manual Call to a Destination that Requires both FAC and CMC

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A goes off-hook	NewCallEvent, CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_APPNEWCALL hDevice=A dwCallbackInstance=0 dwParam1=0 dwParam2=hCall-1 dwParam3=OWNER	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialtone, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALTONE dwParam2=UNAVAIL dwParam3=0	No change
Party A dials Party B	CallStateChangedEvent, CH=C1, State=Dialing, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALING dwParam2=0 dwParam3=0	No change
	CallToneChangedEvent, CH=C1, Tone=ZipZip, Feature=FACCMC, FACRequired=True, CMCRequired=True	LINE_DEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1=SLDSMT_CAL L_TONE_CHANGED dwParam2=CTONE_ZIPZI P dwParam3= CZIPZIP_FACREQUIRED, CZIPZIP_CMCREQUIRED	No change
Party A dials the FAC	CallToneChangedEvent, CH=C1, Tone=ZipZip, Feature=FACCMC, FACRequired=False, CMCRequired=True	LINE_DEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1=SLDSMT_CAL L_TONE_CHANGED dwParam2=CTONE_ZIPZI P dwParam3= CZIPZIP_CMCREQUIRED	No change

Table A-8 Message Sequences for Manual Call to a Destination that Requires both FAC and CMC (continued)

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A dials the CMC, and Party B accepts the call	CallStateChangedEvent, CH=C1, State=Proceeding, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=PROCEEDING dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CALLEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Ringback, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=RINGBACK dwParam2=0 dwParam3=0	No change

lineMakeCall to a Destination that Requires an FAC

Table A-9 describes the message sequences for the Forced Authorization and Client Matter Code scenario of lineMakeCall to a destination that requires an FAC.

Preconditions

Party A is Idle. Party B requires an FAC. Note that the scenario is similar if Party requires a CMC instead of an FAC.

Table A-9 Message Sequences for lineMakeCall to a Destination that Requires an FAC

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A does a lineMakeCall() to Party B	NewCallEvent, CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=ORIGIN dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1= REASON, CALLERID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialing, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALING dwParam2=0 dwParam3=0	No change
	CallToneChangedEvent, CH=C1, Tone=ZipZip, Feature=FACCMC, FACRequired=True, CMCRequired=False	LINE_DEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1=SLDSMT_CALL_TONE_CHANGED dwParam2=CTONE_ZIPZIP dwParam3= CZIPZIP_FACREQUIRED	No change

Table A-9 Message Sequences for lineMakeCall to a Destination that Requires an FAC (continued)

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A does a lineDial() with the FAC in the dial string and Party B accepts the call	NewCallEvent, CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=PROCEEDING dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CALLEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Ringback, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=RINGBACK dwParam2=0 dwParam3=0	No change

lineMakeCall to a Destination that Requires Both FAC and CMC

Table A-10 describes the message sequences for the Forced Authorization and Client Matter Code scenario of lineMakeCall to a destination that requires both FAC and CMC. In this scenario, Party A is Idle and Party B requires both an FAC and a CMC.

Table A-10 Message Sequences for lineMakeCall to a Destination that Requires Both FAC and CMC

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A does a lineMakeCall() to Party B	NewCallEvent, CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=ORIGIN dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1= REASON, CALLERID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialing, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALING dwParam2=0 dwParam3=0	No change
	CallToneChangedEvent, CH=C1, Tone=ZipZip, Feature=FACCMC, FACRequired=True, CMCRequired=True	LINE_DEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1=SLDSMT_CAL L_TONE_CHANGED dwParam2=CTONE_ZIPZI P dwParam3= CZIPZIP_FACREQUIRED, CZIPZIP_CMCREQUIRED	No change
Party A does a lineDial() with the FAC in the dial string	CallToneChangedEvent, CH=C1, Tone=ZipZip, Feature=FACCMC, FACRequired=False, CMCRequired=True	LINE_DEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1=SLDSMT_CAL L_TONE_CHANGED dwParam2=CTONE_ZIPZI P dwParam3= CZIPZIP_CMCREQUIRED	No change

Table A-10 Message Sequences for lineMakeCall to a Destination that Requires Both FAC and CMC (continued)

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A does a lineDial() with the CMC in the dial string and Party B accepts the call	CallStateChangedEvent, CH=C1, State=Proceeding, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=PROCEEDING dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CALLEDID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Ringback, Cause=CauseNoError, Reason=Direct, Calling=A, Called=B, OrigCalled=B, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=RINGBACK dwParam2=0 dwParam3=0	No change

Timeout Waiting for FAC or Invalid FAC

Table A-11 describes the message sequences for the Forced Authorization and Client Matter Code scenario of timeout waiting for FAC or invalid FAC entered. Here, Party A is Idle and Party B requires an FAC.

The scenario remains similar if Party B required a CMC instead of a FAC.

Table A-11 Message Sequences for Timeout Waiting for FAC or Invalid FAC

Actions	CTI Message	TAPI Messages	TAPI Structures
Party A does a lineMakeCall() to Party B	NewCallEvent, CH=C1, GCH=G1, Calling=A, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=ORIGIN dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1= REASON, CALLERID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP
	CallStateChangedEvent, CH=C1, State=Dialing, Cause=CauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DIALING dwParam2=0 dwParam3=0	No change
	CallToneChangedEvent, CH=C1, Tone=ZipZip, Feature=FACCMC, FACRequired=True, CMCRequired=False	LINE_DEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1=SLDSMT_CAL L_TONE_CHANGED dwParam2=CTONE_ZIPZI P dwParam3= CZIPZIP_FACREQUIRED	No change
T302 timer times out waiting for digits, or Party A does a lineDial() with an invalid FAC	CallStateChangedEvent, CH=C1, State=Disconnected, Cause= CtiNoRouteToDDestination , Reason=FACCMC, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=DISCONNECT ED dwParam2=DISCONNECT MODE_FACCMC[1] dwParam3=0	No change
	CallStateChangedEvent, CH=C1, State=Idle, Cause=CtiCauseNoError, Reason=Direct, Calling=A, Called=NP, OrigCalled=NP, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=IDLE dwParam2=0 dwParam3=0	No change

1. dwParam2 get set to DISCONNECTMODE_FACCMC if the extension version on the line is set to at least 0x00050000. Otherwise, dwParam2 get set to DISCONNECTMODE_UNAVAIL.

Refer and Replaces Scenarios

In-Dialog Refer - Referrer in Cisco Unified Communications Manager Cluster

Table A-12 describes the message sequences for the Refer and Replaces scenario of in-dialog refer where referer is in Cisco Unified Communications Manager cluster.

Table A-12 Message Sequences for In-Dialog Refer - Referrer in Cisco Unified Communications Manager Cluster

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referee (B)	CallState/CallInfo @Refer-to-Target (C)
Referrer (A), Referee (B), and Refer-to-Target (C) exist in Cisco Unified Communications Manager cluster, and CTI is monitoring those lines	A-->B has a call in connected state. The call party information at A should be {calling=A, called=B, LRP=null, origCalled=B, reason=direct} TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = B dwReason = Direct dwOrigin =LINECALL ORIGIN_INTERNAL	A-->B has a call in connected state. The call party information at B should be {calling=A, called=B, LRP=null, origCalled=B, reason=direct} TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = A dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	

Table A-12 Message Sequences for In-Dialog Refer - Referrer in Cisco Unified Communications Manager Cluster (continued)

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referee (B)	CallState/CallInfo @Refer-to-Target (C)
(A) initiates REFER (B) to (C)	A gets LINECALLSTATE_ UNKNOWN | CLDSMT_ CALL_WAITING_STATE with extended reason = REFER TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = B dwReason = Direct dwOrigin =LINECALL ORIGIN_INTERNAL		NewCallEvent should be {calling=B, called=C, LRP=A, origCalled=C, reason=REFER} LINECALLSTATE_OFFERING TAPI CallInfo dwCallerID = B dwCalledID = C dwRedirectingID = A dwRedirectionID = C dwConnectedID = “” dwReason =LINECALL REASON_UNKNOWN with extended REFER dwOrigin = LINECALL ORIGIN_INTERNAL
C answers the call, and Refer is successful	LINECALLSTATE_IDLE with extended REFER reason	CallPartyInfoChangedEvent @ B with {calling=B, called=C, LRP=A, origCalled=C, reason=REFER} TAPI callInfo dwCallerID = B dwCalledID = B dwRedirectingID = A dwRedirectionID = C dwConnectedID = C dwReason = DIRECT dwOrigin = LINECALL ORIGIN_INTERNAL	LINECALLSTATE_CONNECTED TAPI callInfo dwCallerID = B dwCalledID = C dwRedirectingID = A dwRedirectionID = C dwConnectedID = B dwReason = LINECALL REASON_UNKNOWN with extended REFER dwOrigin = LINECALL ORIGIN_INTERNAL

In-Dialog Refer Where ReferToTarget Redirects the Call in Offering State

Table A-13 describes the message sequences for the Refer and Replaces scenario of in-dialog refer where ReferToTarget redirects the call in Offering state.

Table A-13 Message Sequences for In-Dialog Refer Where ReferToTarget Redirects the Call in Offering State

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referee (B)	CallState/CallInfo @Refer-to-Target (C)
Referrer (A), Referee (B), and Refer-to-Target (C) exist in Cisco Unified Communications Manager cluster, and CTI is monitoring those lines	A-->B has a call in connected state. The call party information at A should be {calling=A, called=B, LRP=null, origCalled=B, reason=direct} TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = B dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	A-->B has a call in connected state. The call party information at B should be {calling=A, called=B, LRP=null, origCalled=B, reason=direct} TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = A dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	

Table A-13 Message Sequences for In-Dialog Refer Where ReferToTarget Redirects the Call in Offering State (continued)

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referee (B)	CallState/CallInfo @Refer-to-Target (C)
(A) initiates REFER (B) to (C)	A gets LINECALLSTATE_ UNKNOWN | CLDSMT_ CALL_WAITING_STATE with extended reason = REFER TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = B dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	B gets CPIC with (calling = B, called = C, ocdpn=C, LRP = A, reason = REFER, call state = Ringback) TAPI CallInfo dwCallerID = B dwCalledID = C dwRedirectingID = A dwRedirectionID = C dwConnectedID = null dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	NewCallEvent should be {calling=B, called=C, LRP=A, origCalled=C, reason=REFER} LINECALLSTATE_OFFER ING TAPI callInfo dwCallerID = B dwCalledID = C dwRedirectingID = A dwRedirectionID = C dwConnectedID = null dwReason = LINECALL REASON_UNKNOWN with extended REFER dwOrigin = LINECALL ORIGIN_INTERNAL
C Redirects the call to D in offering state, and D answers	LINECALLSTATE_IDLE with extended reason = REFER (REFER considered as successful when D answers)	CallPartyInfoChangedEvent @ B with {calling=B, called=D, LRP=C, origCalled=C, reason=Redirect} Callstate = connected TAPI callInfo dwCallerID = B dwCalledID = B dwRedirectingID = C dwRedirectionID = D dwConnectedID = D dwReason = DIRECT dwOrigin = LINECALL ORIGIN_INTERNAL	IDLE with reason = Redirect TAPI LINECALLSTATE_IDLE D will get NewCallEvent with reason = Redirect call info same as B's call info. (calling=B, called=D, ocdpn = C, LRP = C, reason = redirect) Offering/accepted/connecte d

In-Dialog Refer Where Refer Fails or Refer to Target is Busy

Table A-14 describes the message sequences for the Refer and Replaces scenario of in-dialog refer fails or refer to target is busy.

Table A-14 Message Sequences for In-Dialog Refer Where Refer Fails or Refer to Target is Busy

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referee (B)	CallState/CallInfo @Refer-to-Target (C)
Referrer (A), Referee (B,) and Refer-to-Target (C) exist in Cisco Unified Communications Manager cluster, and CTI is monitoring those lines	A-->B has a call in connected state. The call party information at A should be {calling=A, called=B, LRP=null, origCalled=B, reason=direct} TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = B dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	A-->B has a call in connected state. The call party information at B should be {calling=A, called=B, LRP=null, origCalled=B, reason=direct} TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = A dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	
(A) initiates REFER (B) to (C)	A gets LINECALLSTATE_ UNKNOWN | CLDSMT_ CALL_WAITING_STATE with extended reason = REFER TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = B dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	No change	
C is busy / C does not answer	A gets LINECALLSTATE_ CONNECTED with extended reason = REFER (REFER considered as **failed)**	If B goes to ringback when call is offered to C (C does not answer finally) it should also receive Connected Call State and CPIC event TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = A dwReason = Direct dwOrigin = LINECALL ORIGIN_INTERNAL	

Out-of-Dialog Refer

Table A-15 describes the message sequences for the Refer and Replaces scenario of out-of-dialog refer.

Table A-15 Message Sequences for Out-of-Dialog Refer

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referee (B)	CallState/CallInfo @Refer-to-Target (C)
Referrer (A), Referee (B), and Refer-to-Target (C) exist in Cisco Unified Communications Manager cluster, and CTI is monitoring those lines	There is no preexisting call between A and B.	There is no preexisting call between A and B.	
A initiates REFER B to (C)		B should get NewCallEvent with call info as {calling=A, called=B, LRP=null, origCalled=B, reason=REFER} TAPI CallInfo dwCallerID = A dwCalledID = B dwRedirectingID = null dwRedirectionID = null dwConnectedID = A dwReason = LINECALL REASON_ UNKNOWN with extended REFER dwOrigin =LINECALL ORIGIN_EXTERNAL	

Table A-15 Message Sequences for Out-of-Dialog Refer (continued)

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referee (B)	CallState/CallInfo @Refer-to-Target (C)
B answers		Call state = connected (media does not flow between A and B when call goes to connected state) TAPI CallInfo (no change)	
Cisco Unified Communications Manager redirects the call to C		CallPartyInfoChangedEvent @ B with {calling=B, called=C, LRP=A, origCalled=C, reason=REFER} TAPI callInfo dwCallerID = B dwCalledID = B dwRedirectingID = A dwRedirectionID = C dwConnectedID = C dwReason = LINECALL REASON_ UNKNOWN with extended REFER dwOrigin = LINECALL ORIGIN_EXTERNAL	NewCallEvent should be {calling=B, called=C, LRP=A, origCalled=C, reason=REFER} This info is exactly same as though caller (A) performed REDIRECT operation (except the reason is different here). TAPI callInfo dwCallerID = B dwCalledID = C dwRedirectingID = A dwRedirectionID = C dwConnectedID = B dwReason = LINECALL REASON_ UNKNOWN with extended REFER dwOrigin = LINECALL ORIGIN_INTERNAL

Invite with Replace for Confirmed Dialog

Table A-16 describes the message sequences for the Refer and Replaces scenario of invite with replace for confirmed dialog. Here, A, B, and C exist inside Cisco Unified Communications Manager. A confirmed dialog occurs between A and B. C initiates Invite to A with replace B's dialog ID.

Table A-16 Message Sequences for Invite with Replace for Confirmed Dialog

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referree (B)	CallState/CallInfo @Refer-to-Target (C)
Confirmed dialog occurs between A and B	Call State = connected, Caller=A, Called=B, Connected=B, Reason =direct, gcid = GC1	Call State = connected Caller=A, Called=B, Connected=A, Reason =direct, gcid = GC1	
C Invites A by replacing B's dialog			NewCall at C gcid = GC2, reason=REPLACEs, Call state = Dialing, Caller=C, Called=null, Reason = REPLACEs
Cisco Unified Communications Manager joins A and C in a call and disconnects call leg @ B	GCID Changed to GC2, Reason = REPLACEs CPIC Caller = C, Called = A, ocdpn = A, LRP = B Reason = REPLACEs Callstate = connected TAPI callinfo caller=C, called=B, connected=C, redirecting=B, redirection=A, reason=DIRECT with extended REPLACEs, callID=GC2	Call State = IDLE, extended reason = REPLACEs	CPIC changed Caller = C, Called = A, ocdpn = A, LRP = B, Reason=REPLACEs CallState = connected TAPI callinfo Caller=C, Called=A, Connected=A, Redirecting=B, Redirection=A, reason=UNKNOWN with extended REPLACEs, callID=GC2

Refer with Replace for All in Cluster

Table A-17 describes the message sequences for the Refer and Replaces scenario of refer with replace for all in cluster. Here, a confirmed dialog exists between A and B and A and C. A initiates Refer to C with replace B's dialog ID.

Table A-17 Message Sequences for Refer with Replace for All in Cluster

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referree (B)	CallState/CallInfo @Refer-to-Target (C)
Dialog between A and B and dialog between A and C	Call State = onhold, GC1, Caller=A, Called=C, Connected=C, Reason =direct CallState = connected, GC2, Caller = A, Called = B, Connected=B, Reason =direct	Call State = connected Caller=A, Called=B, Connected=A, Reason =direct, gcid = GC2	Call State = connected Caller=A, Called=C, Connected=A, Reason =direct, gcid = GC1
A completes Refer to C replacing A->B's dialog (B is referred to target)	From CTI (callState = IDLE with reason = TRANSFER) TAPI call state IDLE with Reason = DIRECT with extended reason TRANSFER	GCID changed from CTI reason = TRANSFER CPIC Changed from CTI Caller=B, Called=C, Origcalled = C, LRP=A, Reason=TRANSFER TAPI callinfo Caller=B, Called=B, Connected = C, Redirecting=A, Redirection=C, Reason = DIRECT with extended reason TRANSFER. CallId=GC1	CPIC Changed from CTI with Caller=B, Called=C, Origcalled = C, LRP=A, Reason=TRANSFER TAPI callinfo caller=B, called=C, connected=B, redirecting=A, redirection=C,reason=direct with extended TRANSFER. callId=GC1

Refer with Replace for All in Cluster, Replace Dialog Belongs to Another Station

Table A-17 describes the message sequences for the Refer and Replaces scenario of refer with replace for all in cluster, where replace dialog belongs to another station. In this scenario:

A is Referrer, D is Referee, and C is Refer-to-Target.

A confirmed dialog exists between A(d1) and B & C(d2) and D.

A initiates Refer to D on (d1) with Replaces (d2).

Table A-18 Message Sequences for Refer with Replace for All in Cluster, Replace Dialog Belongs to Another Station

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @B	CallState/CallInfo @Refer-to-Target (C)	CallState/CallInfo @Referree (D)
Dialog between A and B and dialog between C and D	Call State = onhold, Caller=A, Called=B, Connected=B, Reason =direct, gcid=GC1	Call State = connected Caller=A, Called=B, Connected=A, Reason =direct, gcid = GC1	Call State = connected Caller=C, Called=D, Connected=D, Reason =direct, gcid = GC2	Call State = connected Caller=C, Called=D, Connected=C, Reason =direct, gcid = GC2
A initiates Refer to D on (d1) with Replaces (d2)	From CTI (callState = IDLE with reason = REFER) TAPI call state IDLE with reason = DIRECT with extended reason = REFER	CPIC Changed from CTI Caller=B, Called=C, Origcalled = D, LRP=C, Reason=REPLACEs TAPI callinfo Caller=B, Called=B, Connected = D, Redirecting=C, Redirection=D, Reason=DIRECT with extended REPLACEs, CallId=GC1	From CTI (callState = IDLE with reason = REPLACEs.) TAPI call state IDLE with reason = DIRECT with extended reason = REPLACEs	GCID changed from CTI to GC1 CPIC Changed from CTI with Caller=B (referee), Called=D, Origcalled = D, LRP=C, Reason=REPLACEs TAPI callinfo caller=B, called=D, connected=B, redirecting=C, redirection=D, reason=DIRECT with extended REPLACEs, callId=GC1

3XX

Application monitors B.

Table A-19 3XX

Actions	CallState/CallInfo @Referrer (A)	CallState/CallInfo @Referree (B)	CallState/CallInfo @Refer-to-Target (C)
A calls external phone that is running SIP, which has CFDUNC set to B		TSPI: LINE_APPNEWCALL Reason = LINECALL REASON_REDIRECT	

SRTP

Media Terminate by Application (Open Secure CTI Port or RP)

- Negotiate version
- Sends LineOpen with extension version as 0x8007000
- Send CciscoLineDevSpecificUserSetSRTPAlgorithmID
- Send CCiscoLineDevSpecificUserControlRTPStream
- Now, the CTI port or RP gets registered as secure port
- Make call from secure IP phone to the CTI port or RP port
- Answer the call from application
- SRTP indication gets reported as LineDevSpecific event
- SRTP key information get stored in LINECALLINFO::devSpecifc for retrieval

Media Terminate by TSP Wave Driver (open secure CTI port)

- Negotiate version
- Sends LineOpen with extension version as 0x4007000
- Send CciscoLineDevSpecificUserSetSRTPAlgorithmID
- Send CciscoLineDevSpecificSendLineOpen
- Now, the CTI port gets registered as secure port
- Make call from secure IP phone to the CTI port
- Answer the call from application
- SRTP indication gets reported as LineDevSpecific event
- SRTP key information get stored in LINECALLINFO::devSpecifc for retrieval

Intercom

This configuration gets used for all the following use cases:

1. IPPhone A has two lines, line1 (1000) and line2 (5000). Line2 represents an intercom line. Speeddial to 5001 with label ìAssistant_1î gets configured.
2. IPPhone B has three lines, line1 (1001), line2 (5001), and Line3 (5002). Line2 and Line3 represent intercom lines. Speeddial to 5000 with label ìManager_1î gets configured on line2. Line 3 does not have Speeddial configured for it.
3. IPPhone C has two lines, line1 (1002) and line2 (5003). 5003 represents an intercom line that is configured with Speeddial to 5002 with label ìAssistant_5002î.
4. IPPhone D has one line (5004). 5004 represnts an intercom line.
5. CTIPort X has two lines, line1 (2000) and line2 (5555). Line2 represents an intercom line. Speedial to 5001 gets configured with label ìAssistant_1î.
6. Intercom lines (5000 to 5003) exists in same partition = Intercom_Group_1 and they remain reachable from each other. 5004 exists in Intercom_Group_2.

7. Application monitoring all lines on all devices.

Assumption: Application initialized and CTI provided the details on speeddial and lines with intercom line on all the devices. Behavior should act the same for phones that are running SCCP, and those that are running SIP.

Application Invoking Speeddial

Action	Events
LineOpen on 5000 & 5001 Initiate InterCom Call on 5000	For 5000 receive LINE_CALLSTATE cbInst=x0 param1=x03000000 param2=x1, ACTIVE param3=x0, Receive StartTransmission event For 5001 receive LINE_CALLSTATE cbInst=x0 param1= x03000000 param2=x1, ACTIVE param3=x0, Receive StartReception event Receive zipzip tone with reason as intercom

Agent Invokes Talkback

Table 1:

Action	Events
Continuing from the previous use case, 5001 initiates LineTalkBack from application on the InterCom call	For 5000 receive LINE_CALLSTATE device=x10218 param1=x100, CONNECTED param2=x1, ACTIVE param3=x0, Receive StartReception event For 5001 receive LINE_CALLSTATE device=x101f6 cbInst=x0 param1=x100, CONNECTED param2=x1, ACTIVE param3=x0, Receive StartTransmission event

Change the SpeedDial

Action	Events
Open line 5000 LineChangeSpeeddial request (speeddial to 5003, label = “Assistant_5003”)	The new speed dial and label is successfully set for the intercom line Receive LineSpeeddialChangeEvent from CTI Send LINE_DEVSPECIFIC to indicate that speeddial and label changed
Application issues LIneGetDevCaps to retrieve speeddial/label that is set on the line	TAPI returns configured speeddial/label that is configured on the line.

Secure Conferencing

Conference with All Parties as Secure

The conference bridge includes security profile. MOH is not configured. A, B, and C get registered as Encrypted.

Action	CTI Messages	TAPI Messages	TAPI Structures
A calls B; B answers the call	Party A		
	CallStateChangedEvent, CH=C1, GCH=G1, Calling=A, Called=B, OrigCalled=B, LR=NP, State=Connected, Origin=OutBound, Reason=Direct SecurityStaus= NotAuthenticated CtiCallSecurityStatusUpdate LH = A, CH = C1 SecurityStaus= Encrypted	LINE_CALLDEVSPECIFIC hDevice=A dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP Devspecific Data : CallSecurityInfo = Encrypted
	Party B		
	CallStateChangedEvent, CH=C2, GCH=G1, Calling=A, Called=B, OrigCalled=B, LR=NP, State=Connected, Origin=OutBound, Reason=Direct SecurityStaus=NotAuthenticated CtiCallSecurityStatusUpdate LH = B, CH = C2 SecurityStaus= Encrypted	LINE_CALLDEVSPECIFIC hDevice=B dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=B dwCallID=T1 dwOrigin=INTERNAL dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=A dwRedirectionID=NP dwRedirectingID=NP Devspecific Data : CallSecurityInfo = Encrypted
B does lineSetUpConference	Party B		

Action	CTI Messages	TAPI Messages	TAPI Structures
	CtiCallSecurityStatusUpdate LH = B, CH = C2 SecurityStaus= NotAuthenticated	LINE_CALLDEVSPECIFIC hDevice=B dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=B dwCallID=T1 dwOrigin=INTERNAL dwReason=DIRECT dwCallerID=A dwCalledID=B dwConnectedID=A dwRedirectionID=NP dwRedirectingID=NP Devspecific Data : CallSecurityInfo = NotAuthenticated
B calls C; C answers the call	Party B		
	CallStateChangedEvent, CH=C3, GCH=G2, Calling=A, Called=B, OrigCalled=B, LR=NP, State=Connected, Origin=OutBound, Reason=Direct SecurityStaus=NotAuthenticated CtiCallSecurityStatusUpdate LH = B, CH = C3 SecurityStaus= Encrypted	LINE_CALLDEVSPECIFIC hDevice=B dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=B dwCallID=T2 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=B dwCalledID=C dwConnectedID=C dwRedirectionID=NP dwRedirectingID=NP Devspecific Data : CallSecurityInfo = Encrypted
	Party C		
	CallStateChangedEvent, CH=C4, GCH=G2, Calling=B, Called=C, OrigCalled=C, LR=NP, State=Connected, Origin=OutBound, Reason=Direct SecurityStaus= NotAuthenticated CtiCallSecurityStatusUpdate LH = C, CH = C4 SecurityStaus= Encrypted	LINE_CALLDEVSPECIFIC hDevice=C dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=C dwCallID=T2 dwOrigin=INTERNAL dwReason=DIRECT dwCallerID=B dwCalledID=C dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP Devspecific Data : CallSecurityInfo = Encrypted

Action	CTI Messages	TAPI Messages	TAPI Structures
B completes conf	Party B		
	CtiCallSecurityStatusUpdate LH = B, CH = C2 SecurityStaus= Encrypted	LINE_CALLDEVSPECIFIC hDevice=B dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=B dwCallID=T1 dwOrigin=CONFERENCE dwReason=UNKNOWN dwCallerID=NP dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectingID=NP Devspecific Data : CallSecurityInfo = Encrypted

Hold or Resume in Secure Conference

Conference bridge includes security profile. MOH gets configured. A, B, and C represent secure phones and exist in conference with overall call security status as secure.

Action	CTI Messages	TAPI Messages	TAPI Structures
A does lineHold	Party A		
	CtiCallSecurityStatusUpdate, LH = A, CH = C1, SecurityStaus= NotAuthenticated	LINE_CALLDEVSPECIFIC hDevice=A dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=CONFERENCE dwReason=UNKNOWN dwCallerID=NP dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP Devspecific Data : CallSecurityInfo = NotAuthenticated
	Party B		

Action	CTI Messages	TAPI Messages	TAPI Structures
	CtiCallSecurityStatusUpdate, LH = B, CH = C2, SecurityStaus= NotAuthenticated	LINE_CALLDEVSPECIFIC hDevice=B dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=B dwCallID=T1 dwOrigin=CONFERENCE dwReason=UNKNOWN dwCallerID=NP dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP Devspecific Data : CallSecurityInfo = CtiCallSecurityStatusUpdate, LH = A, CH = C1, SecurityStaus= NotAuthenticated
	Party C		
	CtiCallSecurityStatusUpdate, LH = A, CH = C1, SecurityStaus= NotAuthenticated	LINE_CALLDEVSPECIFIC hDevice=C dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine= dwCallID=T1 dwOrigin=CONFERENCE dwReason=UNKNOWN dwCallerID=NP dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP Devspecific Data : CallSecurityInfo = NotAuthenticated
A does lineResume	Party A		
	CtiCallSecurityStatusUpdate, LH = A, CH = C1, SecurityStaus= Encrypted	LINE_CALLDEVSPECIFIC hDevice=A dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=CONFERENCE dwReason=UNKNOWN dwCallerID=NP dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP Devspecific Data : CallSecurityInfo = Encrypted

Action	CTI Messages	TAPI Messages	TAPI Structures
	Party B		
	CtiCallSecurityStatusUpdate, LH = B, CH = C2, SecurityStaus= Encrypted	LINE_CALLDEVSPECIFIC hDevice=B dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDAT A dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine=B dwCallID=T1 dwOrigin=CONFERENCE dwReason=UNKNOWN dwCallerID=NP dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP Devspecific Data : CallSecurityInfo = Encrypted
	Party C		
	CtiCallSecurityStatusUpdate, LH = C, CH = C4, SecurityStaus= Encrypted	LINE_CALLDEVSPECIFIC hDevice=C dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDAT A dwParam2=SLDST_CALL_SECURITY_STATUS dwParam3=0	LINECALLINFO (hCall-1) hLine= dwCallID=T1 dwOrigin=CONFERENCE dwReason=UNKNOWN dwCallerID=NP dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectionID=NP Devspecific Data : CallSecurityInfo = Encrypted

Monitoring and Recording

Monitoring a Call

A (agent) and B (customer) get connected. BIB on A gets set to on.

Action	CTI Messages	TAPI Messages	TAPI Structures
	Party C		
C(supervisor) issues start monitoring req with A's permanentLineID as input	NewCallEvent, CH=C3, GCH=G2, Calling=C, Called=NP, OrigCalled=NP, LR=NP, State=Dialtone, Origin=OutBound, Reason=Direct	LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=ORIGIN dwParam2=0 dwParam3=0 LINE_CALLINFO hDevice=hCall-1 dwCallbackInstance=0 dwParam1=REASON, CALLERID dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=C dwCallID=T2 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=C dwCalledID=NP dwConnectedID=NP dwRedirectionID=NP dwRedirectingID=NP
A's BIB automatically answers	Party C		
	CallStateChangedEvent, CH=C3, State=Connected, Cause=CauseNoError, Reason=Direct, Calling=C, Called=A, OrigCalled=A, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=ACTIVE dwParam3=0	LINECALLINFO (hCall-1) hLine=C dwCallID=T2 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=C dwCalledID=A dwConnectedID=A dwRedirectionID=NP dwRedirectingID=NP
	Party A		
	MonitoringStartedEvent, CH = C1	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_MONITOR_STARTED dwParam2=0 dwParam3=0	LINECALLINFO (hCall-2) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=B dwCalledID=A dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP
	Party C		

Action	CTI Messages	TAPI Messages	TAPI Structures
	LineCallAttributeInfoEvent, CH=C3, Type = 2 (MonitorCall_Target), CI = C1, Address=A's DN, Partition=A's Partition, DeviceName = A's Name	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_ATTRIBUTE_INFO dwParam3=0	LINECALLINFO (hCall-1) hLine=C dwCallID=T2 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=C dwCalledID=A dwConnectedID=A dwRedirectionID=NP dwRedirectingID=NP DevSpecifc Data: Type: CallAttribute_SilentMonitorCall_Target, CI = C1, DN = A's DN, Partition = A's Partition, DeviceName = A's Name
	Party A		
	LineCallAttributeInfoEvent, CH=C1, Type = 1 (MonitorCall), CI = C3 Address=C's DN, Partition=C's Partition, DeviceName = C's Name	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_ATTRIBUTE_INFO dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=INTERNAL dwReason=DIRECT dwCallerID=B dwCalledID=A dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP DevSpecifc Data: Type:CallAttribute_SilentMonitorCall, CI = C3 DN = C's DN, Partition = C's Partition, DeviceName = C's Name
C drops the call	Party C		
	CallStateChangedEvent, CH=C3, State=Idle, Cause=CauseNoError, Reason=Direct, Calling=C, Called=A, OrigCalled=A, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=IDLE dwParam2=0 dwParam3=0	

Action	CTI Messages	TAPI Messages	TAPI Structures
	Party A		
	MonitoringEndedEvent, CH = C1	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_MONITOR_ENDED dwParam2= DisconnectMode_Normal dwParam3=0	

Automatic Recording

Recording type on A (agent Phone) is configured as Automatic. D is configured as a Recorder Device.

Action	CTI Messages	TAPI Messages	TAPI Structures
A recieves a call from B, and A answers the call Recording session gets established between the agent phone and the recorder	Party A		
	CallStateChangedEvent, CH=C1, State=Connected, Cause=CauseNoError, Reason=Direct, Calling=B, Called=A, OrigCalled=A, LR=NP	LINE_CALLSTATE hDevice=hCall-1 dwCallbackInstance=0 dwParam1=CONNECTED dwParam2=ACTIVE dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=INTERNAL dwReason=DIRECT dwCallerID=B dwCalledID=A dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP

Action	CTI Messages	TAPI Messages	TAPI Structures
	RecordingStartedEvent, CH = C1	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_RECORDING_STARTED dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=B dwCalledID=A dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP
	LineCallAttributeInfoEvent CH = C1, Type = 3 (Automatic Recording), Address = D's DN, Partition = D's Partition, DeviceName = D's Name	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_ATTRIBUTE_INFO dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=B dwCalledID=A dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP DevSpecifc Data: Type: App Controlled Recording, DN = D's DN, Partition = D's Partition, DeviceName = D's Name

Application-Controlled Recording

A (C1) and B (C2) connect. Recording Type on A gets configured as 'Application Based'. D gets configured as a Recorder Device.

Action	CTI Messages	TAPI Messages	TAPI Structures
A issues start recording request Recording session gets established between the agent phone and the recorder	Party A		
	RecordingStartedEvent, CH = C1	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_RECORDING_STARTED dwParam2=0 dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=B dwCalledID=A dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP

Action	CTI Messages	TAPI Messages	TAPI Structures
	LineCallAttributeInfoEvent CH = C1, Type = 4 (App Controlled Recording), Address = D's DN, Partition = D's Partition, DeviceName = D's Name	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_CALL_ATTRIBUTE_INFO dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=B dwCalledID=A dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP DevSpecifc Data: Type: App Controlled Recording, DN = D's DN, Partition = D's Partition, DeviceName = D's Name
A issues stop monitoring request	RecordingEndedEvent, CH = C1	LINE_CALLDEVSPECIFIC hDevice=hCall-1 dwCallbackInstance=0 dwParam1 = SLDSMT_RECORDING_ENDED dwParam2= DisconnectMode_Normal dwParam3=0	LINECALLINFO (hCall-1) hLine=A dwCallID=T1 dwOrigin=OUTBOUND dwReason=DIRECT dwCallerID=B dwCalledID=A dwConnectedID=B dwRedirectionID=NP dwRedirectingID=NP

Conference Enhancements

Noncontroller Adding Parties to Conferences

A,B, and C exist in a conference that A created.

Action	Events
A,B, and C exist in a conference	At A: Conference – Caller=”A”, Called=”B”, Connected=”B” Connected Conference – Caller=”A”, Called=”C”, Connected=”C” At B: Conference – Caller=”A”, Called=”B”, Connected=”A” Connected Conference – Caller=”B”, Called=”C”, Connected=”C” At C: Conference – Caller=”B”, Called=”C”, Connected=”B” Connected Conference – Caller=”C”, Called=”A”, Connected=”A”

Action	Events
C issues a linePrepareAddToConference to D	At A: Conference – Caller=”A”, Called=”B”, Connecgted=”B” Connected Conference – Caller=”A”, Called=”C”, Connecgted=”C” At B: Conference – Caller=”A”, Called=”B”, Connecgted=”A” Connected Conference – Caller=”B”, Called=”C”, Connecgted=”C” At C: Conference – Caller=”B”, Called=”C”, Connecgted=”B” OnHoldPendConf Conference – Caller=”C”, Called=”A”, Connecgted=”A” Connected - Caller=”C”, Called=”D”, Connecgted=”D” At D: Connected - Caller=”C”, Called=”D”, Connecgted=”C”
C issues a lineAddToConference to D	At A: Conference – Caller=”A”, Called=”B”, Connecgted=”B” Connected Conference – Caller=”A”, Called=”C”, Connecgted=”C” Conference – Caller=”A”, Called=”D”, Connecgted=”D” At B: Conference – Caller=”A”, Called=”B”, Connecgted=”A” Connected Conference – Caller=”B”, Called=”C”, Connecgted=”C” Conference – Caller=”B”, Called=”D”, Connecgted=”D” At C: Conference – Caller=”B”, Called=”C”, Connecgted=”B” Connected Conference – Caller=”C”, Called=”A”, Connecgted=”A” Conference – Caller=”C”, Called=”D”, Connecgted=”D” At D: Conference – Caller=”C”, Called=”D”, Connecgted=”C” Connected Conference – Caller=”D”, Called=”A”, Connecgted=”A” Conference – Caller=”D”, Called=”B”, Connecgted=”B”

Chaining Two Ad Hoc Conferences by Using Join

Table 2:

Actions	TSP CallInfo
A calls B, B answers, then B initiates conference to C, C answers, and B completes the conference	At A: GCID-1 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = A Called = B CONFERENCED : Caller = A Called = C At B: GCID-1 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = A Called = B CONFERENCED : Caller = B Called = C At C: GCID-1 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = B Called = C CONFERENCED : Caller = C Called = A

Table 2:

Actions	TSP CallInfo
C initiates or completes conference to D and E	No Change for A and B At C: - First conference GCID-1 ONHOLD : Caller = Unknown Caller = Unknown CONFERENCED : Caller = A Called = B CONFERENCED : Caller = A Called = C - Second conference GCID-2 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = C Called = D CONFERENCED : Caller = C Called = E At D: GCID-2 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = C Called = D CONFERENCED : Caller = D Called = E At E: GCID-2 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = C Called = E CONFERENCED : Caller = E Called = D

Table 2:

Actions	TSP CallInfo
C initiates JOIN request to join to conference call together, with GCID as the primary call	At A: GCID-1 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = A Called = B CONFERENCED : Caller = A Called = C CONFERENCED : Caller = A Called = Conference-2 At B : GCID-1 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = A Called = B CONFERENCED : Caller = B Called = C CONFERENCED : Caller = B Called = Conference-2 At C: - First conference GCID-1 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = B Called = C CONFERENCED : Caller = C Called = A CONFERENCED : Caller = C Called = Conference-2

Table 2:

Actions	TSP CallInfo
	At D: GCID-2 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = D Called = E CONFERENCED : Caller = D Called = Conference-1 At E : GCID-2 CONNECTED : Caller = Unknown Caller = Unknown CONFERENCED : Caller = E Called = D CONFERENCED : Caller = E Called = Conference-1

Calling Party IP Address

Basic Call

TAPI application monitors party B

Party A represents an IP phone

A calls B

IP Address of A is available to TAPI application that is monitoring party B

Consultation Transfer

TAPI application monitors party C

Party B represents an IP phone

A talks to B

B intiates a consultation transfer call to C

IP Address of B is available to TAPI application that is monitoring party C.

B Completes the transfer

Calling IP address of A is not available to TAPI application that is monitoring party C (not a supported scenario).

Consultation Conference

TAPI application monitors party C

Party B represents an IP phone

A talks to B

B initiates a consultation conference call to C

IP Address of B is available to TAPI application that is monitoring party C.

B Completes the conference

Calling IP address of A and B is not available to TAPI application that is monitoring party C (not a supported scenario)

Redirect

TAPI application monitors party B and party C

Party A represents an IP phone

A calls B

IP Address of A is available to TAPI application that is monitoring party B

Party A redirects B to party C

Calling IP address is not available to TAPI application that is monitoring party B (not a supported scenario)

Calling IP address B is available to TAPI application that is monitoring party C

Click to Conference

Third-party conference gets created by using click-2-conference feature:

Action	**Events**
Use Click-to-Call to create call from A to B, and B answers	For A: CONNECTED reason = DIRECT Calling = A, Called = B, Connected = B For B: CONNECTED reason = DIRECT Calling = A, Called = B, Connected = A

Action	Events
Use Click-2-Conference feature to add C into conference, and C answers	For A: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = B CONFERENCED Calling = A, Called = C, Connected = C For B: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = A CONFERENCED Calling =B, Called = C, Connected = C For C CONNECTED Reason = UNKNOWN ExtendedCallReason = ClickToConference CONFERENCED Calling = C, Called = A, Connected = A CONFERENCED Calling = C, Called = B, Connected = B

Creating Four-Party Conference by Using Click-2-Conference Feature

Action	Events
Use Click-to-Call to create call from A to B	For A: CONNECTED reason = DIRECT Calling = A, Called = B, Connected = B For B: CONNECTED reason = DIRECT Calling = A, Called = B, Connected = A

Action	Events
Use Click-2-Conference feature to add C into conference	For A: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = B CONFERENCED Calling = A, Called = C, Connected = C For B: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = A CONFERENCED Calling = C, Called = C, Connected = C For C CONNECTED Reason = DIRECT ExtendedCallReason = ClickToConference CONFERENCED Calling = C, Called = A, Connected = A CONFERENCED Calling = C, Called = B, Connected = B

Action	Events
Use Click-2-Conference feature to add party D	For A: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = B CONFERENCED Calling = A, Called = C, Connected = C CONFERENCED Calling = A, Called = D, Connected = D For B: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = A CONFERENCED Calling = B, Called = C, Connected = C CONFERENCED Calling = B, Called = D, Connected = D For C CONNECTED Reason = UNKNOWN ExtendedCallReason = ClickToConference CONFERENCED Calling = C, Called = A, Connected = A CONFERENCED Calling = C, Called = B, Connected = B CONFERENCED Calling = C, Called = D, Connected = D For D CONNECTED Reason = UNKNOWN ExtendedCallReason = ClickToConference

Action	Events
	CONFERENCED Calling = D, Called = A, Connected = A CONFERENCED Calling = D, Called = B, Connected = B CONFERENCED Calling = D, Called = C, Connected = C

Drop Party by Using Click-2-Conference

Action	Events
Conference gets created by using Click-2-Conference feature to add C into conference	For A: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = B CONFERENCED Calling = A, Called = C, Connected = C For B: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = A CONFERENCED Calling = B, Called = C, Connected = C For C CONNECTED Reason = UNKNOWN ExtendedCallReason = ClickToConference CONFERENCED Calling = C, Called = A, Connected = A CONFERENCED Calling = C, Called = B, Connected = B

Drop C from Click-2-Conference feature	For A CONNECTED Reason = DIRECT ExtendedCallReason = DIRECT Calling = A, Called = B, Connected = B For B CONNECTED Reason = DIRECT ExtendedCallReason = DIRECT Calling = A, Called = B, Connected = A For C IDLE

Drop Entire Conference by Using Click-2-Conference Feature

Action	Events

Conference gets created by using Click-2-Conference feature to add C into conference	For A: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = B CONFERENCED Calling = A, Called = C, Connected = C For B: CONNECTED reason = DIRECT ExtendedCallReason = DIRECT CONFERENCED Calling = A, Called = B, Connected = A CONFERENCED Calling = B, Called = C, Connected = C For C CONNECTED Reason = UNKOWN ExtendedCallReason = ClickToConference CONFERENCED Calling = C, Called = A, Connected = A CONFERENCED Calling = C, Called = B, Connected = B
Drop entire conference	For A IDLE For B IDLE For C IDLE

Calling Party Normalization

Incoming Call from PSTN to End Point

Action	CTI Messages	TAPI Messages	TAPI Structures
A Call gets offered from a PSTN number 5551212/<SUBSCRIBER> through a San Jose gateway to a CCM end point 2000	CallStateChangedEvent, UnModified Calling Party=5551212, UnModified Called Party=2000, UnModified Original Called Party=2000, Modified Calling Party=5551212, Modified Called Party=2000, Modified Original Called Party=2000, Globalized Calling party = +14085551212, Calling Party Number Type=SUBSCRIBER, Called Party Number Type=UNKNOWN, Original Called Party Number Type,=UNKNOWN State=Connected, Origin=OutBound, Reason = Direct	LINE_CALLSTATE = CONNECTED	LINECALLINFO Displayed Calling Party=5551212, Displayed Called Party=2000, Displayed Redirection Party=, Displayed Redirected Party=, Globalized Calling Party = +14085551212, Calling Party Number Type=SUBSCRIBER, Called Party Number Type= UNKNOWN, Redirection Party Number Type=, Redirecting Party Number Type=

Incoming Call from National PSTN to CTI-Observed End Point

Action	CTI Messages	TAPI Messages	TAPI Structures
A Call gets offered from a Dallas PSTN number 5551212/<NATIONAL> through a San Jose gateway to a CCM end point 2000	CallStateChangedEvent, UnModified Calling Party=9725551212, UnModified Called Party=2000, UnModified Original Called Party=2000, Modified Calling Party=9725551212, Modified Called Party=2000, Modified Original Called Party=2000, Globalized Calling party = +19725551212, Calling Party Number Type=NATIONAL, Called Party Number Type=UNKNOWN, Original Called Party Number Type,=UNKNOWN State=Connected, Origin=OutBound, Reason = Direct	LINE_CALLSTATE = CONNECTED	LINECALLINFO Displayed Calling Party=9725551212, Displayed Called Party=2000, Displayed Redirection Party=, Displayed Redirected Party=, Globalized Calling Party = +19725551212, Calling Party Number Type=NATIONAL, Called Party Number Type= UNKNOWN, Redirection Party Number Type=, Redirecting Party Number Type=

Incoming Call from International PSTN to CTI-Observed End Point

Action	CTI Messages	TAPI Messages	TAPI Structures

A Call gets offered from a PSTN number in India 22221111/<INTERNATIONAL> through a San Jose gateway to a CCM end point 2000	CallStateChangedEvent, UnModified Calling Party=011914422221111, UnModified Called Party=2000, UnModified Original Called Party=2000, Modified Calling Party=011914422221111, Modified Called Party=2000, Modified Original Called Party=2000, Globalized Calling party = +914422221111, Calling Party Number Type=INTERNATIONAL, Called Party Number Type=UNKNOWN, Original Called Party Number Type,=UNKNOWN State=Connected, Origin=OutBound, Reason = Direct	LINE_CALLSTATE = CONNECTED	LINECALLINFO Displayed Calling Party=011914422221111, Displayed Called Party=2000, Displayed Redirection Party=, Displayed Redirected Party=, Globalized Calling Party = +914422221111, Calling Party Number Type=INTERNATIONAL, Called Party Number Type = UNKNOWN, Redirection Party Number Type=, Redirecting Party Number Type=

Outgoing Call from CTI-Observed End Point to PSTN Number

Action	CTI Messages	TAPI Messages	TAPI Structures
A Call gets initiated from a CCM end point 2000 through a San Jose gateway to a PSTN number 5551212/<NATIONAL>	CallStateChangedEvent, UnModified Calling Party=2000, UnModified Called Party=5551212, UnModified Original Called Party=5551212, Modified Calling Party=2000, Modified Called Party=5551212, Modified Original Called Party=5551212, Globalized Calling party = +14085551212, Calling Party Number Type=UNKNOWN, Called Party Number Type=SUBSCRIBER, Original Called Party Number Type,=SUBSCRIBER State=Connected, Origin=OutBound, Reason = Direct	LINE_CALLSTATE = CONNECTED	LINECALLINFO Displayed Calling Party=2000, Displayed Called Party=5551212, Displayed Redirection Party=, Displayed Redirected Party=, Globalized Calling Party = +14085551212, Calling Party Number Type=UNKNOWN, Called Party Number Type= SUBSCRIBER, Redirection Party Number Type=, Redirecting Party Number Type=

Outgoing Call from CTI-Observed End Point to National PSTN Number

Action	CTI Messages	TAPI Messages	TAPI Structures
A Call gets initiated from a CCM end point 2000 through a San Jose gateway to a Dallas PSTN number 9725551212/<NATIONAL >	CallStateChangedEvent, UnModified Calling Party=2000, UnModified Called Party=9725551212, UnModified Original Called Party=9725551212, Modified Calling Party=2000, Modified Called Party=9725551212, Modified Original Called Party=9725551212, Globalized Calling party = +19725551212, Calling Party Number Type=UNKNOWN, Called Party Number Type=NATIONAL, Original Called Party Number Type,=NATIONAL State=Connected, Origin=OutBound, Reason = Direct	LINE_CALLSTATE = CONNECTED	LINECALLINFO Displayed Calling Party=2000, Displayed Called Party=9725551212, Displayed Redirection Party=, Displayed Redirected Party=, Globalized Calling Party = +19725551212, Calling Party Number Type=UNKNOWN, Called Party Number Type= NATIONAL, Redirection Party Number Type=, Redirecting Party Number Type=

Outgoing Call from CTI-Observed End Point to International PSTN Number

Action	CTI Messages	TAPI Messages	TAPI Structures
A Call gets initiated from a CCM end point 2000 through a San Jose gateway to a PSTN number in India 914422221111/<INTERNATIONAL>	CallStateChangedEvent, UnModified Calling Party=2000, UnModified Called Party=011914422221111, UnModified Original Called Party=011914422221111, Modified Calling Party=2000, Modified Called Party=011914422221111, Modified Original Called Party=011914422221111, Globalized Calling party = +914422221111, Calling Party Number Type=UNKNOWN, Called Party Number Type=INTERNATIONAL, Original Called Party Number Type,=INTERNATIONAL State=Connected, Origin=OutBound, Reason = Direct	LINE_CALLSTATE = CONNECTED	LINECALLINFO Displayed Calling Party=2000, Displayed Called Party=011914422221111, Displayed Redirection Party=, Displayed Redirected Party=, Globalized Calling Party = +914422221111, Calling Party Number Type=UNKNOWN, Called Party Number Type = INTERNATIONAL, Redirection Party Number Type=, Redirecting Party Number Type=

Do Not Disturb–Reject

Application Enables DND-R on a Phone

Action	TAPI Messages	TAPI Structures
Phone A enables DND-Reject in the admin pages	LINE_CALLDEVSPECIFIC hDevice=C dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_DND_OPTION_STATUS dwParam3=2	

Normal Feature Priority

Action	TAPI Messages	TAPI Structures
With Phone B DND-R enabled, Phone A calls Phone B with feature priority as Normal	Party A	
	LINE_CALLSTATE = IDLE	
	Party B	
	No TAPI messages	

Feature Priority - Emergency

Action	TAPI Messages	TAPI Structures
With Phone B DND-R enabled, Phone A calls Phone B with feature priority as Emergency	Party A	
	LINE_CALLSTATE = CONNECTED dwParam1 = 0x00000100 dwParam2 = 0x00000001	LINECALLINFO (hCall-1) hLine=C dwCallID=T2 dwOrigin=INTERNAL dwCallerID=A dwCalledID=B dwRedirectionID=NP dwRedirectingID=NP
	Party B	
	LINE_CALLSTATE = CONNECTED dwParam1 = 0x00000100 dwParam2 = 0x00000001	LINECALLINFO (hCall-1) hLine=C dwCallID=T2 dwOrigin=INTERNAL dwCallerID=A dwCalledID=B dwRedirectionID=NP dwRedirectingID=NP

Shared Line Scenario for DND-R

Action	TAPI Messages	TAPI Structures
Phones B and B' represents shared lines. Phone B' is DND-R enabled but not B. Phone A calls Phone B with feature priority normal	Party A	
	LINE_CALLSTATE = CONNECTED dwParam1 = 0x00000100 dwParam2 = 0x00000001	LINECALLINFO (hCall-1) hLine=C dwCallID=T2 dwOrigin=INTERNAL dwCallerID=A dwCalledID=B dwRedirectionID=NP dwRedirectingID=NP
	Party B	
	LINE_CALLSTATE = CONNECTED dwParam1 = 0x00000100 dwParam2 = 0x00000001	LINECALLINFO (hCall-1) hLine=C dwCallID=T2 dwOrigin=INTERNAL dwCallerID=A dwCalledID=B dwRedirectionID=NP dwRedirectingID=NP
	Party B'	
	LINE_CALLSTATE = CONNECTED dwParam1 = 0x00000100 dwParam2 = 0x00000002	

Application Disables DND-R or Changes the Option for DND

Action	TAPI Messages	TAPI Structures
Phone A changes from DND-Reject to DND-RingerOff.	LINE_CALLDEVSPECIFIC hDevice=C dwCallbackInstance=0 dwParam1= SLDSMT_LINECALLINFO_DEVSPECIFICDATA dwParam2=SLDST_DND_OPTION_STATUS dwParam3=1	

Join Across Lines

Setup

Line A on device A

Line B1 and B2 on device B

Line C on device C

Line D on device D

Line B1’ on device B1’, B1’ is a shared line with B1

Join Two Calls from Different Lines to B1

Action	Expected Events
A → B1 is HOLD C → B2 is connected	For A LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected B1 For B1: LINE_CALLSTATE param1=x100, HOLD Caller = A, Called = B1, Connected = A For B2: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2 , Connected = C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2, Connected = B2 For B1’: LINE_CALLSTATE param1=x100, CONNECTED, INACTIVE Caller = A, Called = B1, Connected = A
Application issues lineDevSpecific(SLDST_JOIN) with the call on B1 as survival call	For A CONNECTED CONFERENCED Caller=A, Called=B1, Connected=B1 CONFERENCED Caller=A Called=C, Connected=C For B1 CONNECTED CONFERENCED Caller=A, Called=B1, Connected=A CONFERENCED Caller=B1 Called=C, Connected=C For B2 Call will go IDLE

Action	Expected Events
	For C CONNECTED CONFERENCED Caller=C, Called=B2, Connected=B1 (or A) CONFERENCED Caller=C Called=A, Connected=A (or B1) For B1' CONNECTED INACTIVE CONFERENCED Caller=A, Called=B1, Connected=A CONFERENCED Caller=B1 Called=C, Connected=C

Join Three Calls from Different Lines to B1

Action	Expected Events
A → B1 is hold,	
C → B2 is hold	
D → B2 is connected	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected B1 For B1: LINE_CALLSTATE param1=x100, HOLD Caller = A, Called = B1, Connected = A For B2: LINE_CALLSTATE for call-1 param1=x100, HOLD Caller = C, Called = B2 , Connected = C LINE_CALLSTATE for call-2 param1=x100, CONNECTED Caller = D, Called = B2 , Connected = D For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2, Connected = B2 For D: LINE_CALLSTATE param1=x100, CONNECTED Caller = D, Called = B2, Connected = B2

Action	Expected Events
	For B1’: LINE_CALLSTATE param1=x100, HOLD Caller = A, Called = B1, Connected = A
Application issues lineDevSpecific(SLDST_JOIN) with the call on B1 as survival call	For A CONNECTED CONFERENCED Caller=A, Called=B1, Connected=B1 CONFERENCED Caller=A Called=C, Connected=C CONFERENCED Caller=A Called=D, Connected=D For B1 CONNECTED CONFERENCED Caller=A, Called=B1, Connected=A CONFERENCED Caller=B1 Called=C, Connected=C CONFERENCED Caller=B1 Called=D, Connected=D For B2 Call-1 and call-2 will go IDLE For C CONNECTED CONFERENCED Caller=B1, Called=C, Connected=B1 CONFERENCED Caller=C Called=A, Connected=A CONFERENCED Caller=C Called=D, Connected=D For D CONNECTED CONFERENCED Caller=B1, Called=C, Connected=B1 CONFERENCED Caller=D Called=A, Connected=A CONFERENCED Caller=D Called=C, Connected=C For B1’

Action	Expected Events
	CONNECTED INACTIVE CONFERENCED Caller=A, Called=B1, Connected=A CONFERENCED Caller=B1 Called=C, Connected=C CONFERENCED Caller=B1 Called=D, Connected=D

Join Calls from Different Lines to B1 with Conference

Action	Expected Events
A,B1,C in conference where B1 is controller D→ B2 Connected	For A: CONNECTED CONFERENCED Caller=A, Called=B1, Connected=A CONFERENCED Caller=A Called=C, Connected=C For B1: CONNECTED CONFERENCED Caller=A, Called=B1, Connected=A CONFERENCED Caller=B1 Called=C, Connected=C For B2: LINE_CALLSTATE for call-1 param1=x100, CONNECTED Caller = D, Called = B2 , Connected = D For C: CONNECTED CONFERENCED Caller=C, Called=A, Connected=A CONFERENCED Caller=B1 Called=C, Connected=C For D: LINE_CALLSTATE param1=x100, CONNECTED Caller = D, Called = B2, Connected = B2 For B1’: LINE_CALLSTATE CONNECTED INACTIVE

Action	Expected Events
	CONFERENCED Caller=A, Called=B1, Connected=A
	CONFERENCED Caller=B1 Called=C, Connected=C
Application issues lineDevSpecific(SLDST_JOIN) with the call on B1 as survival call	For A
	CONNECTED
	CONFERENCED Caller=A, Called=B1, Connected=B1
	CONFERENCED Caller=A Called=C, Connected=C
	CONFERENCED Caller=A Called=D, Connected=D
	For B1
	CONNECTED
	CONFERENCED Caller=A, Called=B1, Connected=A
	CONFERENCED Caller=B1 Called=C, Connected=C
	CONFERENCED Caller=B1 Called=D, Connected=D
	For B2
	Call will go IDLE
	For C
	CONNECTED
	CONFERENCED Caller=B1, Called=C, Connected=B1
	CONFERENCED Caller=C Called=A, Connected=A
	CONFERENCED Caller=C Called=D, Connected=D
	For D
	CONNECTED
	CONFERENCED Caller=B1, Called=C, Connected=B1
	CONFERENCED Caller=D Called=A, Connected=A
	CONFERENCED Caller=D Called=C, Connected=C
	For B1’
	CONNECTED INACTIVE

Action	Expected Events
	CONFERENCED Caller=A, Called=B1, Connected=A CONFERENCED Caller=B1 Called=C, Connected=C CONFERENCED Caller=B1 Called=D, Connected=D

Join Two Calls from Different Lines to B1 while B1 is not Monitored by TAPI

Action	Expected Events
A → B1 is HOLD, C → B2 is connected	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected B1 For B2: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2 , Connected = C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2, Connected = B2
User issues join request from phone with the call on B1 as survival call	For A CONNECTED CONFERENCED Caller=A, Called=B1, Connected=B1 CONFERENCED Caller=A Called=C, Connected=C For B2 Call will go IDLE For C CONNECTED CONFERENCED Caller=C, Called=B2, Connected=B1 (or A) CONFERENCED Caller=C Called=A, Connected=A (or B1)

Join Two Calls from Different Lines to B2

Action	Expected Events
A → B1 is HOLD, C → B2 is connected	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected B1 For B1: LINE_CALLSTATE param1=x100, HOLD Caller = A, Called = B1, Connected = A For B2: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2 , Connected = C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2, Connected = B2 For B1': LINE_CALLSTATE param1=x100, HOLD Caller = A, Called = B1, Connected = A
Application issues lineDevSpecific(SLDST_JOIN) with the call on B1 as survival call	For A CONNECTED CONFERENCED Caller=A, Called=B1, Connected=B1 CONFERENCED Caller=A Called=C, Connected=C For B1 CONNECTED CONFERENCED Caller=A, Called=B1, Connected=A CONFERENCED Caller=B1 Called=C, Connected=C ?? For B2 Call will go IDLE For C CONNECTED

Action	Expected Events
	CONFERENCED Caller=C, Called=B2, Connected=B1 (or A)
	CONFERENCED Caller=C Called=A, Connected=A (or B1)
	For B1’
	CONNECTED INACTIVE
	CONFERENCED Caller=A, Called=B1, Connected=A
	CONFERENCED Caller=B1 Called=C, Connected=C

Action	Expected Events
A → B1 is HOLD,	For A:
B1 issues setup conference	
C → B2 is connected	
	LINE_CALLSTATE
	param1=x100, CONNECTED Caller = A, Called = B1 Connected B1
	For B1:
	Primary call
	LINE_CALLSTATE
	CONNECTED
	CONFERENCED Caller=A, Called=B1, Connected=B1
	Consult call
	DIALTONE
	For B2:
	LINE_CALLSTATE
	param1=x100, CONNECTED Caller = C, Called = B2 , Connected = C
	For C:
	LINE_CALLSTATE
	param1=x100, CONNECTED Caller = C, Called = B2, Connected = B2
	For B1’:
	LINE_CALLSTATE
	param1=x100, HOLD Caller = A, Called = B1, Connected = A

Action	Expected Events
Application issues lineDevSpecific(SLDST_JOIN) with the call on B2 as survival call	For A: CONNECTED CONFERENCED Caller=A, Called=B1, Connected=B2 CONFERENCED Caller=A Called=C, Connected=C For B1 Both calls will go IDLE For B2 CONNECTED CONFERENCED Caller=B1, Called=A, Connected=A CONFERENCED Caller=C Called=B1, Connected=C For C CONNECTED CONFERENCED Caller=C, Called=B2, Connected=B2 (or A) CONFERENCED Caller=C Called=A, Connected=A (or B2) For B1' Calls go IDLE

B1 Performs a Join Across Line Where B1 is already in a Conference Created by A

Action	Expected Events
A, B1, C are in a conference created by A	For A: Conference – Caller="A", Called="B1", Connected="B1" Connected Conference – Caller="A", Called="C", Connected="C" For B1: Conference – Caller="A", Called="B1", Connected="A" Connected Conference – Caller="B1", Called="C", Connected="C" For C:

Action	Expected Events
	Conference – Caller=”B1”, Called=”C”, Connected=”B1” Connected Conference – Caller=”C”, Called=”A”, Connected=”A” For A: B2 calls D, D answers Conference – Caller=”A”, Called=”B1”, Connected=”B1” Connected Conference – Caller=”A”, Called=”C”, Connected=”C” For B1: Conference – Caller=”A”, Called=”B1”, Connected=”A” OnHold Conference – Caller=”B1”, Called=”C”, Connected=”C” For B2: Connected - Caller=”B2”, Called=”D”, Connected=”D” For C: Conference – Caller=”B1”, Called=”C”, Connected=”B1” Connected Conference – Caller=”C”, Called=”A”, Connected=”A” Connected - Caller=”B2”, Called=”D”, Connected=”B2”
B1 issues a lineDevSpecific(SLDST_JOIN) to join the calls on B1 and B2.	For A: Conference – Caller=”A”, Called=”B1”, Connected=”B1” Connected Conference – Caller=”A”, Called=”C”, Connected=”C” Conference – Caller=”A”, Called=”D”, Connected=”D” For B1: Conference – Caller=”A”, Called=”B1”, Connected=”B1”

Action	Expected Events
	Conference – Caller=”A”, Called=”B1”, Connected=”A” Connected Conference – Caller=”B1”, Called=”C”, Connected=”C” Conference – Caller=”B1”, Called=”D”, Connected=”D” For B2: Call will go IDLE For C: Conference – Caller=”B1”, Called=”C”, Connected=”B1” Connected Conference – Caller=”C”, Called=”A”, Connected=”A” Conference – Caller=”C”, Called=”D”, Connected=”D” For D: Conference – Caller=”B1”, Called=”D”, Connected=”B1” Connected Conference – Caller=”D”, Called=”A”, Connected=”A” Conference – Caller=”D”, Called=”C”, Connected=”C”

B2 Performs a Join Across Line Where B1 is already in a Conference Created by A

Action	Expected Events
A,B1,C are in a conference created by A	For A: Conference – Caller=”A”, Called=”B1”, Connected=”B1” Connected Conference – Caller=”A”, Called=”C”, Connected=”C” For B1: Conference – Caller=”A”, Called=”B1”, Connected=”A” Connected Conference – Caller=”B1”, Called=”C”, Connected=”C” For C:

Action	Expected Events
	Conference – Caller=”B1”, Called=”C”, Connected=”B1” Connected Conference – Caller=”C”, Called=”A”, Connected=”A”
B2 calls D, D answers	For A: Conference – Caller=”A”, Called=”B1”, Connected=”B1” Connected Conference – Caller=”A”, Called=”C”, Connected=”C” For B1: Conference – Caller=”A”, Called=”B1”, Connected=”A” OnHold Conference – Caller=”B1”, Called=”C”, Connected=”C” For B2: Connected - Caller=”B2”, Called=”D”, Connected=”D” For C: Conference – Caller=”B1”, Called=”C”, Connected=”B1” Connected Conference – Caller=”C”, Called=”A”, Connected=”A” For D: Connected - Caller=”B2”, Called=”D”, Connected=”B2”
B2 issues a lineDevSpecific(SLDST_JOIN) to join the calls on B1 and B2.	For A: Conference – Caller=”A”, Called=”B1”, Connected=”B2” Connected Conference – Caller=”A”, Called=”C”, Connected=”C” Conference – Caller=”A”, Called=”D”, Connected=”D” For B1: Conference – Caller=”A”, Called=”B1”, Connected=”A”

Action	Expected Events
	Connected Conference – Caller=”B1”, Called=”C”, Connected=”C” Conference – Caller=”B1”, Called=”D”, Connected=”D” For B2: Call will go IDLE For C: Conference – Caller=”B2”, Called=”C”, Connected=”B2” Connected Conference – Caller=”C”, Called=”A”, Connected=”A” Conference – Caller=”C”, Called=”D”, Connected=”D” For D: Conference – Caller=”B2”, Called=”D”, Connected=”B2” Connected Conference – Caller=”D”, Called=”A”, Connected=”A” Conference – Caller=”D”, Called=”C”, Connected=”C”

B1 Performs a Join Across Line Where B1 is in One Conference and B2 is in a Separate Conference

Action	Expected Events
A,B1,C are in conference1 D, B2, E are in conference2	For A (GCID-1): Conference – Caller=”A”, Called=”B1”, Connected=”B1” Connected Conference – Caller=”A”, Called=”C”, Connected=”C” For B1 (GCID-1): Conference – Caller=”A”, Called=”B1”, Connected=”A” OnHold Conference – Caller=”B1”, Called=”C”, Connected=”C” For C (GCID-1):

Action	Expected Events
	Conference – Caller=”B1”, Called=”C”, Connected=”B1” Connected Conference – Caller=”C”, Called=”A”, Connected=”A” For D (GCID-2): Conference – Caller=”D”, Called=”B2”, Connected=”B2” Connected Conference – Caller=”D”, Called=”E”, Connected=”E” For B2 (GCID-2): Conference – Caller=”D”, Called=”B2”, Connected=”D” Connected Conference – Caller=”B2”, Called=”E”, Connected=”E” For E (GCID-2): Conference – Caller=”B2”, Called=”E”, Connected=”B2” Connected Conference – Caller=”E”, Called=”D”, Connected=”D”
B1 issues a lineDevSpecific(SLDST_JOIN) to join the calls on B1 and B2.	For A: Conference – Caller=”A”, Called=”B1”, Connected=”B1” Connected Conference – Caller=”A”, Called=”C”, Connected=”C” Conference – Caller=”A”, Called=”CFB-2”, Connected=” CFB-2” For B1: Conference – Caller=”A”, Called=”B1”, Connected=”A” Connected Conference – Caller=”B1”, Called=”C”, Connected=”C” Conference – Caller=”B1”, Called=” CFB-2”, Connected=” CFB-2” For B2:

Action	Expected Events
	Call will go IDLE For C: Conference – Caller=”B1”, Called=”C”, Connected=”B1” Connected Conference – Caller=”C”, Called=”A”, Connected=”A” Conference – Caller=”C”, Called=” CFB-2”, Connected=” CFB-2” For D: Connected Conference – Caller=”D”, Called=”E”, Connected=”E” conference – Caller=”D”, Called=” CFB-1”, Connected=” CFB-1” For E: Connected Conference – Caller=”E”, Called=”D”, Connected=”D” Conference – Caller=”E”, Called=” CFB-1”, Connected=” CFB-1”

IPv6 Use Cases

The use cases related to IPv6 are provided below:

Register CTI Port with IPv4 when Unified CM is IPv6 Disabled and Common Device Configuration is IPv4 .

Steps	Expected Result
1. Enterprise parameter for IPv6 is disabled. IP addressing mode for CTI Port = IPv4 only on common device config page. 2. Open provider and do a LineNegotiateExtensionVersion with the higher bit set on both dwExtLowVersion and dwExtHighVersion 3. Application does a LineOpen with new Ext ver. The lineopen will be delayed till user specifies the Addressing mode 4. Application uses CCiscoLineDevSpecificSetIPAddressMode to set the addressing mode as IPv4. Application uses CciscoLineDevSpecificSendLineOpen to trigger Lineopen.	Application is able to register CTI Port with IPv4 address.

Register CTI Port with IPv6 when Unified CM is IPv6 Disabled and Common Device Configuration is IPv6.

Steps	Expected Result
1. Enterprise parameter for IPv6 is disabled. IP addressing mode for CTI Port = IPv6 only on common device config page. 2. Open provider and do a LineNegotiateExtensionVersion with the higher bit set on both dwExtLowVersion and dwExtHighVersion 3. Application does a LineOpen with new Ext ver. The lineopen will be delayed till user specifies the Addressing mode 4. Application uses CCiscoLineDevSpecificSetIPAddressMode to set the addressing mode as IPv6. Application uses CciscoLineDevSpecificSendLineOpen to trigger Lineopen.	Application is not able to register CTI Port. TSP returns error LINEERR_OPERATIONUNAVAIL

Register CTI Port with IPv6 when Unified CM is IPv6 Disabled and Common Device Configuration is IPv4_v6.

Steps	Expected Result
1. Enterprise parameter for IPv6 is disabled. IP addressing mode for CTI Port = IPv4_v6 on common device config page. 2. Open provider and do a LineNegotiateExtensionVersion with the higher bit set on both dwExtLowVersion and dwExtHighVersion 3. Application does a LineOpen with new Ext ver. The lineopen will be delayed till user specifies the Addressing mode 4. Application uses CCiscoLineDevSpecificSetIPAddressMode to set the addressing mode as IPv6. Application uses CciscoLineDevSpecificSendLineOpen to trigger Lineopen.	Application is not able to register CTI Port. TSP returns error LINEERR_OPERATIONUNAVAIL

IPv6 Phone A calls IPv6 Phone B

Steps	Expected Result
1. Enterprise parameter for IPv6 is enabled.	
2. Open two lines A and B	
3. Phone A which is IPv6 calls Phone B which is IPv6	
4. Events at Phone B	FireCallState = Offering, Do a GetlineCallInfo. LineCallInfo contains the following in devspecific part, FarEndIPAddress: Blank FarEndIPAddressIpv6: IPv6 address of A
5. While Media is established:	
• Events on phone A	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv6 address of B. ReceptionRTPDestinationAddress = IPv6 address of A.
• Event on phone B	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv6 address of A. ReceptionRTPDestinationAddress = IPv6 address of B.

IPv4_v6 Phone calls IPv6 Phone.

Steps	Expected Result
1. Enterprise parameter for IPv6 is enabled. 2. Open two lines A and B 3. Phone A which is IPv4_v6 calls Phone B which is IPv6 4. Events at Phone B	FireCallState = Offering, Do a GetlineCallInfo. LineCallInfo contains the following in devspecific part, FarEndIPAddress: IPv4 address of A FarEndIPAddressIpv6: IPv6 address of A
5. While Media is established: • Events on phone A	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv6 address of B. ReceptionRTPDestinationAddress = IPv6 address of A.
• Event on phone B	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv6 address of A. ReceptionRTPDestinationAddress = IPv6 address of B.

IPv4 Phone Calls IPv6 Phone.

Steps	Expected Result
1. Enterprise parameter for IPv6 is enabled.	
2. Open two lines A and B	
3. Phone A which is IPv4 calls Phone B which is IPv6	
4. Events at Phone B	FireCallState = Offering, Do a GetlineCallInfo. LineCallInfo contains the following in devspecific part, FarEndIPAddress: IPv4 address of A FarEndIPAddressIpv6:
5. While Media is established:	
• Events on phone A	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv4 address of MTP Resource. ReceptionRTPDestinationAddress = IPv4 address of A.
• Event on phone B	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv6 address of MTP Resource. ReceptionRTPDestinationAddress = IPv6 address of B.

IPv6 Phone Calls IPv4 Phone.

Steps	Expected Result
1. Enterprise parameter for IPv6 is enabled.	
2. Open two lines A and B	
3. Phone A which is IPv6 only calls Phone B which is IPv4	
4. Events at Phone B	FireCallState = Offering, Do a GetlineCallInfo. LineCallInfo contains the following in devspecific part, FarEndIPAddress: FarEndIPAddressIpv6: IPv6 address of A
5. While Media is established:	
• Events on phone A	Do a GetLinecallInfo, LineCallInfo will contain the following in devspecific part, TransmissionRTPDestinationAddress = IPv6 address of MTP Resource. ReceptionRTPDestinationAddress = IPv6 address of A.
• Event on phone B	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv4 address of MTP Resource. ReceptionRTPDestinationAddress = IPv4 address of B.

IPv6 Phone Calls IPv4_v6 Phone.

Steps	Expected Result
1. Enterprise parameter for IPv6 is enabled.	
2. Phone A which is IPv6 only calls Phone B which is IPv4_v6 only.	
3. Open lines A and B	
4. Events at Phone B	Existing Call, Do a GetlineCallInfo. LineCallInfo contains the following in devspecific part, FarEndIPAddress: FarEndIPAddressIpv6: IPv6 address of A
5. While Media is established:	
• Events on phone A	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv6 address of MTP Resource. ReceptionRTPDestinationAddress = IPv6 address of A.
• Event on phone B	Do a GetLinecallInfo, LineCallInfo contains the following in devspecific part, TransmissionRTPDestinationAddress = IPv6 address of Phone A. ReceptionRTPDestinationAddress = IPv6 address of B.

Common Device C onfiguration Device Mode Changes from IPv4_v6 to IPv4.

Steps	Expected Result
User changes the device configuration on common device configuration from IPv4_v6 to IPv4 only	Application receives LineDevSpecific for the opened CTI Ports/RP in the device config indicating that Addressing mode has changed. All lines registered as IPv6 get a LINE_CLOSE Event. Application can then re-register these lines later.

Common Device Configuration Device Mode Changes from IPv4 to IPv6 .

Steps	Expected Result
User changes the device configuration on common device configuration from IPv4 only to IPv6 only	Application receives LineDevSpecific for the opened CTI Ports/RP in the device config indicating that Addressing mode has changed. All lines registered as IPv4 get a LINE_CLOSE Event. Application can then re-register these lines later.

Direct Transfer Across Lines

Use cases related to Direct Transfer Across Lines feature are mentioned below:

Note The device mentioned in the use cases also apply to SCCP device and SIP TNP phones when Direct Transfer is issued from application.

Direct Transfer across Lines on RoundTable Phones via Application

Device A, B, and C where B is roundtable phone and has line B1 and B2 configured.

Action	Expected Events
A →B1 is connected, C →B2 is on hold	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected B1 For B1: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1, Connected = A For B2: LINE_CALLSTATE param1=x100, HOLD Caller = C, Called = B2 , Connected = C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2, Connected = B2
Application sends CciscoLineDevSpecificDirectTransfer on B1 with B2 as consult call	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected C For B1: Call goes IDLE For B2: Call goes IDLE For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2, Connected = A

Direct Transfer on Same Line on RoundTable Phones via Application

Device A, B, C where B is roundtable phone.

Action	Expected Events
A → B (c1) is connected, C → B (c2) is on hold	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B For B: Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A Call-2 LINE_CALLSTATE param1=x100, HOLD Caller = C, Called = B, Connected = C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B, Connected = B
Application sends CciscoLineDevSpecificDirectTransfer on B (c1) with c2 as consult call	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected C For B: Call-1 and Call-2 will go IDLE For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B, Connected = A

Direct Transfer Across Lines on RoundTable Phones via Application with call in Offering State

Device A, B, C where B is roundtable phone and has line B1 and B2 configured.

Action	Expected Events
A (c1) → B1(c2) is on hold, B2 (c3) → C (c4) is ringing	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected B1 For B1: LINE_CALLSTATE param1=x100, HOLD Caller = A, Called = B1, Connected = A For B2: LINE_CALLSTATE param1=x100, RINGBACK Caller = B2, Called = C For C: LINE_CALLSTATE param1=x100, OFFERING Caller = B2, Called = C
Application sends CciscoLineDevSpecificDirectTransfer on B1 (c2) with B2 (c3) as consult call	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected C For B1: Call goES IDLE For B2: Call goes IDLE For C: LINE_CALLSTATE param1=x100, OFFERING Caller = C, Called = B,

Failure of Direct Transfer Calls Across Lines

Device A, B, C where B is roundtable phone and has line B1 and B2 configured.

Action	Expected Events
A (c1) → B1(c2) is on hold, Initiate new call (c3) on B2	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected B1 For B1: LINE_CALLSTATE param1=x100, HOLD Caller = A, Called = B1, Connected = A For B2: LINE_CALLSTATE param1=x100, DIALTONE
Application sends CciscoLineDevSpecificDirectTransfer on B1 (c2) with B2 (c3) as consult call	CciscoLineDevSpecificDirectTransfer gets error as LINEERR_INVALCALLSTATE.

Direct Transfer Calls Across Lines in Conference Scenario

Device A, B, C, D and E where C is roundtable phone and has line C1 and C2 configured.

Action	Expected Events
A/B/C1 in conference, B is controller, call on C1 is in hold state. C2 /D/E in conference, D is controller, call on C2 is in connect state.	For A: CONNECTED CONFERENCED Caller = A, called = B, connected = B CONFERENCED Caller = A, called = C1, connected = C1
	For B: CONNECTED CONFERENCED Caller = A, called = B, connected = B CONFERENCED Caller = B, called = C1, connected = C1
	For C1: ONHOLD CONFERENCED Caller = B, called = C1, connected = B CONFERENCED Caller = C1, called = A, connected = A
	For C2: CONNECTED CONFERENCED Caller = C2, called = D, connected = D CONFERENCED Caller = C2, called = E, connected = E
	For D: CONNECTED CONFERENCED Caller = D, called = C1, connected = C1 CONFERENCED Caller = D, called = E, connected = E

Action	Expected Events
	For E: CONNECTED CONFERENCED Caller = D, called = E, connected = D CONFERENCED Caller = E, called = C2, connected = C2
Application sends CciscoLineDevSpecificDirectTransfer on C1 with C2-call as consult call	CciscoLineDevSpecificDirectTransfer will succeed. For A: CONNECTED CONFERENCED Caller = A, called = B, connected = B CONFERENCED Caller = A, called = CB-2, connected = CB-2 For B: CONNECTED CONFERENCED Caller = A, called = B, connected = B CONFERENCED Caller = B, called = CB-2, connected = CB-2 For C1: IDLE For C2: IDLE For D: CONNECTED CONFERENCED Caller = D, called = CB-1, connected = CB-1 CONFERENCED Caller = D, called = E, connected = E For E: CONNECTED CONFERENCED Caller = D, called = E, connected = D CONFERENCED Caller = E, called = CB-1, connected = CB-1

Connect Transfer Across Lines on RoundTable Phones

Device A, B, C where B is roundtable phone and has line B1 and B2 configured.

Action	Expected Events
A → B1 is connected, C → B2 is on hold	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected B1 For B1: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1, Connected = A For B2: LINE_CALLSTATE param1=x100, HOLD Caller = C, Called = B2, Connected = C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2, Connected = B2
User performs connect transfer on B.	For A: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B1 Connected C For B1: Call goes IDLE For B2: Call goes IDLE For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = C, Called = B2, Connected = A

Swap or Cancel Support

Use cases related to Swap or Cancel feature are mentioned below:

Connected Transfer

Device A, B, C where A is a Cisco Unified IP Phone (future version)..

Action	Expected Events
A → C is on hold A → B is connected,	For A: Call-1 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C Call-2 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
A press transfer	For A: Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = B Connected B Call-3 DIALTONE
A picks "Active Calls"	Call-3 goes IDLE
A picks call (A→C) and presses transfer to complete transfer	For A: Both calls go IDLE For B1: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = B

Connected Transfer on Phones with Shared Lines

Device A, B, C, A' where A and A' are sharedline.

Action	Expected Events
A → C is on hold A → B is connected,	For A: Call-1 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C Call-2 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A For A': Call-1 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C Call-2 LINE_CALLSTATE param1=x100, CONNECTED_INACTIVE Caller = A, Called = B Connected B
User performs connected transfer on Cisco Unified IP phone (future version)	For A and A': All calls go IDLE For B1: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = B

Connected Transfer: Initiate from Phone, Complete from CTI

Device A, B, C .

Action	Expected Events
A → C is on hold A → B is connected,	For A: Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
Application sends either CompleteTransfer or DirectTransfer on A	For A and A': All calls go IDLE For B1: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = B

Consult Transfer: Resume Primary Call (Implicit Cancel).

Action	Expected Events
A → B A setup consult transfers to C And C answer	For A: Call-1 LINE_CALLSTATE param1=x100, ONHOLDPENDINGTRANSFER Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C Connected C For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
A press resume to resume A→ B call	For A: Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

Consult Transfer: Swap Calls.

Action	Expected Events
A →B A setup consult transfer to C And C answer	For A: Call-1 LINE_CALLSTATE param1=x100, ONHOLDPENDINGTRANSFER Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C Connected C
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
A press Swap	For A: The scenario will look exactly the same when resume primary call. Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C

Action	Expected Events
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
A press "Transfer" to complete transfer	For A: Calls go IDLE For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = C For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = B

Consult Transfer on Phone: Swap Calls; CTI sends SetupTransfer on Connected Call

Action	Expected Events
A → B A setup consult transfer to C And C answer	For A: Call-1 LINE_CALLSTATE param1=x100, ONHOLDPENDINGTRANSFER Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C Connected C
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

Action	Expected Events
A press Swap	For A: The scenario will look exactly the same when resume primary call. Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
Application calls LineSetupTransfer on A's connected call (A→B) to initiate transfer	Request succeeds as phone cancels existing feature plan and allow CTI request to go through.

Consult Transfer: Swap and Cancel

Action	Expected Events
A → B A setup consult transfer to C And C answer	For A: Call-1 LINE_CALLSTATE param1=x100, ONHOLDPENDINGTRANSFER Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C Connected C
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

Action	Expected Events
A press Swap	For A: The scenario will look exactly the same when resume primary call. Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
A presses Cancel	No TSP event since it is handled during swap operation

RoundTable Connected Conference.

Action	Expected Events
A → B A puts call on hold A creates new call to C, C answer	For A: Call-1 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C Connected C For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

Action	Expected Events
A presses "Conference"	For A: Call-1 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, ONHOLDPENDINGCONFENRENCE Caller = A, Called = C Connected C Call-3 DIALTONE
A picks active call (A→ C) on phone UI, and presses "Conference" to complete the conference	For A: CONNECTED CONFERENCED Caller=A, called = B, connected = B CONFERENCED Caller = A, called = C, connected = C Call-3 IDLE For B: For A: CONNECTED CONFERENCED Caller=A, called = B, connected = B CONFERENCED Caller = B, called = C, connected = C For C: For A: CONNECTED CONFERENCED Caller=A, called = C, connected = C CONFERENCED Caller = C, called = B, connected = B

RoundTable Connected Conference: Cancel.

Action	Expected Events
A → B A puts call on hold A creates new call to C, C answers	For A: Call-1 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C Connected C For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
A presses "Conference"	For A: Call-1 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, CONFERENCED Caller = A, Called = C Connected C Call-3 LINE_CALLSTATE param1=x100, ONHOLDPENDINGCONFENRENCE Caller = A, Called = C Connected C Call-4 DIALTONE

Action	Expected Events
A picks "Active Calls"	For A: Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C Call-3 / Call-4 IDLE
A presses Cancel softkey	For A: Call-1 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

Set Up Consult Conference from RT, then Swap and Complete Conference from RT.

Action	Expected Events
A → B A sets up conference to C, C answer	For A: ONHOLDPENDINGCONF CONFERENCED Caller = A, called = B, connected = B CONNECTED Caller = A, called = C, connected = C
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

Action	Expected Events
A presses "Swap"	For A: Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x100, HOLD Caller = A, Called = C Connected C
A presses "Conference" to complete conference	For A: CONNECTED CONFERENCED Caller=A, called = B, connected = B CONFERENCED Caller = A, called = C, connected = C For B: CONNECTED CONFERENCED Caller=A, called = B, connected = B CONFERENCED Caller = B, called = C, connected = C For C: For A: CONNECTED CONFERENCED Caller=A, called = C, connected = C CONFERENCED Caller = C, called = B, connected = B

Set Up Consult Conference from RT, then Swap and Cancel from Phone with Shared Line Scenario

A and A’ are shared lines..

Action	Expected Events
A → B A sets up conference to C, C answers	For A: ONHOLDPENDINGCONF CONFERENCED Caller = A, called = B, connected = B CONNECTED Caller = A, called = C, connected = C For A' CONNECTED INACTIVE Caller = A, celled = B, connected = B CONNECTED INACTIVE Caller = A, celled = C, connected = C For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A
A presses "Swap"	For A: The scenario looks the same when primary call resumes Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C

Action	Expected Events
A presses "Cancel"	For A: Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected = C
	For A' Call-1 LINE_CALLSTATE CONNECTED INACTIVE Caller = A, Called = B Connected = B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected = C
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

Set Up Consult Conference from RT: Resume Primary Call (Implicit Cancel).

Action	Expected Events
A → B A sets up conference to C, C answer	For A: ONHOLDPENDINGCONF CONFERENCED Caller = A, called = B, connected = B CONNECTED Caller = A, called = C, connected = C
	For A' CONNECTED INACTIVE Caller = A, celled = B, connected = B CONNECTED INACTIVE Caller = A, celled = C, connected = C
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

Action	Expected Events
A resumes A→B call	For A: Call-1 LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B Connected B Call-2 LINE_CALLSTATE param1=x400, HOLD Caller = A, Called = C Connected C
	For B: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = B, Connected = A
	For C: LINE_CALLSTATE param1=x100, CONNECTED Caller = A, Called = C, Connected = A

User is Removed from Standard Supports Connected Xfer/Conf Group.

Action	Expected Events
User is in Standard Supports Connected Xfer/Conf group RT phone A is in user's control list Application does LineInitialize	RT PHONE/LINE is enumerated to APP
Remove user from "Standard Supports Connected Xfer/Conf" user group	APP receives PHONE_REMOVE / LINE_REMOVE

User is Removed from Standard Supports Connected Xfer/Conf Group.

Action	Expected Events
User is in Standard Supports Connected Xfer/Conf group RT phone A is in user's control list Application does LineInitialize	RT PHONE/LINE is enumerated to APP
Remove user from Standard Supports Connected Xfer/Conf user group	APP receives PHONE_REMOVE / LINE_REMOVE

User is Removed from Standard Supports Connected Xfer/Conf Group while Line is Open.

Action	Expected Events
user is in "Standard Supports Connected Xfer/Conf" group RT phone A is in user's control list Application does LineInitialize	RT PHONE/LINE is enumerated to APP
App sends LineOpen to open line on Cisco Unified IP phone (future version) phone	Successful
Remove user from Standard Supports Connected Xfer/Conf group	TSP sends LINE_CLOSE APP receives LINE_REMOVE

User is Added to Standard Supports Connected Xfer/Conf Group.

Action	Expected Events
user is not in "Standard Supports Connected Xfer/Conf" group RT phone A is in user's control list Application does LineInitialize	RT PHONE/LINE is not enumerated to APP
Add user to Standard Supports Connected Xfer/Conf group	APP receives PHONE_CREATE / LINE_CREATE

Drop Any Party

Use cases related to Drop Any Party feature are mentioned below:

Conference: Unified CM Service Parameter Advanced Ad Hoc Conference Enabled = False.

Action	Expected Events
A,B,C and D are in conference; B is conference Controller.	Conference Model: Each line in conference will be having 4 callLegs, 3 conferenced and 1 connected
	CallLegs on A: Connected - to Conference Bridge Conferenced - (Connected Id - B) Conferenced - (Connected Id - C) Conferenced - (Connected Id - D)
	CallLegs on B: Connected - to Conference Bridge Conferenced - (Connected Id - A) Conferenced - (Connected Id - C) Conferenced - (Connected Id - D)
	CallLegs on C: Connected - to Conference Bridge Conferenced - (Connected Id - A) Conferenced - (Connected Id - B) Conferenced - (Connected Id - D)
	CallLegs on D: Connected - to Conference Bridge Conferenced - (Connected Id - A) Conferenced - (Connected Id - B) Conferenced - (Connected Id - C)
Application does a LineOpen (B) with new Ext ver.	

Action	Expected Events
1. Application does LineRemoveFromConference on the 'Conferenced' callLeg on B which is connected to A.	A is dropped out of conference. CallLegs after the Party is dropped from Conference: Each line in conference will be having 4 callLegs, 2 Conferenced,1 IDLE and 1 connected
	CallLegs on A: All 4 CallLegs will be in IDLE state
	CallLegs on B: Connected - to Conference Bridge Conferenced - (Connected Id - C) Conferenced - (Connected Id - D) IDLE - (on the conferenced callLeg which was connected to A)
	CallLegs on C: Connected - to Conference Bridge IDLE - (on the conferenced callLeg which was connected to A) Conferenced - (Connected Id - B) Conferenced - (Connected Id - D)
	CallLegs on D: Connected - to Conference Bridge IDLE - (on the conferenced callLeg which was connected to A) Conferenced - (Connected Id - B) Conferenced - (Connected Id - C) **Note** All IDLE CallLegs will have CallStateChange Reason as CtiDropConferee.
Application does a LineOpen (A) with new Ext ver.	
2. Application does LineRemoveFromConference on the 'Conferenced' callLeg on A which is connected to B.	Error Message LINEERR_OPERATIONUNAVAIL will be sent to application

Conference - Unified CM Service Parameter Advanced Ad Hoc Conference Enabled = True'

Action	Expected Events
A,B,C and D are in conference; B is conference Controller.	Conference Model: Each line in conference will be having 4 callLegs, 3 conferenced and 1 connected
	CallLegs on A: Connected - to Conference Bridge Conferenced - (Connected Id - B) Conferenced - (Connected Id - C) Conferenced - (Connected Id - D)
	CallLegs on B: Connected - to Conference Bridge Conferenced - (Connected Id - A) Conferenced - (Connected Id - C) Conferenced - (Connected Id - D)
	CallLegs on C: Connected - to Conference Bridge Conferenced - (Connected Id - A) Conferenced - (Connected Id - B) Conferenced - (Connected Id - D)
	CallLegs on D: Connected - to Conference Bridge Conferenced - (Connected Id - A) Conferenced - (Connected Id - B) Conferenced - (Connected Id - C)
Application does a **LineOpen (A)** with new Ext ver. Application does LineRemoveFromConference on the 'Conferenced' callLeg on A which is connected to B.	

Action	Expected Events
1. Drop Ad Hoc Conference = Never	B is dropped out of conference. **CallLegs after the Party is dropped from Conference**: Each line in conference will be having 4 callLegs, 2 Conferenced,1 IDLE and 1 connected
	CallLegs on B: All 4 CallLegs will be in IDLE state
	CallLegs on A: Connected - to Conference Bridge Conferenced - (Connected Id - C) Conferenced - (Connected Id - D) IDLE - (on the conferenced callLeg which was connected to B)
	CallLegs on C: Connected - to Conference Bridge IDLE - (on the conferenced callLeg which was connected to B) Conferenced - (Connected Id - A) Conferenced - (Connected Id - D)
	CallLegs on D: Connected - to Conference Bridge IDLE - (on the conferenced callLeg which was connected to B) Conferenced - (Connected Id - A) Conferenced - (Connected Id - C) **Note** All IDLE CallLegs will have CallStateChange Reason as CtiDropConferee.
2. Drop Ad Hoc Conference = 'When Conference Controller Leaves'	B is dropped out of conference and Conference will be ended. **CallLegs after the Party is dropped from Conference**: Each line in conference will be having 4 callLegs, all in IDLE state **CallLegs on A,B,C and D**: All 4 CallLegs will be in IDLE state

Shared Line-Scenario

Action	Expected Events
A,B,C and A' are in conference; A is conference Controller Unified CM Parameter "Drop Ad Hoc Conference = Never"	**Conference Model**: Lines B and C in conference will be having 4 callLegs, 3 conferenced and 1 connected Lines A and A' will be having 8 CallLegs
	CallLegs on A: Connected - to Conference Bridge (Active) Conferenced - (caller Id - A ;Called Id - B; Connected Id - B) (Active) Conferenced - (caller Id - A ;Called Id - C; Connected Id - C) (Active) Conferenced - (caller Id - A ;Called Id - A' ; Connected Id - A') (Active) Connected - to Conference Bridge (Remote in Use) Conferenced - (caller Id - A' ;Called Id - B; Connected Id - B) (Remote in Use) Conferenced - (caller Id - A' ;Called Id - C; Connected Id - C) (Remote in Use) Conferenced - (caller Id - A' ;Called Id - A; Connected Id - A) (Remote in Use)
	CallLegs on A': Connected - to Conference Bridge (Active) Conferenced - (caller Id - A' ;Called Id - B; Connected Id - B) (Active) Conferenced - (caller Id - A' ;Called Id - C; Connected Id - C) (Active) Conferenced - (caller Id - A' ;Called Id - A; Connected Id - A) (Active) Connected - to Conference Bridge (Remote in Use) Conferenced - (caller Id - A ;Called Id - B; Connected Id - B) (Remote in Use) Conferenced - (caller Id - A ;Called Id - C; Connected Id - C) (Remote in Use) Conferenced - (caller Id - A ;Called Id - A'; Connected Id - A') (Remote in Use)

Action	Expected Events
	CallLegs on B: Connected - to Conference Bridge Conferenced - (caller Id - B ;Called Id - A; Connected Id - A) Conferenced - (caller Id - B ;Called Id - C; Connected Id - C) Conferenced - (caller Id - B ;Called Id - A'; Connected Id - A')
	CallLegs on C: Connected - to Conference Bridge Conferenced - (caller Id - C ;Called Id - A; Connected Id - A) Conferenced - (caller Id - C ;Called Id - B; Connected Id - B) Conferenced - (caller Id - C ;Called Id - A' ; Connected Id - A')
Application does a **LineOpen (A)** with new Ext ver. Unified CM Parameter '**Advanced Ad Hoc Conference Enabled = False**'	
1. Application does **LineRemoveFromConference** on the 'Conferenced' CallLeg on A which is connected to B and mode is "Inactive or Remote In use".	Error LINEERR_INVALCALLSTATE is sent to application.

Action	Expected Events
2. Application does **LineRemoveFromConference** on the 'Conferenced' CallLeg on A which is connected to B and mode is 'Active'.	B will be dropped out of conference. LINECALLSTATE Event will be sent to Application with state = Idle.
	CallLegs after the Party is dropped from Conference: **CallLegs on A**: Connected - to Conference Bridge (Active) IDLE - (on the conferenced callLeg which was connected to A - B) Conferenced - (caller Id - A ;Called Id - C; Connected Id - C) (Active) Conferenced - (caller Id - A ;Called Id - A'; Connected Id - A') (Active) Connected - to Conference Bridge (Remote in Use) IDLE - (on the conferenced callLeg which was connected to A' - B) Conferenced - (caller Id - A' ;Called Id - C; Connected Id - C) (Remote in Use) Conferenced - (caller Id - A' ;Called Id - A; Connected Id - A) (Remote in Use)
	CallLegs on A': Connected - to Conference Bridge (Active) IDLE - (on the conferenced callLeg which was connected to A' - B) Conferenced - (caller Id - A' ;Called Id - C; Connected Id - C) (Active) Conferenced - (caller Id - A' ;Called Id - A; Connected Id - A) (Active) Connected - to Conference Bridge (Remote in Use) IDLE - (on the conferenced callLeg which was connected to A - B) Conferenced - (caller Id - A ;Called Id - C; Connected Id - C) (Remote in Use) Conferenced - (caller Id - A ;Called Id - A'; Connected Id - A') (Remote in Use)
	CallLegs on B: All 4 CallLegs are in IDLE state

Action	Expected Events
	CallLegs on C: Connected - to Conference Bridge Conferenced - (caller Id - C ;Called Id - A; Connected Id - A) IDLE - (on the conferenced callLeg which was connected to C - B) Conferenced - (caller Id - C ;Called Id - A'; Connected Id - A')
Application does a **LineOpen (B)** with new Ext ver. Unified CM Parameter **Advanced Ad Hoc Conference Enabled = True**	

Action	Expected Events
3. Application does **LineRemoveFromConference** on the 'Conferenced' CallLeg on B which is connected to A and mode is "Active".	A will be dropped out of conference. LINECALLSTATE Event will be sent to Application with state = Idle.
	CallLegs after the Party is dropped from Conference: **CallLegs on A**: IDLE - (on the Connected callLeg which was connected to Conference Bridge,A- CFB) IDLE - (on the conferenced callLeg which is connected to A - B) IDLE - (on the conferenced callLeg which is connected to A - C) IDLE -(on the conferenced callLeg which is connected to A - A') Connected - to Conference Bridge (Remote in Use) Conferenced - (caller Id - A' ;Called Id - C; Connected Id - C) (Remote in Use) Conferenced - (caller Id - A' ;Called Id - B; Connected Id - B) (Remote in Use)
	CallLegs on A': IDLE - (on the Connected callLeg which was connected to Conference Bridge,A - CFB) IDLE - (on the conferenced callLeg which is connected to A - B) IDLE - (on the conferenced callLeg which is connected to A - C) IDLE -(on the conferenced callLeg which is connected to A - A') Connected - to Conference Bridge Conferenced - (caller Id - A' ;Called Id - C; Connected Id - C) (Active) Conferenced - (caller Id - A' ;Called Id - B; Connected Id - B) (Active)
	CallLegs on B: Connected - to Conference Bridge Conferenced - (caller Id - B ;Called Id - A; Connected Id - A') IDLE - (on the conferenced callLeg which was connected to B - A) Conferenced - (caller Id - B ;Called Id - C; Connected Id - C)

Action	Expected Events
	CallLegs on C: Connected - to Conference Bridge Conferenced - (caller Id - C ;Called Id - A'; Connected Id - A') IDLE - (on the conferenced callLeg which was connected to C - A) Conferenced - (caller Id - C ;Called Id - B; Connected Id - B)

Chained Conference.

Action	Expected Events
A,B and CB2 are in conference(CB1); B is conference Controller C,D and E are in Conference (CB2); D is conference Controller Unified CM Parameter Advanced Ad Hoc Conference Enabled = True Application does a **LineOpen (A)** with new Ext ver.	
1. Application does **LineRemoveFromConference** on the Conferenced" CallLeg on A which is connected to B.	B is disconnected and dropped out of Conference. A is now in conference with CB2. LINECALLSTATE Event is sent to Application for Line B with state = Idle.

C-Barge: Unified CM Service Parameter Advanced Ad Hoc Conference Enabled = True.

Action	Expected Events
B call A and A'; A answers the call and on A' do c-Barge; A,B and A' will be in conference; A is conference Controller Unified CM Parameter "Drop Ad Hoc Conference = Never" Application does a LineOpen (A) with new Ext ver.	

Action	Expected Events
Application does a LineOpen (A) with new Ext ver. 1. Application does **LineRemoveFromConference** on the "Conferenced" CallLeg on A which is connected to B and mode is Active	B is dropped out of conference. LINECALLSTATE Event will be sent to Application with state = Idle. **CallLegs after the Party is dropped from Conference**: **CallLegs on A**: Connected - (on the conferenced callLeg which was connected to A - A') (Active) Connected - on the conferenced callLeg which was connected to A' - A) (Remote in Use) IDLE - (on the conferenced callLeg which was connected to A - B) IDLE - (on the connected callLeg which is connected to conference Bridge; A - CFB) IDLE - (on the conferenced callLeg which was connected to A' - B) IDLE - (on the connected callLeg which is connected to conference Bridge; A' - CFB)
	CallLegs on A': Connected - (on the conferenced callLeg which was connected to A' - A) (Active) Connected - on the conferenced callLeg which was connected to A - A') (Remote in Use) IDLE - (on the conferenced callLeg which was connected to A - B) IDLE - (on the connected callLeg which is connected to conference Bridge; A - CFB) IDLE - (on the conferenced callLeg which was connected to A' - B) IDLE - (on the connected callLeg which is connected to conference Bridge; A' - CFB)
	CallLegs on B: All 4 CallLegs are in IDLE state A' is dropped out of conference. LINECALLSTATE Event will be sent to Application with state = Idle.

Action	Expected Events
2. Application does LineRemoveFromConference on the Conferenced CallLeg on A which is connected to A' and mode is Active.	**CallLegs after the Party is dropped from Conference**: **CallLegs on A**: Connected -(on the conferenced callLeg which was connected to A - B) (Active) IDLE -(on the conferenced callLeg which was connected to A' - B) (Remote in Use) IDLE - (on the conferenced callLeg which was connected to A - A') (active) IDLE - (on the connected callLeg which is connected to conference Bridge; A - CFB) IDLE - (on the conferenced callLeg which was connected to A' - A) (Remote in Use) IDLE - (on the connected callLeg which is connected to conference Bridge; A' - CFB)
	CallLegs on A': Connected -(on the conferenced callLeg which was connected to A - B) (Remote in Use) IDLE -(on the conferenced callLeg which was connected to A' - B) IDLE - (on the conferenced callLeg which was connected to A - A') (active) IDLE - (on the connected callLeg which is connected to conference Bridge; A - CFB) IDLE - (on the conferenced callLeg which was connected to A' - A) (Remote in Use) IDLE - (on the connected callLeg which is connected to conference Bridge; A' - CFB)
	CallLegs on B: Connected -(on the conferenced callLeg which was connected to B - A) IDLE -(on the conferenced callLeg which was connected to A' - B) IDLE - (on the connected callLeg which is connected to conference Bridge; B - CFB)

Park Monitoring

Use cases related to Park Monitoring feature are mentioned below:

Park Monitoring Feature Disabled

Setup:

The Park Monitoring message flag is disabled by default.

Cisco Unified IP Phones (future version) running SIP: A(3000), B(3001)

All lines are monitered by TSP

Action	Expected Events
1. A(3000) calls B(3001)	
2. B(3001) receives the call and parks the call	Application will not be notified about the New Parked call through LINE_NEWCALL event as the park Monitoring flag is disabled.

Park Monitoring Feature Enabled

Setup:

Cisco Unified IP Phones (future version) running SIP: A(3000), B(3001),C(3002)

All lines are monitered by TSP

Action	Expected Events
Scenario 1: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. A(3000) calls B(3001) 3. B(3001) receives the call and parks the call at 5555	Park Status Event on B: At Step 3: Application will be notified about the New Parked call through LINE_NEWCALL event At Step 3: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. Application does a LineGetCallInfo.
	LineCallInfo will contain the following: hline : LH = 1 dwCallID : CallID dwReason :LINECALLREASON_PARKED dwRedirectingIDName : TransactionIDID = Sub1. dwBearerMode: ParkStatus = 2 dwCallerID : ParkDN = 5555 dwCallerName : ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.

Action	Expected Events
Scenario 2: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. A(3000) calls B(3001) 3. B(3001) receives the call and parks the call at 5555 4. The Park Monitoring Reversion Timer expires while the call is still parked.	Park Status Event on B: At Step 3: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. At Step 4: Application will receive the LINE_CALLSTATE event with the Park Status = Reminder. Application does a LineGetCallInfo. LineCallInfo will contain the following: hline : LH = 1 dwCallID : CallID dwReason :LINECALLREASON_PARKED dwRedirectingIDName : TransactionIDID = Sub1. dwBearerMode: ParkStatus = 3 dwCallerID : ParkDN = 5555 dwCallerName : ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.

Action	Expected Events
Scenario 3: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. The Park Monitoring Forward No Retrieve destination configured on B(3001) as C(3002) 3. A(3000) calls B(3001) 4. B(3001) receives the call and parks the call 5. The Park Monitoring Reversion Timer Expires while the call is still parked.	Park Status Event on B: At Step 4: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. At Step 5: Application will receive the LINE_CALLSTATE event with the Park Status = Reminder. At Step 6: Application will receive the LINE_CALLSTATE event with the Park Status = Forwarded Application will receive the LINE_CALLSTATE event with callstate IDLE. The reason code CtiReasonforwardedNoRetrieve will be updated in the LINECALLINFO::dwDevSpecificData.ExtendedCallInfo.dwExtendedCallReason = CtiReasonforwardedNoRetrieve.
6. The Park Monitoring Forward No Retrieve timer expires and now the call is forwarded to the Park Monitoring Forward No Retrieve Destination C(3002).	Application does a LineGetCallInfo. LineCallInfo will contain the following: hline : LH = 1 dwCallID : CallID dwReason :LINECALLREASON_PARKED dwRedirectingIDName : TransactionIDID = Sub1. dwBearerMode: ParkStatus = 6 dwCallerID : ParkDN = 5555 dwCallerName : ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.

Action	Expected Events
Scenario 4: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. A(3000) calls B(3001) 3. B(3001) receives the call and parks the call 4. A(3000) hangs up the call.	Park Status Event on B: At Step 3: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. At Step 4: Application will receive the LINE_CALLSTATE event with the Park Status = Abandoned. Application will receive the LINE_CALLSTATE event with callstate IDLE. Application does a LineGetCallInfo.
	LineCallInfo will contain the following: hline : LH = 1 dwCallID : CallID dwReason :LINECALLREASON_PARKED dwRedirectingIDName TransactionIDID = Sub1. dwBearerMode: ParkStatus = 4 dwCallerID : ParkDN = 5555 dwCallerName : ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.

Action	Expected Events
Scenario 5: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. A(3000) calls B(3001) 3. B(3001) receives the call and parks the call 4. The Park Monitoring Reversion Timer Expires while the call is still parked. 5. C(3002) retrieves the call	Park Status Event on B: At Step 3: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. At Step 4: Application will receive the LINE_CALLSTATE event with the Park Status = Reminder. At Step 5: Application will receive the LINE_CALLSTATE event with the Park Status = Retrieved. Application will receive the LINE_CALLSTATE event with callstate IDLE.
	Application does a LineGetCallInfo. hline: LH = 1 dwCallID: CallID dwReason: LINECALLREASON_PARKED dwRedirectingIDName: TransactionIDID = Sub1. dwBearerMode: ParkStatus = 5 dwCallerID: ParkDN = 5555 dwCallerName: ParkDNPartition = P1 dwcalled: ParkedParty = 3000 dwCalledIDName: ParkedPartyPartition = P1.

Action	Expected Events
Scenario 6: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. The Park Monitoring Forward No retrieve destination not configuered. 3. A(3000) calls B(3001) 4. B(3001) receives the call and parks the call 5. The Park Monitoring Reversion Timer Expires while the call is still parked 6. The Park Monitoring Forward No Retrieve timer expires and the call is forwarded to the Parkers line.	Park Status Event on B At Step 4: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. At Step 5: Application will receive the LINE_CALLSTATE event with the Park Status = Reminder. At Step 6: Application will receive the LINE_CALLSTATE event with the Park Status = Forwarded. Application will receive the LINE_CALLSTATE event with callstate IDLE. Application does a LineGetCallInfo.
	LineCallInfo will contain the following: hline: LH = 1 dwCallID: CallID dwReason: LINECALLREASON_PARKED dwRedirectingIDName: TransactionIDID = Sub1. dwBearerMode: ParkStatus = 6 dwCallerID: ParkDN = 5555 dwCallerName: ParkDNPartition = P1 dwcalled: ParkedParty = 3000 dwCalledIDName: ParkedPartyPartition = P1.

Action	Expected Events
Scenario 7: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. The Park Monitoring Forward No retrieve destination configuered as self(Parkers Line) 3. A(3000) calls B(3001) 4. B(3001) receives the call and parks the call 5. The Park Monitoring Reversion Timer Expires while the call is still parked 6. The Park Monitoring Reversion Timer Expires while the call is still parked 7. The Park Monitoring Forward No Retrieve timer expires and the call is forwarded to the Parkers line.	Park Status Event on B At Step 5: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. At Step 6: Application will receive the LINE_CALLSTATE event with the Park Status = Reminder. At Step 7: Application will receive the LINE_CALLSTATE event with the Park Status = Forwarded. Application will receive the LINE_CALLSTATE event with callstate IDLE.
	Application does a LineGetCallInfo. LineCallInfo will contain the following: hline: LH = 1 dwCallID: CallID dwReason: LINECALLREASON_PARKED dwRedirectingIDName: TransactionIDID = Sub1. dwBearerMode: ParkStatus = 6 dwCallerID: ParkDN = 5555 dwCallerName: ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.

Parked Call Exists

Setup:

Cisco Unified IP Phones (future version) running SIP: A(3000), B(3001).

B is not monitered by TSP.

Action	Expected Events
Scenario 1: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. A(3000) calls B(3001) 3. B(3001) receives the call and parks the call 4. Now the Line B(3001) is monitered by TSP	Park Status Event on B: At Step 4: Application will be notified about the Parked call through LINE_NEWCALL event.when ever cisco TSP recives the LINE_PARK_STATUS event for already parked call. Application does a LineGetCallInfo. LineCallInfo will contain the following: hline : LH = 1 dwCallID : CallID dwReason :LINECALLREASON_PARKED dwRedirectingIDName TransactionIDID = Sub1. dwBearerMode: ParkStatus = 2 dwCallerID : ParkDN = 5555 dwCallerName : ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.

Shared Line Scenario

Setup:

A(3000) ,D(3003) are Cisco Unified IP Phones (future version) running SIP

B(3001) and B'(3001) are shared lines for Cisco Unified IP Phones (future version) running SIP

C(3002) and C'(3002) are shared lines where C is a Cisco Unified IP Phone (future version) running SIP and C' is a Cisco Unified IP Phone 7900 Series running SIP .

For the shared lines the events will be delivered to the phone which parks the call .Events will not be delivered to the other phone though the line is shared.

Action	Expected Events
Scenario 1: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. A(3000) calls B(3001) 3. B(3001) and B'(3001) starts ringing. B(3001) receives the call and parks the call 4. Park Monitoring reversion timer expires while the call is still parked. 5. D(3003) retrieves the call	Park Status Event on B: At Step 3: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. At Step 4: Application will receive the LINE_CALLSTATE event with the Park Status = Reminder.
	At Step 5: Application will receive the LINE_CALLSTATE event with the Park Status = Retrieved Application will receive the LINE_CALLSTATE event with callstate IDLE.
	Application does a LineGetCallInfo. hline : LH = 1 dwCallID : CallID dwReason :LINECALLREASON_PARKED dwRedirectingIDName :TransactionIDID = Sub1. dwBearerMode: ParkStatus = 5 dwCallerID : ParkDN = 5555 dwCallerName : ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.

Action	Expected Events
Scenario 2: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. The Park Monitoring Forward No retrieve destination configured as B(3001) 3. A(3000) calls B(3001) 4. B(3001) and B'(3001) starts ringing. B(3001)receives the call and parks the call 5. The Park Monitoring Reversion Timer Expires while the call is still parked. 6. The Park Monitoring Forward No Retrieve timer expires and call is forwarded to B(3001).Both B(3001) and B'(3001) starts ringing as they are shared lines.	Park Status Event will be sent only to B not B'. At Step 4: Application will receive the LINE_CALLSTATE event with the Park Status = Parked. At Step 5: Application receives the LINE_CALLSTATE event with the Park Status = Reminder. At Step 6: Application receives the LINE_CALLSTATE event with the Park Status = Forwarded.
	Application receive the LINE_CALLSTATE event with callstate IDLE. Application does a LineGetCallInfo. LineCallInfo contains the following: hline : LH = 1 dwCallID : CallID dwReason :LINECALLREASON_PARKED dwRedirectingIDName : TransactionIDID = Sub1. dwBearerMode: ParkStatus = 6 dwCallerID : ParkDN = 5555 dwCallerName : ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.
Scenario 3: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. A(3000) calls C(3002) 3. C(3002) and C'(3002) starts ringing. C'(3002) receives the call and parks the call 4. D(3003) retrieves the call	Park Status Event on C'. At Step 3: Application is notified about the New Parked call through LINE_NEWCALL event as the call is parked by the Normal TNP phone.

Park Monitoring Feature Disabled

Setup:

The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for line B(3001).

A(3000), D(3003) is a Cisco Unified IP Phones (future version)

Application invokes the Line_open () API on provider to monitor ParkDN

.

Action	Expected Events
Scenario 1: 1. The Park Monitoring message flag is Enabled using SLDST_SET_STATUS_MESSAGES request for Line B(3001). 2. A(3000) calls B(3001) 3. B(3001) receives the call and parks the call	Park Status Event on B: At Step 3: Application receives the LINE_NEW_CALL event for PARKDN. At Step 3: Application receives the LINE_PARK_STATUS event with the Park Status = Parked.
4. The Park Monitoring Reversion Timer Expires while the call is still parked.	At Step 4: Application will receive the LINE_CALL_STATE event with the Park Status = Reminder. Application does a LineGetCallInfo. LineCallInfo will contain the following: hline : LH = 1 dwCallID : CallID dwReason :LINECALLREASON_PARKED dwRedirectingIDName :TransactionIDID = Sub1. dwBearerMode: ParkStatus = 3 dwCallerID : ParkDN = 5555 dwCallerName : ParkDNPartition = P1 dwcalled : ParkedParty = 3000 dwCalledIDName : ParkedPartyPartition = P1.

Logical Partitioning Support

Use cases related to Logical Partitioning feature are mentioned below:

Basic Call Scenario

Basic Call Scenario ; Logical partitioning Enabled = true	
Description	Basic Call failure due to Logical partitioning Feature Policy.
Test Setup	A (VOIP) on one Geolocation A calls B: LineMakeCall on A Dails B (DN) Variant 1: B Geo-Location was not Configured;B(PSTN);Policy Config : Interior to Interior Variant 2: B (PSTN) on another GeoLocation
Expected Results	Variant 1: Call will be successful; Reason: LP_IGNORE. Variant 2: A goes to Proceeding State and then On A there will be a DISCONNECTED call state will be sent to application with cause as LINEDISCONNECTMODE_UNKNOWN.

Redirect Scenario

Redirect Scenario ; Logical partitioning Enabled = true	
Description	Redirect Call failure due to Logical partitioning Feature Policy.
Test Setup	Two Clusters (Cluster1 and Cluster2) configured with logical partition policy that will restrict the VOIP calls from Cluster1 to PSTN calls on Cluster2. (vice versa PSTN to VIOP) A on Cluster1 (VOIP) B on Cluster2 (VOIP) C on Cluster2 (PSTN) A calls B B redirects the call to C
Expected Results	Operation fails with error code LINEERR_OPERATION_FAIL_PARTITIONING_POLICY. Error code is processed on Cluster2
Variants	For Forward Operation same behaviour will be observed.

Transfer Call Scenario

Transfer Call Scenario ; Logical partitioning Enabled = true	
Description	Transfer Call failure due to Logical partitioning Feature Policy.
Test Setup	A (VOIP) in one GeoLocation (GeoLoc 1) B (VOIP) in another GeoLocation(GeoLoc 2) C (PSTN)in same GeoLocation as B (GeoLoc 2) A calls B SetUpTransfer on B. On Consult Call at B; Dials C. Complete Transfer on B.
Expected Results	Operation fails with error code "LINEERR_OPERATIONUNAVAIL".
Variants	For Operation Adhoc Conference same behaviour will be observed.

Join Scenario

Join Scenario; Logical partitioning Enabled = true	
Description	Join failure due to Logical partitioning Feature Policy.
Test Setup	A (VOIP) in one GeoLocation (GeoLoc 1) B (VOIP) in another GeoLocation(GeoLoc 2) C (VOIP)in same GeoLocation as B (GeoLoc 2) D (PSTN) in same GeoLocation as B (GeoLoc 2) B has Three Calls 1. B -> A 2. B -> C 3. B -> D Variant 1: Join on B with B -> A as Primary Call. Variant 2: Join on B with B -> D as Primary Call. Variant 3: Join on B with B -> C as Primary Call.
Expected Results	Variant 1: A, B and C will be in conference. Variant 2: B, C and D will be in conference. Variant 3:Either A or D will be in conference with B and C.

Shared Line Scenario

CallPickUp Scenario ; Logical partitioning Enabled = true	
Description	CallPickUp Failure due to Logical partitioning Feature Policy.
Test Setup	A (PSTN) on one Geolocation - GeoLoc1 B (VOIP) on one Geolocation - GeoLoc1 C (VOIP) on one Geolocation - GeoLoc2 A Dails B B Parks the call C does LineUnPark
Expected Results	Call will be successful on A and A' call will not be present
Variants	Shared line features like barge, cbarge, hold & remote resume should be disabled for calls.

CallPark: Retrieve Scenario

CallPickUp Scenario ; Logical partitioning Enabled = true	
Description	CallPickUp Failure due to Logical partitioning Feature Policy.
Test Setup	A (PSTN) on one Geolocation - GeoLoc1 B (VOIP) on one Geolocation - GeoLoc1 C (VOIP) on one Geolocation - GeoLoc2 A Dails B B Parks the call C does LineUnPark
Expected Results	CallUpark Will fail with error code "LINEERR_OPERATIONUNAVAIL".

Basic Call Scenario

Basic Call Scenario ; Logical partitioning Enabled = true	
Description	Basic Call failure due to Logical partitioning Feature Policy.

Basic Call Scenario ; Logical partitioning Enabled = true	
Test Setup	A (VOIP) on one Geolocation A calls B: LineMakeCall on A Dails B (DN) Variant 1: B Geo-Location was not Configured;B(PSTN);Policy Config: Interior to Interior Variant 2: B (PSTN) on another GeoLocation
Expected Results	Variant 1: Call will be successful; Reason: LP_IGNORE. Variant 2: A goes to Proceeding State and then On A there will be a DISCONNECTED call state will be sent to application with cause as LINEDISCONNECTMODE_UNKNOWN.

Support for Cisco IP Phone 6900 Series

Use cases related to Cisco Unified IP Phone 6900 Series support feature are mentioned below:

Monitoring Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll over Mode when User is Added to New User Group

Monitoring Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 behavior when User is added to new user Group.
Test Setup	A - Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 Phone with Roll Over Mode User is added to New User Group. Application does Line Initialize
Expected Results	Lines on the Cisco Unified IP Phone 7931 will be enumerated. Application would be able to Open Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 and it would be able to control and perform call operations on Phone.

Monitoring Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Monitoring Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 behavior when User is added to new user Group.
Test Setup	A - Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode Step 1: Application does Line Initialize Step 2: User is added to New User Group.
Expected Results	Step 1: Lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 will not be enumerated Application will not be notified about the device A and it will not be able to monitor. Step 2: Application will be receiving PHONE_CREATE and LINE_CREATE events for the Device and lines on that Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode. Now Applications would be able to Monitor and control Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931.

Transfer Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Transfer Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Transfer scenario on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 when User is added to new user Group.

Transfer Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Outbound Roll Over Mode - "Roll Over to any Line" Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. C calls A,A answers SetupTransfer on A. Variants: Application Opens only Line A on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931
Expected Results	Call on A will go to OnHold State. New call will be created on Line B. Application then has to complete Transfer using DTAL feature. Variants: Applications would not be able to Complete Transfer from Application as the Line B is not monitored.

Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is added to New User Group

Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll over Mode when User is added to New User Group	
Description	Testing Conference Scenario on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 when User is added to New User Group.

Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll over Mode when User is added to New User Group	
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D are two SCCP phones Outbound Roll Over Mode - "Roll Over to any Line" Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize C calls A,A answers SetupConference on A.
Expected Results	Call on A will go to OnHold State. New call will be created on Line B. Application then has to complete Conference using Join Across Lines feature.

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll over Mode when User is Added to New User Group

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is added to New User Group	
Description	Testing Transfer/Conference Scenario on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 when User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Outbound Roll Over Mode - "Roll Over to any Line" Max Number of Calls: 2 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. C calls A,A answers SetupTransfer on A.

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is added to New User Group	
Expected Results	Call on A will go to OnHoldPendingTransfer/OnHoldPendingConference. New Consult call will be created on Line A. Application then has to complete Transfer using CompleteTransfer or DTAL feature.
Variants	Test the same Scenario with Conference LineCompleteTransfer with Mode as Conference to complete Conference

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Transfer/Conference Scenario on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 When User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Outbound Roll Over Mode - Roll Over to any Line Max Number of Calls: 2 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. C calls A,A answers SetupTransfer on A.
Expected Results	Call on A will go to OnHoldPendingTransfer/OnHoldPendingConference. New Consult call will be created on Line A. Application then has to complete Transfer using CompleteTransfer or DTAL feature.
Variants	Test the same Scenario with Conference LineCompleteTransfer with Mode as Conference to complete Conference

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Transfer/Conference Scenario on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 when User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Lines A and B are configured with Different DN Outbound Roll Over Mode - Roll Over within same DN Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. C calls A,A answers SetupTransfer on A.
Expected Results	SetupTransfer Request will fail with error "LINEERR_CALLUNAVAIL".
Variants	Test the same Scenario with SetupConference

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Transfer/Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Transfer/Conference Scenario on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 when User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Lines A and B are configured with Different DN Outbound Roll Over Mode - Roll Over within same DN Max Number of Calls: 2 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. C calls A,A answers SetupTransfer on A.
Expected Results	Call on A will go to OnHoldPendingTransfer/Conference State. New Consult call will be created on Line A. Application then has to complete Transfer using CompleteTransfer or DTAL feature.
Variants	Test the same Scenario with SetupConference

LineMakeCall Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

LineMakeCall Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing LineMakeCall Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 when User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Lines A and B are configured with Different DN Outbound Roll Over Mode - Roll Over within same DN" or "Roll Over to Any Line Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. C calls A,A answers LineMakeCall on A.
Expected Results	LineMakeCall Operation will fail with error "LINEERR_CALLUNAVAIL". Roll Over Doesn't Happen to second line as the roll over is only for Outbound Calls.

LineUnPark Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll over Mode when User is Added to New User Group

LineUnPark Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing LineUnPark Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 When User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Lines A and B are configured with Different DN Outbound Roll Over Mode - Roll Over within same DN" or "Roll Over to Any Line Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. C calls A,A answers LineUnPark on A.(tires to retrieve the available Parked Call from Park DN)
Expected Results	LineUnPark Operation will fail with error "LINEERR_CALLUNAVAIL". Roll Over Doesn't Happen to second line as the roll over is only for Outbound Calls.

EM Login/Logout Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

EM Login/Logout Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing EM Log In/Out Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 When User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. EM Profile is logged onto the Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. Test the Use Case from UseCase#1 to UseCase#10
Expected Results	Same as the Use Case tested.

Manual Transfer Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Manual Transfer Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Existing Call Events on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 when User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Outbound Roll Over Mode - Roll Over to any Line Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. Step 1: From Phone C call A Step 2: Answer the Call on A Step 3: Press Transfer Button on Cisco Unified IP Phone 6900 Series and Dial D. Step 4: Answer the Call on D Step 5: Complete Transfer from Phone A Variant: Monitor Phones after Transfer is completed from Phone.
Expected Results	Step 4: Call on Line A will be in OnHold State. Call on Line B will be in Connected State. **Note** When consult call is created on the same Line; Call will be on ONHOLDPENDINGTRANSFER state. Step 5: Both the calls on A and B will go to IDLE state. C and D will be in Simple Call. Variant: Same as this Use Case

Manual Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Manual Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 behavior When User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Outbound Roll Over Mode - "Roll Over to any Line" Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. Step 1: From Phone C call A Step 2: Answer the Call on A Step 3: Press conference Button on Cisco Unified IP Phone 6900 Series and Dial D. Step 4: Answer the Call on D Step 5: Complete Conference from Phone Variant: Monitor Phones after Conference is completed from Phone.
Expected Results	Step 4: Call on Line A will be in OnHold State. Call on Line B will be in Connected State. **Note** When consult call is created on the same Line; Conference Model is created as today on Non-Cisco Unified IP Phone 6900 Series. Step 5: A ,C and D will be in conference Conference model will be created on Line A. Variant: Same as this Use Case.

Manual Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Manual Conference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 behavior When User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Outbound Roll Over Mode - "Roll Over to any Line" Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. Step 1: From Phone C call A Step 2: Answer the Call on A Step 3: Press conference Button on Cisco Unified IP Phone 6900 Series Phone and Dial D. Step 4: Answer the Call on D Step 5: Complete Conference from Phone Variant: Monitor Phones after Conference is completed from Phone.
Expected Results	Step 4: Call on Line A will be in OnHold State. Call on Line B will be in Connected State. **Note** When consult call is created on the same Line; Conference Model is created as today on Non-Cisco Unified IP Phone 6900 Series Phone. Step 5: A ,C and D will be in conference Conference model will be created on Line A. Variant: Same as this Use Case.

SetupConference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

SetupConference Operation on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 behavior When User is added to New User Group and different Roll Over Mode.
Test Setup	User is added to New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode C, D is two SCCP phones. Outbound Roll Over Mode - "Roll Over to any Line" Max Number of Calls: 1 Busy Trigger : 1 Application does Line Initialize; Application opens all the lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. C calls A,A answers Step 1: SetupTransfer on A. Step 2: Complete Conference From Phone.
Expected Results	Step 1: Call on Line A will be in OnHold State. Call on Line B will be in Connected State. Step 5: A ,C and D will be in conference Conference model will be created on Line A.

BWC on Cisco Unified IP Phone 7931 in Non Roll Over Mode when User is Removed from New User Group

BWC on Cisco Unified IP Phone 7931 in Non Roll Over Mode When User is removed from New User Group	
Description	Testing Cisco Unified IP Phone 7931 Phone behavior in Non Roll Over Mode When User is removed from New User Group.

BWC on Cisco Unified IP Phone 7931 in Non Roll Over Mode When User is removed from New User Group	
Test Setup	User is Removed from New User Group. A,B are two lines on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Non-Roll Over Mode C, D is two SCCP phones. Outbound Roll Over Mode - "Non Roll Over Mode" Max Number of Calls: 1 Busy Trigger: 1 Application does Line Initialize
Expected Results	Lines on the Cisco Unified IP Phone 7931 will be enumerated. Application would be able to Open Cisco Unified IP Phone 7931 with Non-Roll Over Mode and it would be able to control and perform call operations on Phone.

Acquire Device on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode when User is Added to New User Group

Acquire Device on Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode When User is added to New User Group	
Description	Testing Behavior of Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 on Super Provider when User is added to new user Group.
Test Setup	A - Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 with Roll Over Mode User is Added to New User Group. Step 1: Application does Line Initialize Step 2: LineDevSpecific to Acquire Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931. Step 3: User is removed from New User Group.
Expected Results	Step 2: Application will be receiving PHONE_CREATE and LINE_CREATE events for the Device and lines on that Cisco Unified IP Phone 6900 Series/Cisco Unified IP Phone 7931 in Roll Over Mode. Step 3: Application will be receiving LINE_REMOVE and PHONE_REMOVE for the Cisco Unified IP Phone 7931 and Application will no longer be able to monitor or control that device.

APPENDIX B

Cisco Unified TAPI Interfaces

This appendix contains a listing of APIs that are supported and not supported.

Cisco Unified TAPI Version 2.1 Interfaces

Core Package

Table B-1 lists each TAPI interface

Table B-1 Compliance to TAPI 2.1

API/Message/Structure	Cisco TAPI Support	Comments
TAPI Line Functions		
lineAccept	Yes	
lineAddProvider	Yes	
lineAddToConference	Yes	
lineAnswer	Yes	
lineBlindTransfer	Yes	
lineCallbackFunc	Yes	
lineClose	Yes	
lineCompleteCall	No	
lineCompleteTransfer	Yes	
lineConfigDialog	No	
lineConfigDialogEdit	No	
lineConfigProvider	Yes	
lineDeallocateCall	Yes	
lineDevSpecific	Yes	
lineDevSpecificFeature	Yes	
lineDial	Yes	

Table B-1 Compliance to TAPI 2.1 (continued)

API/Message/Structure	Cisco TAPI Support	Comments
lineDrop	Yes	
lineForward	Yes	
lineGatherDigits	No	
lineGenerateDigits	Yes	
lineGenerateTone	Yes	
lineGetAddressCaps	Yes	
lineGetAddressID	Yes	
lineGetAddressStatus	Yes	
lineGetAppPriority	No	
lineGetCallInfo	Yes	
lineGetCallStatus	Yes	
lineGetConfRelatedCalls	Yes	
lineGetCountry	No	
lineGetDevCaps	Yes	
lineGetDevConfig	No	
lineGetIcon	No	
lineGetID	Yes	
lineGetLineDevStatus	Yes	
lineGetMessage	Yes	
lineGetNewCalls	Yes	
lineGetNumRings	Yes	
lineGetProviderList	Yes	
lineGetRequest	Yes	
lineGetStatusMessages	Yes	
lineGetTranslateCaps	Yes	
lineHandoff	Yes	
lineHold	Yes	
lineInitialize	Yes	
lineInitializeEx	Yes	
lineMakeCall	Yes	
lineMonitorDigits	Yes	
lineMonitorMedia	No	
lineMonitorTones	Yes	
lineNegotiateAPIVersion	Yes	
lineNegotiateExtVersion	Yes	

Table B-1 Compliance to TAPI 2.1 (continued)

API/Message/Structure	Cisco TAPI Support	Comments
lineOpen	Yes	
linePark	Yes	
linePickup	No	
linePrepareAddToConference	Yes	
lineRedirect	Yes	
lineRegisterRequestRecipient	Yes	
lineReleaseUserUserInfo	No	
lineRemoveFromConference	No	
lineRemoveProvider	Yes	
lineSecureCall	No	
lineSendUserUserInfo	No	
lineSetAppPriority	Yes	
lineSetAppSpecific	No	
lineSetCallData	No	
lineSetCallParams	No	
lineSetCallPrivilege	Yes	
lineSetCallQualityOfService	No	
lineSetCallTreatment	No	
lineSetCurrentLocation	No	
lineSetDevConfig	No	
lineSetLineDevStatus	No	
lineSetMediaControl	No	
lineSetMediaMode	No	
lineSetNumRings	Yes	
lineSetStatusMessages	Yes	
lineSetTerminal	No	
lineSetTollList	Yes	
lineSetupConference	Yes	
lineSetupTransfer	Yes	
lineShutdown	Yes	
lineSwapHold	No	
lineTranslateAddress	Yes	
lineTranslateDialog	Yes	
lineUncompleteCall	No	
lineUnhold	Yes	

Table B-1 Compliance to TAPI 2.1 (continued)

API/Message/Structure	Cisco TAPI Support	Comments
lineUnpark	Yes	
TAPI Line Messages		
LINE_ADDRESSSTATE	Yes	
LINE_APPNEWCALL	Yes	
LINE_CALLINFO	Yes	
LINE_CALLSTATE	Yes	
LINE_CLOSE	Yes	
LINE_CREATE	Yes	
LINE_DEVSPECIFIC	Yes	
LINE_DEVSPECIFICFEATURE	Yes	
LINE_GATHERDIGITS	Yes	
LINE_GENERATE	Yes	
LINE_LINEDEVSTATE	Yes	
LINE_MONITORDIGITS	Yes	
LINE_MONITORMEDIA	No	
LINE_MONITORTONE	Yes	
LINE_REMOVE	Yes	
LINE_REPLY	Yes	
LINE_REQUEST	Yes	
TAPI Line Structures		
LINEADDRESSCAPS	Yes	
LINEADDRESSSTATUS	Yes	
LINEAPPINFO	Yes	
LINECALLINFO	Yes	
LINECALLLIST	Yes	
LINECALLPARAMS	Yes	
LINECALLSTATUS	Yes	
LINECALLTREATMENTENTRY	No	
LINECARDENTRY	Yes	
LINECOUNTRYENTRY	Yes	
LINECOUNTRYLIST	Yes	
LINEDEVCAPS	Yes	
LINEDEVSTATUS	Yes	
LINEDIALPARAMS	No	
LINEEXTENSIONID	Yes	

Table B-1 Compliance to TAPI 2.1 (continued)

API/Message/Structure	Cisco TAPI Support	Comments
LINEFORWARD	Yes	
LINEFORWARDLIST	Yes	
LINEGENERATETONE	Yes	
LINEINITIALIZEEXPARAMS	Yes	
LINELOCATIONENTRY	Yes	
LINEMEDIACONTROLCALLSTATE	No	
LINEMEDIACONTROLDIGIT	No	
LINEMEDIACONTROLMEDIA	No	
LINEMEDIACONTROLTONE	No	
LINEMESSAGE	Yes	
LINEMONITORTONE	Yes	
LINEPROVIDERENTRY	Yes	
LINEPROVIDERLIST	Yes	
LINEREQMEDIACALL	No	
LINEREQMAKECALL	Yes	
LINETERMCAPS	No	
LINETRANSLATECAPS	Yes	
LINETRANSLATEOUTPUT	Yes	
TAPI Phone Functions		
phoneCallbackFunc	Yes	
phoneClose	Yes	
phoneConfigDialog	No	
phoneDevSpecific	Yes	
phoneGetButtonInfo	No	
phoneGetData	No	
phoneGetDevCaps	Yes	
phoneGetDisplay	Yes	
phoneGetGain	No	
phoneGetHookSwitch	No	
phoneGetIcon	No	
phoneGetID	No	
phoneGetLamp	No	
phoneGetMessage	Yes	
phoneGetRing	Yes	
phoneGetStatus	No	

Table B-1 Compliance to TAPI 2.1 (continued)

API/Message/Structure	Cisco TAPI Support	Comments
phoneGetStatusMessages	Yes	
phoneGetVolume	No	
phoneInitialize	Yes	
phoneInitializeEx	Yes	
phoneNegotiateAPIVersion	Yes	
phoneNegotiateExtVersion	No	
phoneOpen	Yes	
phoneSetButtonInfo	No	
phoneSetData	No	
phoneSetDisplay	Yes	
phoneSetGain	No	
phoneSetHookSwitch	No	
phoneSetLamp	No	
phoneSetRing	No	
phoneSetStatusMessages	Yes	
phoneSetVolume	No	
phoneShutdown	Yes	
TAPI Phone Messages		
PHONE_BUTTON	Yes	
PHONE_CLOSE	Yes	
PHONE_CREATE	Yes	
PHONE_DEVSPECIFIC	No	
PHONE_REMOVE	Yes	
PHONE_REPLY	Yes	
PHONE_STATE	Yes	
TAPI Phone Structures		
PHONEBUTTONINFO	No	
PHONECAPS	Yes	
PHONEEXTENSIONID	No	
PHONEINITIALIZEEXPARAMS	Yes	
PHONEMESSAGE	Yes	
PHONESTATUS	No	
VARSTRING	Yes	

Table B-1 Compliance to TAPI 2.1 (continued)

API/Message/Structure	Cisco TAPI Support	Comments
TAPI Assisted Telephony Functions		
tapiRequestDrop	No	
tapiRequestMediaCall	No	
TAPI Call Center Functions		
lineAgentSpecific	No	
lineGetAgentActivityList	No	
lineGetAgentCaps	No	
lineGetAgentGroupList	No	
lineGetAgentStatus	No	
lineProxyMessage	No	
lineProxyResponse	No	
lineSetAgentActivity	No	
lineSetAgentGroup	No	
lineSetAgentState	No	
TAPI Call Center Messages		
LINE_AGENTSPECIFIC	No	
LINE_AGENTSTATUS	No	
LINE_PROXYREQUEST	No	
TAPI Call Center Structures		
LINEAGENTACTIVITYENTRY	No	
LINEAGENTACTIVITYLIST	No	
LINEAGENTCAPS	No	
LINEAGENTGROUPENTRY	No	
LINEAGENTGROUPLIST	No	
LINEAGENTSTATUS	No	
LINEPROXYREQUEST	No	
Wave Functions		
waveInAddBuffer	Yes	
waveInClose	Yes	
waveInGetDevCaps	No	
waveInGetErrorText	No	
waveInGetID	Yes	
waveInGetNumDevs	No	
waveInGetPosition	Yes	
waveInMessage	No	

Table B-1 Compliance to TAPI 2.1 (continued)

API/Message/Structure	Cisco TAPI Support	Comments
waveInOpen	Yes	
waveInPrepareHeader	Yes	
waveInProc	No	
waveInReset	Yes	
waveInStart	Yes	
waveInStop	No	
waveInUnprepareHeader	Yes	
waveOutBreakLoop	No	
waveOutClose	Yes	
waveOutGetDevCaps	Yes	
waveOutGetErrorText	No	
waveOutGetID	Yes	
waveOutGetNumDevs	No	
waveOutGetPitch	No	
waveOutGetPlaybackRate	No	
waveOutGetPosition	No	
waveOutGetVolume	No	
waveOutMessage	No	
waveOutOpen	Yes	
waveOutPause	No	
waveOutPrepareHeader	Yes	
waveOutProc	No	
waveOutReset	Yes	
waveOutRestart	No	
waveOutSetPitch	No	
waveOutSetPlaybackRate	No	
waveOutSetVolume	No	
waveOutUnprepareHeader	Yes	
waveOutWrite	Yes	

APPENDIX C

Troubleshooting Cisco Unified TAPI

This appendix contains information about troubleshooting Cisco Unified Communication manager. It contains the following sections:

Cisco TSP 3.1 Installation Issues

When you are upgrading a system to Cisco TSP 3.1 in which Cisco TSP 3.0 is installed, run the Cisco TSP 3.1 installation on the Cisco TSP 3.0 system to perform the upgrade. If you are installing Cisco TSP 3.0 on a system in which Cisco TSP 3.1 is installed, you must first uninstall Cisco TSP 3.1 by using the Cisco TSP 3.1 installation and then run the Cisco TSP 3.0 installation. If you try to run the Cisco TSP 3.0 install on a system in which CiscoTSP 3.1 is already installed, you will experience significant problems.

If you accidently install Cisco TSP 3.0 on top of Cisco TSP 3.1 then you must perform the following steps to clean up the install for both Cisco TSP 3.0 and Cisco TSP 3.1.

Windows NT/95/98

Step 1 Go to **ControlPanel\Telephony** and remove all Cisco TSP entries in the provider list.

Step 2 Click to **Telephony Drivers** tab and select all the Cisco TSP entries and remove them from provider list.

Windows 2000

Step 1 Go to **ControlPanel\Phone & Modem Options**

Step 2 Click to **Advanced tab**, select all CiscoTSP entries and remove them from provider list.

Common to All Platforms

Step 1 Go to registry **HKEY_LOCAL_MACHINE\SOFTWARE\Cisco Systems, Inc.** and delete Cisco TSP registry key with its subkeys.

Step 2 Delete all **CiscoTSP*.tsp** and **CiscoTUISP*.dll** from winnt\system32 directory. You may need to reboot the system, so it will allow you to remove these files.

Step 3 Go to **registry** key **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall** and Delete sub keys {AF198881-AF5B-11D4-9DA2-000039ED6324} and CiscoTSP.

> **Note** Do not delete the entire Uinstall key, just delete the **{AF198881-AF5B-11D4-9DA2-000039ED6324}** and Cisco TSP key.

You have successfully removed all the Cisco TSP from the system. You can now safely install either Cisco TSP 3.0 or Cisco TSP 3.1 onto the system.

Cisco TSP Configuration in Windows

With Cisco TSP 3.1, you must configure all the installed TSPs in the following steps.

Windows 95/98/NT

Step 1 From Control Panel, execute the “Telephony” utility.

Step 2 Click the **Telephony Drivers** tab and look in the list of service providers for CiscoTSP 0xx.

Step 3 Highlight this entry and click the **Configure** button. The Cisco TSP configuration window should display.

Windows 2000

Step 1 From Control Panel execute, the **Phone and Modem Options** utility.

Step 2 Click the **Advanced** tab and look in the list of service providers for Cisco TSP 0xx.

Step 3 Highlight this entry and click the **Configure** button. The Cisco TSP configuration window should display.

Windows NT/95/98

Step 1 Click **User** tab. The security fields comprise the Username and Password. To change the user name that is stored in the registry, overwrite the name in the user name edit box.

Step 2 Enter the new password and confirm the password in the **verify password** edit box. The passwords must match. This user and password must get configured in the Cisco Unified Call Manager user administration for authentication to pass. Only one user name and password can remain active at a time.

Step 3 Click **CTI Manager** tab. The Cisco Unified Call Manager location information gets entered in this dialog box. If the TSP is on the same machine as the Unified Call Manager, click the **Local Machine** radio button. If the Unified Call Manager is on a different machine, then click the **Call Manager IP Address** radio button and enter the IP address or click the **Call Manager Name** radio button and enter the host name.

Step 4 Click **Wave** tab. Select the number of wave devices that this TSP will use. A limit exists on the maximum number of wave devices that can be installed on a system. Users must have to choose the number of wave devices through the available numbers.

Step 5 Click **Trace** tab. Refer to **Turning on tracing for the TAPI Service Provider** for details on this.

Step 6 Click **Advanced** tab. The Synchronous Message Timeout specifies the time that the TSP should wait to receive a response to a CTI synchronous message. The value gets expressed in milliseconds, and the default is 15000 ms.

> **Note** *TAPI User's Guide* describes the rest of the fields in the TSP configuration dialog box.

Step 7 Restart the telephony service after configuring the TSP, so that an application can run and get the devices.

> **Note** To Add/delete voice lines from the TSP configuration, you must first uninstall (using instructions in "Uninstalling Wave Driver" section) and then add/delete the voice lines and then install wave driver (using instructions in "Installing the Wave Drivers" section) if wave driver has been already installed on the system. If no wave driver is installed on the system, add the devices and then install the wave driver by using instructions in the "Installing the Wave Drivers" section.

Wave Driver Installation in Windows

Windows NT

Step 1 From Control Panel, execute the **Multimedia** utility.

Step 2 Click the **Devices** tab. Highlight **Audio Devices** and click **Add**.

Step 3 Select **Unlisted or Updated Driver** and click **OK**.

Step 4 On the **Install Driver** window click **Browse** and browse to the C:\Program Files\Cisco\Wave Drivers directory and click **OK**.

Step 5 In the **Install Driver** window, click **OK** again

Step 6 Select **Cisco TAPI Wave Driver** in the Add Unlisted or Updated Driver window and click **OK**. When you are prompted to reboot your machine, do so.

For Windows 2000

Step 1 From Control Panel, execute the **Add/Remove Hardware** utility.

Step 2 Select **Add/Troubleshoot** when the prompt – **Choose a Hardware Task** displays.

Step 3 Click the **Next** button.

Step 4 Select **Add a New Device** when the prompt – **Choose a Hardware Device** displays.

Step 5 Click the **Next** button.

Step 6 Select **no** when the question **Do you want to search for new hardware devices?** displays.

Step 7 Click the **Next** button. Select **Sound, video and game controller** when you are prompted for hardware type.

Step 8 Click the **Next** button. Click the **Have Disk** button when you are prompted to **Select a Device Driver**.

Step 9 Click the **Browse** button on the **Install from Disk** window. Browse to C:\Program Files\Cisco\Wave Drivers and select the – file OEMSETUP.

Step 10 To install the Cisco Wave Driver, click **Open**. When you prompted for **Install from disk 1** for file avaudio32.dll, choose **Browse** button and select path – C:\Program Files\Cisco\Wave Drivers and click **Open** to install the avaudio32.dll. When you are prompted to reboot your machine, do so.

Note Be aware that the Cisco Wave Driver is not supported on Windows 95 or Windows 98.

Wave Driver Uninstallation in Windows

Windows NT

Step 1 From Control Panel, execute the **Multimedia** utility.

Step 2 Click the **Devices** tab.

Step 3 From **Audio Devices** highlight **Audio for Cisco TAPI Wave Driver** and select **Remove**. It will prompt you with **Are you sure, you want to remove Cisco TAPI Wave Driver?** choose **Yes**.

Windows 2000

Step 1 From Control Panel, execute the **Add/Remove Hardware** utility.

Step 2 Select **Uninstall** when the prompt – **Choose a Hardware Task** displays.

Step 3 Click **Next** button.

Step 4 From **Add/Remove Hardware Wizard** choose the device to be removed from the list that is displayed.

Step 5 Select **Cisco TAPI Wave Driver** device. When prompted to uninstall, select **Yes, want to uninstall**. Device gets uninstalled. Click on **Finish** button to complete uninstall. Restart Windows.

TSP Trace of Internal Messages

Procedure

Step 1 Choose **Start > Settings > Control Panel** and select **Phone and Modem Options**.

Step 2 Click **Advanced** tab and select the **CiscoTSP 0xx** and click **Configure** button.

Step 3 Click **Trace** tab. Select **Trace On** check box and select **1**. **TSP Trace** to trace the TSP internal messages. Select **Error** to just log errors in the TSP Select **Detailed** to log internal messages for debugging purposes. Select **2. CTI Trace** to trace the messages sent between CTI and TSP. Select **3. TSPI Trace** to trace the requests and events that are sent between TSP and TAPI.

Step 4 Set up a Directory that is the path for the trace log. For example, c:\Temp No. of Files: Setting this to a value greater than or equal to 1 enables rolling log files. For example, a value of 10 will cause up to 10 log files to be used in a cyclic fashion. Max lines/file: specifies the maximum number of trace statements that will be written to each log file. For example, a value of 1000 will cause up to 1000 trace statements to be written to each log file.

CTI Ports and Cisco Unified Communications Manager Administration

To add a CTI port, use the Cisco Unified Communications Manager Administration window and follow these steps:

Procedure

Step 1 Choose **Device > Phone**. A window with the title **Find and List Phones** will display.

Step 2 On the right, click the link **Add a New Phone**. This will display a page titled **Add a Phone**.

Step 3 Select **CTI Port** from the drop down list of devices and click the **Next** button. Enter the device name of the CTI port and tab to the next **Description** box. The name will display in the description field automatically. You can edit this to be something else or leave it as is.

Step 4 Click the **Insert** button. To add a line to this CTI port, click the line number on the left and enter a directory number.

Route Points and Cisco Unified Communications Manager Administration

To add a route point, use the Cisco Unified Communications Manager Administration window and follow these steps:

Procedure

Step 1 Choose **Device > CTI Route Point**. The **Find and List CTI Route Points** window is displayed.

Step 2 Click **Add a New CTI Route Point**. The **CTI Route Point Configuration** window is displayed.

Step 3 Enter the device name of the CTI route point and tab to the **Description** edit box. The name is displayed in the description field automatically. Edit the Description field or click the **Insert** button.

TSP Operation Verification

To verify the TSP operation on the machine where the TSP is installed, use the Microsoft Windows Phone Dialer Application. Find this application in the C:\Program Files\Windows NT directory under the name dialer.exe. When the program is run, a dialog box displays that asks which line and address the user wants to use to connect. If there are no lines in the Line drop down list, then a problem may exist between the TSP and the Cisco Unified Communications Manager. If lines are available, choose one of the lines, keep the Address set to zero (0) and click **OK**. Enter a Number to dial, and a call should be placed to that number. If call is successful, you know that the TSP is operational on the machine where the TSP is installed. If problems are encountered with installation and setup of Remote TSP, this test represents a good way to verify whether the TSP is operating properly and that the problem is with the configuration and setup of Remote TSP.

Version Compatibility

Cisco recommends that the TSP client should always use the plug-in that is downloaded from corresponding Cisco Unified Communications Manager server.

Cisco TSP Readme

The Cisco Unified Communications Manager TSP readme file is copied to the client PC when TSP plug-in is installed.

Common Issues

Table C-1 describes common issues and the work around.

Table C-1 Common Issues and Solutions

Issue	Symptom	TSP Version	UCM Release	Solution	Reason
Cannot configure TSP	TSP configuration button stays grayed out.	5.1(1.6)	5.1(3)	Copy Cisco version of libeay32.dll and ssleay32.dll.	Cisco version of libeay32.dll and ssleay32.dll gets overwritten or old dlls are locked by Windows during TSP installation.
TSP upgrade issue	TSP auto install does not work. User cannot run TSP auto upgrade from 4.1 to 6.x.	4.1	4.1	1) Manually run silent upgrade from Client PC CiscoTsp.exe /s /v" /qn". 2) Modify TspAutoInstall.exe to support silent upgrade from 4.1 to 6.x.	Silent install parameter in 6.x TSP changed due to IS 12 upgrade. Corresponding pre-6.x release also needs to be changed to support auto upgrade.

Table C-1 Common Issues and Solutions

Issue	Symptom	TSP Version	UCM Release	Solution	Reason
TSP crash with EM operation	Cisco TSP announced an error code of 0x80000050 (LINEERR_UNINITIALIZED). When EM user logs in, TSP sends line create to application. Application sends LineOpen and PhoneOpen at the same time.	5.1	5.1.2	Apply ES that contains fix.	The buffer overflow on qbehelper caused the TSP crash.
No EM device reported	When EM logs in, application does not see the device information about the EM device	5.1(1.6)	5.1(3)	Apply ES that has fix.	The problem is caused by race condition in CiscoTSP. Where Line is being initialized because of lineOpen and in the same time application sends LineClose immediately. During LineOpen process, TSP will send request to CTIManager to query Line info, and wait for the synchronized response. However, at CTIManager, while processing the line info request, it also almost at the same time receives subsequent LineClose request. Due to LineClose request, CTIManager cancels/ignores the processing of the line info request. But TSP side is still waiting for the response. This explains the reason that the TSP worker thread got stuck.

APPENDIX D

Cisco Unified TAPI Operations-by-Release

The following tables list new, changed, and "under consideration or review" features for Cisco Unified TAPI by Cisco Unified Communications Manager release.

Table Legend

✔	Supported
✖	Not Supported
ℹ	Modified
UCR	Under Consideration or Review

Table D-1 API Interfaces

TSP Features	3.1	3.2	3.3	4.0	4.1	4.2	4.3	5.0	5.1	6.0	6.1	7.0	7.1.2	7.1.3 (UCR)	8.0 (UCR)
CTI Manager and Support for fault tolerance	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Support for Cisco CallManager Extension Mobility	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Table D-1 API Interfaces (continued)

Support for Multiple CiscoTSP	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
(Redirect Support for) Blind Transfer	✔	✔	✔	ⓘ	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Support for swap hold and setup transfer with the lineDevSpecific() function	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Support for lineForward()	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Support to Reset the Original Called Party upon Redirect with the lineDevSpecific function	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Support to Set the Original Called Party upon Redirect with the lineDevSpecific function	✖	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Line In-Service or Out-of-Service	ⓘ	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Support for multiple languages in the CiscoTSP installation program and in the CiscoTSP configuration dialogs	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
User Deletion from Directory	✖	✖	ⓘ	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Opening Two Lines on One CTI Port Device	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Support for linePark and lineUnpark	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Support for monitoring Call Park Directory Numbers using lineOpen	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Call Reason Enhancements	✖	✖	ⓘ	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Device Data Passthrough	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
CiscoTSP Auto Update	✖	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Multiple Calls per Line Appearance	✖	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Shared Line Appearance	✖	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Select Calls	✖	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Transfer Changes	✖	✖	✖	ⓘ	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Table D-1 API Interfaces (continued)

Direct Transfer	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Conference Changes	✗	✗	✗	ⓘ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Join	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Privacy Release	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Barge and cBarge	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Dynamic Port Registration	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Media Termination at Route Points	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
QoS support	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Support for Presentation Indication	✗	✗	✗	✓	✓	✓	✓	✓	ⓘ	✓	✓	✓	✓	✓	✓
Windows 2003 Support	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Unicode Support	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
SRTP support	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Partition Support	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
SuperProvider Functionality	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Security (TLS) support	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
FAC/CMC Support	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CTI Port Third Party Monitoring	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Alternate Script Support	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
SIP Features Refer/Replaces	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
SIP URI	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
SIP phone support	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Change Notification of SupetProvider and CallParkDN Monitoring flags	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
3XX	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
Intercom Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Secure Conferencing Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Monitoring & Recording	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓

Table D-1 API Interfaces (continued)

Arabic and Hebrew Language Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Do-Not-Disturb Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Conference Enhancement	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓
Join AcrossLine (SCCP)	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓
Join AcrossLine (SIP)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
Locale Infrastructure Enhancement	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
Do-Not-Disturb Rejection	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
Call Party Normalization	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
Click-To-Conference	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
IPv6 Support on Linux	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Windows Vista Support	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓
Enhaced MWI	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Direct Transfer Across Lines	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Support for > 100DNs	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Swap/Cancel support on RoundTable phone	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Drop Any Party	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Park Reversion	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Conditional Reset	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Logical Partition	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Assisted DPark	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
RT_Lite Phone Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Device State Server	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Exposing Busy Trigger / Line Number / Voice Mail Pilot / Line Label / New call outbound rollover/ Consult call rollover/JAL/DTAL flag and IP address (IPv4 & IPv6) of the device	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	ⓘ

Table D-1 API Interfaces (continued)

Hunt List Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Call Intercept Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Entension Mobility Cross Cluster (EMCC) Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Call Pickup Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
End-To-End Call Tracing	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Secure Monitoring Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ViPR support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
iSAC Codec Support	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Exposing Meeting Place Bridge Participant ID (CIA 1733)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Table D-2 TAPI Line Functions

TAPI Line Functions	3.1	3.2	3.3	4.0	4.1	4.2	4.3	5.0	5.1	6.0	6.1	7.0	7.1.2	7.1.3 (UCR)	8.0 (UCR)
LineAddToConference	✓	✓	✓	ⓘ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LineCompleteTransfer	✓	✓	✓	ⓘ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LineDevSpecific	ⓘ	✓	✓	ⓘ	ⓘ	✓	✓	✓	ⓘ	ⓘ	✓	ⓘ	ⓘ	✓	ⓘ
LineForward	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LinePark	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LineUnpark	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LineRedirect	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LineBlindTransfer	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LineDevSpecificFeature	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
LineRemoveFromConference	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓

Table D-3 TAPI Line Messages

TAPI Line Messages	3.1	3.2	3.3	4.0	4.1	4.2	4.3	5.0	5.1	6.0	6.1	7.0	7.1.2	7.1.3 (UCR)	8.0 (UCR)
LINE_ADDRESSSTATE	ⓘ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table D-3 TAPI Line Messages (continued)

LINE_CALLINFO	ⓘ	✓	✓	✓	✓	✓	✓	ⓘ	ⓘ	ⓘ	ⓘ	ⓘ	ⓘ	✓	✓
LINE_CALLSTATE	✓	✓	✓	ⓘ	ⓘ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LINE_REMOVE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LINE_DEVSPECIFIC	✓	✓	✓	✓	ⓘ	✓	✓	✓	ⓘ	ⓘ	✓	✓	✓	ⓘ	ⓘ
LINE_DEVSPECIFICFEATURE	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
LINE_CALLDEVSPECIFIC	✓	✓	✓	✓	ⓘ	✓	✓	✓	ⓘ	ⓘ	✓	✓	ⓘ	✓	ⓘ

Table D-4 TAPI Line Structures

TAPI Line Structures	**3.1**	**3.2**	**3.3**	**4.0**	**4.1**	**4.2**	**4.3**	**5.0**	**5.1**	**6.0**	**6.1**	**7.0**	**7.1.2**	**7.1.3 (UCR)**	**8.0 (UCR)**
LINEADDRESSCAPS	ⓘ	✓	✓	ⓘ	ⓘ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LINECALLSTATUS	✓	✓	✓	ⓘ	ⓘ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LINEFORWARD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LINEFORWARDLIST	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LINEDEVCAPS	✓	✓	ⓘ	✓	✓	✓	✓	ⓘ	ⓘ	ⓘ	✓	✓	ⓘ	ⓘ	ⓘ
LINEDEVSTATUS	✓	✓	✓	✓	✓	✓	✓	ⓘ	✓	ⓘ	✓	✓	✓	✓	✓
LINECALLINFO	✓	✓	✓	✓	✓	✓	✓	✓	ⓘ	ⓘ	ⓘ	ⓘ	ⓘ	ⓘ	ⓘ

Table D-5 TAPI Phone Functions

TAPI Phone Functions	**3.1**	**3.2**	**3.3**	**4.0**	**4.1**	**4.2**	**4.3**	**5.0**	**5.1**	**6.0**	**6.1**	**7.0**	**7.1.2**	**7.1.3 (UCR)**	**8.0 (UCR)**
PhoneDevSpecific	✗	✗	✓	✓	✓	✓	✓	ⓘ	✓	✓	✓	✓	✓	✓	✓
PhoneGetStatus	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table D-6 TAPI Phone Messages

TAPI Phone Messages	**3.1**	**3.2**	**3.3**	**4.0**	**4.1**	**4.2**	**4.3**	**5.0**	**5.1**	**5.2**	**6.0**	**6.1**	**7.0**	**7.1.2**	**7.1.3 (UCR)**	**8.0 (UCR)**
PHONE_REMOVE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table D-7 TAPI Phone Structures

TAPI Phone Structures	3.1	3.2	3.3	4.0	4.1	4.2	4.3	5.0	5.1	5.2	6.0	6.1	7.0	7.1.2	7.1.3 (UCR)	8.0 (UCR)
PHONECAPS	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	ℹ	✔
PHONESTATUS	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

APPENDIX E

CTI Supported Devices

Table E-1 provides information about CTI supported devices.

Table legend:

- ✓—Supported
- ✗—Not supported

Table E-1 CTI Supported Device Matrix

Device/Phone Model	SCCP	SIP	Comments
Analog Phone	✓	✗	
Cisco 12 S	✓	✗	
Cisco 12 SP	✓	✗	
Cisco 12 SP	✓	✗	
Cisco 30 SP	✓	✗	
Cisco 30 VIP	✓	✗	
Cisco 3911	✗	✗	
Cisco 7902	✓	✗	
Cisco 7905	✓	✗	
Cisco 7906	✓	✓	
Cisco 7910	✓	✗	
Cisco 7911	✓	✓	
Cisco 7912	✓	✗	
Cisco 7914 Sidecar	✓	✗	
Cisco 7915 Sidecar	—	—	Not yet tested
Cisco 7916 Sidecar	—	—	Not yet tested

Table E-1 CTI Supported Device Matrix (continued)

Device/Phone Model	SCCP	SIP	Comments
Cisco 7920	✓	✗	
Cisco 7921	✓	✗	
Cisco 7931	✓	✗	CTI supported only if rollover is disabled
Cisco 7935	✓	✗	
Cisco 7936	✓	✗	
Cisco 7937	✓	✗	
Cisco 7940	✓	✗	
Cisco 7941	✓	✓	
Cisco 7941G-GE	✓	✓	
Cisco 7942	✓	✓	
Cisco 7945	✓	✓	
Cisco 7960	✓	✗	
Cisco 7961	✓	✓	
Cisco 7961G-GE	✓	✓	
Cisco 7962	✓	✓	
Cisco 7965	✓	✓	
Cisco 7970	✓	✓	
Cisco 7971	✓	✓	
Cisco 7975	✓	✓	
Cisco 7985	✓	✗	
Cisco ATA	✓	✗	
Cisco IP Communicator	✓	✗	CTI support when running in desktop mode depends on physical device; Softphone mode not yet tested
Cisco Unified Personal Communicator	✗	✗	
Cisco VGC Phone	✓	✗	
VG224	✗	✗	Not a CTI supported device.
VG248	✓	✗	

Table E-1 CTI Supported Device Matrix (continued)

Device/Phone Model	SCCP	SIP	Comments
CTI Port	—	—	CTI supported virtual device that does not use SCCP or SIP
CTI Route Point	—	—	CTI supported virtual device that does not use SCCP or SIP
CTI Route Point (Pilot Point)	—	—	CTI supported virtual device that does not use SCCP or SIP
ISDN BRI Phone	—	—	Not a CTI supported device

INDEX

M

O

P

R

S

T

W

X