Legal Information

Telephony Application Programming Interface (TAPI) Programmer's Reference

Using the *TAPI Programmer's Reference*

The Microsoft Win32 Telephony application programming interface (TAPI) provides services that enable an application developer to add telephone communications to applications developed for operating systems that support the Microsoft® Win32® API, such as Microsoft® Windows NT® Workstation and Microsoft® Windows® 95.

The *TAPI Programmer's Reference* is intended to help experienced programmers learn the API for telephony for these operating systems, and to be used by developers familiar with the Win32 programming environment. Previous development experience with telecommunications or other telephony applications is helpful but not necessary.

Document Overview

This document presents general information about how to develop telephonic applications and offers specific information about the functions, messages, and data types of the Telephony API. The sections include:

- Telephony Overview describes the Win32 Telephony API and explains how to use it in Win32 applications.
- The Telephony Programming Model explains the concepts programmers should know before they begin development of telephonic applications. It includes sections that describe the uses of Telephony's device classes, addresses, and other topics.
- TAPI Applications refers to the source code of a telephonic application (supplied with the Win32 SDK) to illustrate how to create a functional Win32 application.
- Multiple-Application Programming details what developers need to know when programming more than one application to work together or among other applications.
- Call States and Events describes the call states through which a call transitions as it is established or disconnected.
- Supplementary Line Functions describes some of the more advanced functions used to create telephonic applications, including media monitoring, media control, and call conferencing.
- Line Devices Overview explains the line device class and describes the operations that can be carried out on line devices.
- Phone Devices Overview describes the phone device class and the functions TAPI provides to use phone devices.
- Assisted Telephony Overview tells how to add basic telephonic functionality to applications that don't require the flexibility or control provided by the full Telephony API. Readers whose development goals do not include the creation of new applications with a variety of telephonic functions may need to read no further than this section.
- Device Classes describes device classes, the related physical devices or device drivers through which applications send and receive the information or data that makes up a call.
- Quick Function Reference lists the functions of TAPI, a brief description of each, and the nature of each result: synchronous or asynchronous.
- Unicode Support identifies which TAPI functions have Unicode versions and identifies the parameters and structure members that contain Unicode strings. This section also lists functions that do not have Unicode versions.
- Reference contains detailed information about the line device, phone device and assisted telephony functions, messages, structures and constants.

What's New for TAPI Version 2.0

To provide the best performance and support on the Windows NT platform and on future releases of the Windows 95 platform, the Win32 Telephony API and its service providers and supporting components are fully implemented as 32-bit components in Win32. In addition to full 32-bit implementation, Win32 TAPI includes these many new features:

- **Native 32-bit support**. All core TAPI components are Win32, with full support for non-Intel processors (running Windows NT), symmetrical multiprocessing, multithreaded applications, and preemptive multitasking.
- **32-bit application portability**. Existing Win32 full TAPI and assisted TAPI applications which currently run on Windows 95 (using the TAPI 1.4 API) run on Windows NT on the Intel *x*86 family of microprocessors without modification or recompilation.
- **16-bit application portability**. Existing Win16 full TAPI and assisted TAPI applications which currently run on Windows 95 and Windows® 3.1 operating system (using the TAPI 1.3 API) run on Windows NT without modification or recompilation.
- **Unicode support**. Win32 applications can choose to call the existing ANSI TAPI functions or to call Unicode versions of functions that pass or return strings (functions with a "W" suffix).
- **Service processes**. TAPI 2.0 adds mechanisms for notifying applications of telephony events that do not require the application to have a window message queue, thereby enabling background service processes to easily use TAPI services.
- **NDISTAPI compatibility**. The existing support in Windows NT 3.5 for ISDN WAN miniports under Remote Access Service is preserved. NDIS WAN miniport drivers are supported under a kernel mode service provider without modification.
- **Registry support**. All telephony parameters are stored in the registry. Telephony service providers and all stored parameters can be updated across the LAN.
- **Call Center support**. TAPI supports functionality required in a call center environment, including the modeling of predictive dialing ports and queues, ACD agent control, station set status control, and centralized event timing.
- **Quality of Service (QOS) support**. Applications can request, negotiate, and renegotiate quality of service (performance) parameters with the network, and receive indication of QOS on inbound calls and when QOS is changed by the network. The QOS structures are binary-compatible with those used in the Windows Sockets 2.0 specification.
- **Enhanced device sharing**. Applications can restrict handling of inbound calls on a device to a single address, to support features such as distinctive ringing when used to indicate the expected media mode of inbound calls. Applications making outbound calls can set the device configuration when making a call.
- **User mode components**. The full TAPI system, including top-level service provider DLLs, runs in user mode.

The following are additional enhancements to existing TAPI features:

- Applications now receive LINE_APPNEWCALL messages (instead of LINE_CALLSTATE) as the first messages notifying the application of a new call.
- Applications now receive LINE_REMOVE and PHONE_REMOVE messages whenever a line or phone device has been removed from the system.
- LINECONNECTEDMODE_ constants now indicate when a call has been placed in the *onhold* state by the remote party. Also, an additional LINECONNECTEDMODE_ constant indicates to applications when entry into the *connected* state was confirmed by the network, or if it is just being assumed because confirmation from the network is impossible.
- Applications now receive notification that ringing has stopped on a line device by receiving a LINE_LINEDEVSTATE message with the *dwParam1* parameter set to LINEDEVSTATE_RINGING

and both *dwParam2* and *dwParam3* set to zero.

- The **LINEDEVCAPS**, **LINEADDRESSCAPS**, and **PHONECAPS** structures now include a listing of device classes supported by the device, with each supported device class terminated by a zero byte and the final class terminated by two zero bytes. A typical list for a voice modem might be:

  ```
  "tapi/line\0comm\0comm/datamodem\0wave/in\0wave/out\0\0"
  ```

  Applications can scan this list to see if a particular device supports device classes required for the application to properly function.
- The LINEFEATURE_, LINEADDRFEATURE_, and LINECALLFEATURE_ sets of constants have been extended to allow applications to detect when various "flavors" of a function are available for use. For example, applications will be able to detect not only that a call can be transferred, but whether it is permitted to resolve the transfer as a three-way conference.
- Applications can carry out a "one-step transfer" by using LINECALLPARAMFLAGS_ONESTEPTRANSFER with the **lineSetupTransfer** function.
- Applications can carry out a "no hold conference" by using the LINECALLPARAMFLAGS_NOHOLDCONFERENCE option with the **lineSetupConference** function, allowing another device, such as a supervisor or recording device, to be silently attached to the line.
- Applications can carry out a "transfer through hold" (on the systems with this capability) by using the **linePickup** function with a NULL target address. Check for the LINEADDRFEATURE_PICKUPHELD bit in **LINEADDRESSCAPS** and **LINEADDRESSSTATUS** for this capability.
- The **PHONECAPS** structure now includes an indication of which hookswitch states can be *set* for each hookswitch device, and which can be *detected* and *reported*. Previously, applications could detect only the existence of each device without being able to determine which characteristics could be only monitored and not set.
- The **PHONESTATUS** structure now also includes a **dwPhoneFeatures** member that indicates which phone operations can be performed at the particular moment in time that **phoneGetStatus** is called.

New TAPI Functions, Messages, Structures, and Constants for Version 2.0

The Win32 Telephony API version 2.0 includes a number of functions, messages, structures, and constants that are either new (not available in previous versions of TAPI), or that have changed in TAPI version 2.0. The following tables lists these new or changed items.

Functions	New/Changed
lineAgentSpecific	New
lineGetAgentActivityList	New
lineGetAgentCaps	New
lineGetAgentGroupList	New
lineGetAgentStatus	New
lineGetIcon	Changed
lineGetMessage	New
lineInitialize	Obsolete
lineInitializeEx	New
lineOpen	Changed
lineProxyMessage	New
lineProxyResponse	New
lineSetAgentActivity	New
lineSetAgentGroup	New
lineSetAgentState	New
lineSetCallData	New
lineSetCallQualityOfService	New
lineSetCallTreatment	New
lineSetLineDevStatus	New
phoneGetIcon	Changed
phoneGetMessage	New
phoneInitialize	Obsolete
phoneInitializeEx	New

Messages	New/Changed
LINE_AGENTSPECIFIC	New
LINE_AGENTSTATUS	New
LINE_APPNEWCALL	New
LINE_GATHERDIGITS	Changed
LINE_GENERATE	Changed
LINE_MONITORDIGITS	Changed
LINE_MONITORMEDIA	Changed
LINE_MONITORTONE	Changed
LINE_PROXYREQUEST	New
LINE_REMOVE	New
PHONE_REMOVE	New

Structures	New/Changed

LINEADDRESSCAPS	Changed
LINEAGENTACTIVITYENTRY	New
LINEAGENTACTIVITYLIST	New
LINEAGENTCAPS	New
LINEAGENTGROUPENTRY	New
LINEAGENTGROUPLIST	New
LINEAGENTSTATUS	New
LINEAPPINFO	New
LINECALLINFO	Changed
LINECALLPARAMS	Changed
LINECALLSTATUS	Changed
LINECALLTREATMENTENTRY	New
LINEDEVCAPS	Changed
LINEDEVSTATUS	Changed
LINEINITIALIZEEXPARAMS	New
LINEMESSAGE	New
LINEPROXYREQUEST	New
PHONECAPS	Changed
PHONEINITIALIZEEXPARAMS	New
PHONEMESSAGE	New
PHONESTATUS	Changed

Constants	**New/Changed**
LINEADDRCAPFLAGS_ Constants	Changed
LINEADDRFEATURE_ Constants	Changed
LINEAGENTFEATURE_ Constants	New
LINEAGENTSTATE_Constants	New
LINEAGENTSTATUS_Constants	New
LINEBEARERMODE_Constants	Changed
LINECALLFEATURE_Constants	Changed
LINECALLFEATURE2_ Constants	New
LINECALLINFOSTATE_ Constants	Changed
LINECALLPARAMFLAGS_ Constants	Changed
LINECALLREASON_Constants	Changed
LINECALLTREATMENT_ Constants	New
LINECONNECTEDMODE_ Constants	Changed
LINEDISCONNECTMODE_ Constants	Changed
LINEERR_Constants	Changed

LINEFEATURE_ Constants	Changed
LINEINITIALIZEEXOPTION_ Constants	New
LINEOPENOPTION_ Constants	New
LINEPROXYREQUEST_ Constants	New
LINETRANSLATERESULT_ Constants	Changed
LINETSPIOPTION_ Constants	New
PHONEFEATURE_ Constants	New
PHONEINITIALIZEEXOPTION_ Constants	New

Document Conventions for *TAPI Programmer's Reference*

The type conventions used in the *TAPI Programmer's Reference* are as follows:

Bold text	Bold letters indicate terms, such as function and structure names, that you must use exactly as shown.
ALL CAPS	All capitals typically indicate terms, such as message and constant names, that must be used exactly as shown.
Italic text	In introductory and explanatory text, italicized words indicate that a key term or concept is being introduced. In function and message descriptions, italics indicates a placeholder for which you are expected to provide a value or the name of a variable.
`Monospaced text`	Monospaced type indicates code samples and data-structure definitions.

Related Documentation on Telephony Services

Other documentation that may help you understand the Telephony services as they apply to Windows includes:

- *Microsoft Win32 Programmer's Reference*.
- The *Microsoft Win32 Telephony Service Provider Reference*, which provides information on how to write a Win32 Telephony service provider.

Overviews

This section contains overviews on telephony, the telephony programming model, TAPI applications, multiple-application programming, call states and events, and supplementary line functions.

Telephony Overview

Telephony is a technology that integrates computers with the telephone network. With telephony, people can use their computers to take advantage of a wide range of sophisticated communications features and services over a telephone line.

The Telephony application programming interface (TAPI) lets programmers develop applications that provide *personal telephony* to users. TAPI supports both speech and data transmission, allows for a variety of terminal devices, and supports complex connection types and call-management techniques such as conference calls, call waiting, and voice mail. TAPI allows all elements of telephone usage—from the simple dial-and-speak call to international e-mail—to be controlled within applications developed for the Microsoft® Win32® application programming interface.

Using Telephony in Applications

Telephony capabilities help people get the most from telecommunications systems, allowing them to more efficiently manage their voice calls *and* control their data-transfer operations. You can use TAPI to bring this efficiency to any application – database manager, spreadsheet, word-processing application, personal information manager–any application that can benefit by sending and receiving data through the telephone network.

TAPI gives you a consistent set of tools for incorporating these features into your applications:

- Connect directly to the telephone network rather than rely on a separate communications application
- Dial phone numbers automatically
- Transmit documents as files, faxes, or electronic mail
- Access data from news retrieval and other information services
- Set up and manage conference calls
- Receive, store, and sort voice mail
- Use caller-ID to automate the handling of incoming calls
- Control the operations of a remote computer
- Compute collaboratively over telephone lines

TAPI provides your application with access to the telephone network, you provide your users with access to these features. This means you choose and create a user interface that is consistent with the rest of your application. For example, if you use drag and drop extensively, you could let the user send files or faxes through the telephone to a colleague by dragging the icon of the file to an icon representing the colleague's destination. Similarly, you could let the user initiate conference calls by dragging three or four names from an electronic directory into a "Conference box" and clicking a "Connect" command. You choose the interface, and let TAPI carry out the work needed to make and manage the telephone connections.

Telephone Network Services

TAPI provides access to a variety of telephone network services. Although these services may use different technologies to establish calls and transmit voice and data, TAPI makes these service-specific details transparent to applications. This means you can create applications that can take advantage of any available service without including service-specific code in your application.

Historically, most telephone connections in the world have been of the type POTS, or *Plain Old Telephone Service*. Most POTS calls are transmitted digitally except while in the *local loop*—the part of the telephone network between the telephone and the telephone company's central switching office. Within this loop, human speech from a household telephone is usually transmitted in analog format and the digital data from a computer must first be converted to analog by a modem. Digital networks are gradually replacing analog in the local loop.

Using TAPI for POTS is straightforward because POTS is comparatively simple. It normally uses only one type of information (such as data or voice) per call, supports one channel per line, and so on. The vast majority of uses for TAPI are still POTS, and most telephony programmers will use TAPI only for POTS applications.

But TAPI is not restricted to POTS. TAPI also lets you make connections over other types of networks. More advanced kinds of data transmission methods are being developed, refined, and installed. For example, one important digital service is Integrated Services Digital Network (ISDN), which is expected to grow significantly in availability. ISDN networks have these advantages over POTS:

- All digital
- Less prone to error
- Faster data transmission, with speeds up to 128 kilobytes per second (Kbps) on basic service
- From 3 to 32 channels for simultaneous transmission of voice and data
- An international standard

On ISDN networks, error rates are lower than with analog transmission because data travels from one end of the ISDN network to the other in digital format. Speeds of up to 128 Kbps are possible on Basic Rate Interface (BRI-ISDN) standard lines and much higher on Primary Rate Interface (PRI-ISDN) standard lines. By contrast, today's maximum dial-up modem data rates of 28.8 Kbps. When ISDN connections become more widespread, users will be able to send data to the recipient simultaneously with a voice call to that or another person. Each ISDN line, depending on its transmission rate, provides at least three channels (two for voice or data and one strictly for data or signaling information) and as many as 32 channels, for simultaneous, independently operated transmission of voice and data. BRI-ISDN lines provide two 64-Kbps "B" channels (B channels carry voice or data) and one 16-Kbps "D" channel (D channels carry signaling information or packet data). The PRI-ISDN lines for the U.S., Canada, and Japan have twenty-three 64-Kbps B channels and one 64-Kbps D channel. The European PRI standard offers thirty B channels and two D channels.

TAPI can also be used with other digital networks such as T1/E1 and Switched 56 service. With Switched 56, some local and long-distance telephone companies provide signaling at 56 Kbps over dial-up telephone lines. Switched 56 is quickly becoming available throughout the U.S. and in many other countries. It requires special equipment, and though its connection capabilities are limited to calls to other specially-equipped facilities, its high speed and pricing make it a reasonable choice for many data communications needs. Switched 56 is used for data calls only.

TAPI can also be used with other services such as CENTREX, which provides a set of centralized network services (such as conferencing) without the need to install special equipment. With CENTREX, you pay for the use of telephone-company equipment over regular telephone lines. In addition, TAPI can be used with digital Private Branch Exchanges (PBXs) and key systems. Because TAPI is independent of the underlying telephone network, programming a PBX application using TAPI is the same as

programming a POTS application using TAPI. An application that was originally programmed for a POTS environment can be used within a PBX environment with no changes to the application's source code.

Telephony Components

Based on the Windows Open Services Architecture (WOSA) model, Windows Telephony consists of the TAPI and TAPI32 dynamic-link libraries (which forward application requests to the Telephony Service for processing), TAPISRV.EXE (which implements and manages the TAPI functions) and one or more telephony service providers (drivers). TAPI provides a device-independent interface for carrying out telephony tasks. Service providers are dynamic-link libraries that carry out low-level and possibly device-specific actions needed to complete telephony tasks through hardware devices such as fax boards, ISDN cards, telephones, and modems. Applications link to and call functions in the TAPI dynamic-link library only; they never call the service providers directly.

When an application calls a TAPI function, the TAPI dynamic-link library validates and marshalls the parameters of the function and forwards it to TAPISRV.EXE. TAPISRV (the Telephony Service) processes the call and routes a request to the appropriate service provider. To receive requests from TAPISRV, the service provider must implement the Telephony service provider interface (TSPI). A service provider can provide different levels of the service provider interface: basic, supplementary, or extended. For example, a simple service provider might provide basic telephony service, such as support for outgoing calls, through a Hayes-compatible modem. A custom service provider, written by a third-party vendor, might provide a full range of incoming and outgoing call support.

A user can install any number service providers on a computer as long as the service providers do not attempt to access the same hardware device at the same time. The user associates the hardware and the service provider when installing. Some service providers may be capable of accessing multiple devices. In some cases, the user may need to install a device driver along with the service provider.

Applications use the TAPI functions to determine which services are available on the given computer. TAPI determines what service providers are available and provides information about their capabilities to the applications. In this way, any number of applications can request services from the same service provider; TAPI manages all access to the service provider.

As long as an application does not depend on optional features, the applications can, without modification, use any services to carry out telephony tasks, even services made available after the application is developed. This is because the application always accesses the many different services through TAPI which translates the requests the application makes into the actual protocols and interfaces required.

The Telephony SPI is beyond the scope of this reference. For more information about the TSPI and service providers, see the *Microsoft Win32 Telephony Service Provider Reference*.

Media Stream

The media stream consists of the information exchanged over a call. TAPI by itself provides control only for line and phone devices and does not give access to the content of the media stream. To manage the media stream, an application uses Win32 functions, such as the Communication, Wave Audio, or Media Control Interface (MCI) functions. For example, an application that provides an interface for managing fax or data transmission uses the TAPI functions to control and monitor the line over which bits are sent, but uses the Communications functions to transmit the actual data.

In the same manner, the media stream in a speech call (where *speech* refers exclusively to human speech) is produced and controlled not by TAPI, but by one human talking to another. However, the line on which that call is established and monitored, and the call itself, remain in control of the TAPI application. (Note that *voice* is considered to be any signal that can travel over a 3.1 kHz-bandwidth channel.)

Special Hardware

Some of the more advanced capabilities of TAPI require that an application be able, for example, to retrieve data from telephones. Most telephones cannot be connected directly to computers to control speech calls and thus are currently incapable of supporting Telephony functions beyond the passive role they play in POTS. In the future, users will install and configure telephone sets like other peripheral devices. The sets will be accompanied by cards that will control the flow of information between the computer and the telephone. Client/server configurations will also be possible that allow users to take advantage of telephonic services by connecting over a LAN to a server that has such a board and associated software installed.

Physical Connections

Lines and phones can be connected in a variety of ways to the desktop computer and the telephone network. The following examples show a selection of configurations that could be supported by a service provider. Note that some of the telephone hardware required to implement some of these example is not yet widely available.

A *phone-centric* connection consists of a single POTS line in which the computer is connected to the switch through the desktop phone set. Such phone sets typically connect to the computer through one of its serial ports. When an application requests an action, the corresponding service provider sends telephony commands, which are often based on the Hayes AT command set (ANSI/TIA/EIA-602), over the serial connection to the telephone. This configuration is limited because it generally provides only line control. The computer does not have access to the media stream.

A *computer-centric* connection uses a computer add-in card or external box that is connected to both the telephone network and the phone set. The service provider can easily integrate modem and fax functions, as well as the use of the telephone as an audio I/O device.

A *BRI-ISDN* connection is similar to the computer-centric connection but allows for using the two B-channels in a variety of line configurations. A service provider can treat this connection in a number of ways:

- A single line device with a pool of two channels, allowing both channels to be combined for establishing 128 Kbps calls.
- Two separate line devices, each with exclusive use of a single B-channel.
- Two separate line devices, each drawing up to two channels from a shared pool of two B-channels.
- Three line devices: one for each of the two B-channels and one for the combination.

In the latter two models, channels may be assigned to different line devices at different times.

In *client/server networks*, a pool of telephone ports attached to a server may be shared among multiple client computers using a local area network. The ports may be configured to assign a maximum number of line devices (the *quota*) to each client workstation. It is not unusual for the sum of all quotas to exceed the total number of lines.

Also, the assignment of lines through ports is dynamic. For example, a client computer with a quota of 2 may use ports 1 and 2 at one time and ports 7 and 11 at a later time.

The service provider for the pool may model this arrangement by giving each client workstation access to two line devices. This implies that the device IDs (which are fixed) for each client are 0 and 1. If the application later requests information for device 0 and again for device 1, it must assume that the device capabilities for each device are constant, because that is the Windows device model. For server-based devices that are pooled as described in the example above, this constancy holds only for line devices that have identical device capabilities.

A *LAN-based server* might have multiple telephone-line connections to the switch. TAPI operations invoked at any of the client computers are forwarded over the LAN to the server. The server uses third-party call control between the server and the switch to implement the client's call-control requests.

This model offers a lower cost per computer for call control if the LAN is already in use, and it also offers reduced cost for media stream access if shared devices such as voice digitizers, fax and/or data modems, and interactive voice response cards are installed in the server. The digitized media streams can be carried over the LAN, although real-time transfer of media may be problematic with some LAN technologies due to inconsistent throughput.

A LAN-based host can be connected to the switch using a *switch-to-host link*. TAPI operations invoked at

any of the client computers are forwarded over the LAN to the host, which uses a third-party switch-to-host link protocol to implement the client's call-control requests.

Note that it is also possible for a private branch exchange (PBX) to be directly connected to the LAN, and for the server functions to be integrated into the PBX. Within this model, different sub-configurations are possible:

- To provide personal telephony to each desktop, the service provider could model the PBX line associated with the computer (on a desktop) as a single line device with one channel. Each client computer would have one line device available.
- Each third-party station can be modeled as a separate line device to allow applications to control calls on other stations. (In a PBX, a *station* is anything to which a wire leads from the PBX). This enables the application to control calls on other stations. This solution requires that the application open each line it wants to manipulate or monitor, which may be satisfactory if only a small number of lines is of interest, but may generate excessive overhead if a large number of lines is involved.
- Model the set of all third-party stations as a single line device with one address (one phone number) assigned to it per station. Only a single device is to be opened, providing monitoring and control of all addresses (all stations) on the line. To originate a call on any of these stations, the application must only specify the station's address to the function that makes the call. No extra line opening operations are required. However, this modeling implies that all stations have the same line-device capabilities, although their address capabilities could be different.

A potential advantage of this model is a lowered cost per computer if the LAN is already in use, but a limitation would be a possible lack of media-stream access by the computers.

The computer in use need not be a desktop computer. It can also be a laptop or other portable computer connected to the telephone network over a *wireless connection*.

In a *shared telephony* connection, the computer's connection may be shared by other telephony equipment, such as the telephone set shown below. For an application to operate properly in this arrangement, neither the application nor the service provider can assume that there are no other active devices on the line.

The Telephony Programming Model

The Telephony application programming interface (TAPI) simplifies the development of telephonic applications by hiding the complexities of low-level communications programming. TAPI accomplishes this by abstracting telephony services to make them independent of the underlying telephone network and of the way the computer is connected to the switch and phone set. Connections to the switch may be established in a variety of arrangements including directly from the user's workstation or through a server on a local area network. Regardless of their nature, telephony devices and connections are handled in a single, consistent manner, allowing developers to apply the same programming techniques to a broad range of communications functions.

Telephony API

Telephony services are divided into Assisted Telephony services and the services provided by the full Telephony API. In general, the full Telephony API is used to implement powerful telephonic applications and Assisted Telephony is used to add minimal but useful telephonic functionality to non-telephony applications. Telephony's services are divided into the groups shown in the following illustration:

{ewc msdncd, EWGraphic, bsd23547 0 /a "SDK.WMF"}

Assisted Telephony

A valuable feature of Win32 Telephony is the small set of functions called Assisted Telephony. Assisted Telephony is designed to make the establishment of voice calls and of media calls available to any Win32-based application, not just those dedicated to telephonic functionality. In other words, Assisted Telephony lets applications make telephone calls without needing to be aware of the details of the services of the full Telephony API. It extends telephony to word processors, spreadsheets, databases, personal information managers, and other non-Telephony applications. For example, adding the Assisted Telephony function **tapiRequestMakeCall** to a spreadsheet lets users automatically dial telephone numbers stored in the spreadsheet (or in a connected database).

The power of Assisted Telephony can be illustrated by the following example. A spreadsheet application can incorporate functions that dial a telephone number for a speech call. As long as the application needs none of the detailed call control provided by the full Telephony API, Assisted Telephony is the easiest and most efficient way to give it telephonic functionality. Functionality beyond dialing such as the transmission and reception of data would require additional data-transfer APIs, including the communications functions of the Comm API.

{ewc msdncd, EWGraphic, bsd23547 1 /a "SDK.WMF"}

Because Assisted Telephony and the full Telephony API are used and implemented in different ways, it is not advised to mix Assisted Telephony function calls and Telephony API function calls within a single application.

For more information about the uses and functions of Assisted Telephony, see Assisted Telephony Overview.

Service Levels

Applications whose telephony functionality goes beyond the most basic call control or are meant to handle inbound calls must be built using the Telephony API (not Assisted Telephony). The Telephony API defines three levels of service:

- The elementary level of service, called Basic Telephony, which provides a minimum set of functions that corresponds to Plain Old Telephone Service (POTS). TAPI service providers are required to support all Basic Telephony functions.
- The Supplementary Telephony level of service, which provides advanced switch features such as hold, transfer, and so on. All supplementary services are optional; that is, the service provider is not required to support them.
- The Extended Telephony level of service, in which the API provides well-defined API extension mechanisms that enable application developers to access service provider-specific functions not directly defined by the Telephony API.

Basic Telephony Services

Basic Telephony Services are a minimal subset of the Win32 Telephony specification. Because all service providers must support the functions of Basic Telephony Services, applications that use only these functions will work with any TAPI service provider. The functionality contained in Basic Telephony roughly corresponds to the features of POTS.

Today, many programmers will use only the services provided by Basic Telephony. But others, such as those writing code for PBX phone systems, will need the functions of Supplementary Telephony. Soon, the demand for ISDN and other network services, along with advancements in telephone equipment, will drive even greater usage of Supplementary Telephony.

For a list of the functions of Basic Telephony, see Quick Function Reference.

Because control of phone devices is not assumed to be offered by all service providers, phone-device services are considered to be optional. That is, they are not a part of Basic Telephony. For a list of phone-device services, see the following topic on Supplementary Telephony services, and for more information on phone devices, see Device Classes.

Supplementary Telephony Services

Supplementary Telephony Services are the collection of all the services defined by the API other than those included in the Basic Telephony subset. It includes all so-called supplementary features found on modern PBXs, such as hold, transfer, conference, park, and so on. All supplementary features are considered optional; that is, the service provider decides which of these services it does or does not provide.

An application can query a line or phone device for the set of supplementary services it provides using functions such as **lineGetDevCaps** or **lineGetAddressCaps**. Note that a single supplementary service may consist of multiple function calls and messages. The Telephony API, and not the service provider developer, defines the behavior of each of these supplementary features. A service provider should provide a Supplementary Telephony service only if it can implement the exact meaning as defined by the API. If not, the feature should be provided as an Extended Telephony Service.

As mentioned in Basic Telephony Services, phone-device services are considered optional. Therefore, all phone-device services are part of Supplementary Telephony. For a list of the functions of Supplementary Telephony, see Quick Function Reference.

Extended Telephony Services

The API contains a mechanism that allows service-provider vendors to extend the Telephony API using device-specific extensions. Extended Telephony Services (or Device-Specific Services) include all extensions to the API defined by a particular service provider. Because the API defines the extension mechanism only, the definition of the Extended-Telephony Service behavior must be completely specified by the service provider.

TAPI's extension mechanism allows service-provider vendors to define new values for some enumeration types and bit flags and to add fields to most data structures. The interpretation of extensions is keyed off the service provider's Extension ID, an identifier for the specification of the set of extensions supported, which may cross several manufacturers. Special functions and messages such as **lineDevSpecific** and **phoneDevSpecific** are provided in the API to allow an application to directly communicate with a service provider. The parameters for each function are also defined by the service provider.

Vendors are not required to register in order to be assigned Extension IDs. Instead, a utility is provided that allows the generation of Extension IDs locally. This unique ID is composed of an Ethernet-adapter address, a random number, and the time of day. An ID is assigned to a set of extensions (before distribution), not to each individual instance of an implementation of those extensions. A tool called EXTIDGEN.EXE is provided within the Win32 SDK that allows service provider authors to generate these IDs.

Extending Data Structures and Types

A range of values is reserved to accommodate future extensions to the Basic and Supplementary TAPI function set. The Extensibility section in this reference tells the amount by which a data structure can be extended. For a list of the functions used for extending Telephony, see Quick Function Reference.

Version Parameters

Every function that takes a *dwAPIVersion* or similar parameter must set this parameter to either the highest API version supported by the application or the API version negotiated using the **lineNegotiateAPIVersion** or **phoneNegotiateAPIVersion** function on a particular device. Use the following table as a guide:

Function	Meaning
lineGetAddressCaps	Use version returned by **lineNegotiateAPIVersion**
lineGetCountry	Use highest version supported by the application
lineGetDevCaps	Use version returned by **lineNegotiateAPIVersion**
lineGetProviderList	Use highest version supported by the application
lineGetTranslateCaps	Use highest version supported by the application
lineNegotiateAPIVersion	Use highest version supported by the application
lineNegotiateExtVersion	Use version returned by **lineNegotiateAPIVersion**
lineOpen	Use version returned by **lineNegotiateAPIVersion**
lineTranslateAddress	Use highest version supported by the application
lineTranslateDialog	Use highest version supported by the application
phoneGetDevCaps	Use version returned by **phoneNegotiateAPIVersion**
phoneNegotiateAPIVersion	Use highest version supported by the application
phoneNegotiateExtVersion	Use version returned by **phoneNegotiateAPIVersion**
phoneOpen	Use version returned by **phoneNegotiateAPIVersion**

Important When negotiating an API version, always set the high and low version numbers to the range of versions that your application can support. For example, never use 0x00000000 for the low version or 0xFFFFFFFF for the high since these values require that your application support all versions of TAPI, both future and past.

Device Classes in TAPI

Device classes simplify development by letting programmers treat devices that have similar properties in a similar manner. Real-world devices such as telephones, modems, and telephone lines belong to device classes. Applications access devices belonging to a given class using the same functions.

An application never needs to know which service provider controls which device.

Device classes help make TAPI extensible by providing a framework from which to classify and support new equipment.

Application developers should keep in mind the existence of other applications that share telephony services, as explained in Multiple-Application Programming.

There are two device classes: line device and phone device.

It also defines two sets of functions and messages, one used for line devices and one used for phone devices.

The line device class is a device-independent representation of a physical line device, such as a modem. It can contain one or more identical communications channels (used for signaling and/or information) between the application and the switch or network. Because channels belonging to a single line have identical capabilities, they are interchangeable. In many cases (as with POTS), a service provider will model a line as having only one channel. Other technologies, like ISDN, offer more channels, and the service provider should treat them accordingly.

A service provider may allow an application to request that multiple channels be combined in a single call (as, for example, when ISDN "B" channels are combined into "H" channels) to give the call wider bandwidth, using a technique often referred to as inverse multiplexing. This added bandwidth enables the call to transmit more information at the same time. For most current telephonic purposes, inverse multiplexing is not necessary.

In POTS, it is normally necessary to assign one channel per line, but with ISDN, a line's channels are dynamically allocated when an application makes or answers a call. Because these channels have identical capabilities and are interchangeable, the application need not identify which channel is to be used in a given function call. Channels are owned and assigned by the service provider for the line device in a way that is transparent to applications. This channel management is a method of abstraction that eliminates the need to introduce the naming of channels by TAPI.

Just as a line device class is an abstraction of a physical line device, the phone device class represents a device-independent abstraction of a telephone set. TAPI treats line and phone devices as devices that are independent of each other. In other words, you can use a phone (device) without using an associated line, and you can use a line (device) without using a phone.

Service providers that fully implement this independence can offer uses for these devices not defined by traditional telephony protocols. For example, a person can use the handset of the desktop's phone as a waveform audio device for voice recording or playback, perhaps without the switch's knowledge that the phone is in use. In such an implementation, lifting the local phone handset need not automatically send an offhook signal to the switch.

This independence also allows an application to ring the local telephone in a manner that is independent of inbound calls. The capabilities of service providers is limited by the capabilities of the hardware and software used to interconnect the switch, the phone, and the computer. For detailed information about specific device classes, see Device Classes.

Addresses to Lines Assignments

An address is the telephone number, complete with national or international codes, of a telephone, fax machine, or other device that can receive calls. Addresses can be dialed by a human or stored in an electronic directory for retrieval and use by a telephony application. For more complete information on addresses assignments, channels, and lines, see Line Devices Overview.

The local assignment of an address to a line (that is controlled in TAPI) takes place in the setup operation for the service provider. This can be done using the Control Panel to configure the service provider or by calling the **lineConfigDialog** function from within the application. On the local side of the central office, everything about a line is controlled by a service provider, such as whether there are multiple addresses and what these addresses are.

Usually, there is exactly one address per line, with the following exceptions:

- **Multiple Addresses with POTS**. In POTS, multiple addresses work only with systems that support distinctive ringing or are connected to a *DID trunk*. (DID—direct inward dialing—is an extra-fee service provided by the phone company.) With DID in a multi-user voice mail system, the dialed number is signaled to the system on the DID trunk before the call rings. This allows the system to play the called party's pre-stored announcement message and to store any incoming messages in the correct voice mail box.

  On a residential line with distinctive ringing service, different ringing patterns correspond to multiple numbers assigned to the same line.
- **Multiple Addresses with ISDN**. ISDN was designed to allow simultaneous multiple addresses by providing multiple channels, each of which can have its own address. On an ISDN network, *call offering* (which means a call-setup message has been sent from the switch) takes place before ringing, so the call can be redirected before it is answered. The **lineAccept** function means *start ringing* for ISDN. For POTS, it means that some application has accepted responsibility for the call and has presented it to the user.

Call Control

The developer's view of telephony is one in which telephone lines and phone sets are logically connected through TAPI. This logical connection also provides a point of termination for the telephone line. The physical connection can be made at the desktop, or at a LAN-based host or server, where a LAN protocol *extends* the connection of the phone lines or phone to the client application. TAPI uses a first-party call-control model on the logically terminated line as well as control of the associated phone device, if any.

Applications access Telephony API services using a first-party call control model. This means that the application controls telephone calls as if it is an endpoint (the initiator or the recipient) of the call. The application can make calls, be notified about inbound calls, answer inbound calls, invoke switch features such as hold, transfer, conference, pickup, and park, and can detect and generate DTMF tones for signaling remote equipment. An application can also use TAPI functions to monitor call-related activities occurring in the system.

In contrast, third-party call control means that the controlling application does not act as an endpoint of the call. A third-party call-control model allows an application to establish or answer a call between any two parties—the application does not act as either of these parties.

A service provider may implement TAPI's line and phone functions by treating the set of all stations on the switch as a single line device to which multiple phone numbers are assigned. Each phone number on the line device maps to one of the stations on the switch—that is, calls passing through the switch can reach a local station by using its address (telephone number). The application can answer calls or make calls, selecting any one of the addresses on the line device as the origination number. Although the application appears to be the originating party, a call is actually established between the station whose address was selected by its originating number and the other party. However, this implementation is a type of third-party call control and is not a design goal of TAPI, which emphasizes first-party call control applications.

Media Access

The media mode is the form in which data is transmitted on a line. The four main types of media mode are voice, speech, fax, and data. With TAPI, calls can be established independently of the call's media mode.

The media stream is the actual stream of information that travels on the line. Phone devices and calls on line devices are capable of carrying media streams. The Telephony-API line and phone device classes provide a wide range of control operations for these devices, but access to the media stream itself is not provided by TAPI. Instead, the application must use other APIs for the Win32 environments to access or manage these media streams. These APIs include the Waveform API, the Comm API, and the MCI (Media Control Interface). The Waveform API is used for multimedia programming, the MCI provides a high-level generalized interface for controlling media devices, and the Comm API is the set of communications functions provided by the Win32 SDK.

For example, for line devices, an application can use TAPI to establish a connection to another station. Once the connection is established, the application can then use the Waveform API (or the MCI Waveaudio API) on the associated device to play back (send) and record (receive) audio data over the connection. Similarly, if the connection's media stream is from a modem, an application would use the modem configuration extensions of the Comm API to control the media stream.

To provide TAPI and media-stream access to either a phone or a call on a line device, the service provider must implement both the Telephony SPI and the appropriate media stream SPI or DDI (device-driver interface). The service provider can support lines and phones simultaneously.

Because these device classes and media stream interfaces function independently of one another, coordination of their usage must occur at the application level. Multiple applications that share calls and media streams in nontrivial ways will likely need to coordinate their activities at the application level to prevent conflicting usage of TAPI and the media stream API in use. For more information on preventing conflicts, see Multiple-Application Programming.

TAPI reports changes in the type of media stream (voice, fax, data modem, and so on) to participating applications. This process is sometimes referred to as call classification. The mechanism used to determine the type of media stream is specific to the service provider. For example, a service provider may filter the media stream for energy or tones that characterize the media type, or it may use distinctive ringing, information exchanged in messages over the network, or knowledge about the caller or called ID to make this determination.

TAPI also provides limited support for control of the media stream on a call, particularly in server-based networks. The actual data does not pass through TAPI, but TAPI can be used to a limited extent to control functions that control the media stream. This control is provided to avoid latency (delay) problems that could arise in client/server configurations for which the application is forced to use the stream's media API. An application can request actions on a call's media stream if these actions are to be triggered by events normally reported by TAPI, such as the detection of a tone or DTMF digit, or the transition of a call to a specified call state.

For example, an application can request that a call's media stream be suspended (with **lineSetMediaControl**) when a # DTMF digit is detected on the call, and that the media stream be resumed when a * DTMF digit is detected. Note that some implementations or configurations will be unable to provide any media-control functions or media access to the phone or line. Providing media control is optional to the service provider; it should provide performance benefits primarily for client/server implementations. Because it is optional and because only limited control is provided, its usage is generally discouraged. If possible, applications should use the media stream's control functions instead.

Application Notifications

The programming model for Win32 Telephony matches the standard programming model for the Win32 environment with regard to device naming, the use of sections and entries in the registry, and function calling conventions. But it deviates from this model in one important way—the synchronous/asynchronous operational model, which is a callback scheme through which applications are notified of the success or failure of function calls and other events.

Application Message Notification Mechanisms

TAPI 2.0 and above support three mechanisms for notifying applications of changes in the status of calls, lines, and phones: a callback function, Win32 events, and completion ports. These are described in detail in the documentation for **lineInitializeEx** and **phoneInitializeEx**.

Prior to TAPI version 0x0002000, only one such mechanism existed: the callback function. When the callback mechanism is used, the application's callback function is invoked from within the application's thread (at the time the application calls the **GetMessage** function), providing a normal, fully functional execution environment in which all Win32 APIs can be safely invoked.

A LINE_REPLY or PHONE_REPLY (asynchronous completion) message sent to the application carries the request ID and an error indication. Valid error indications for this reply are identical to those that are returned synchronously for the associated request, or zero for success. Only the application that issued the request will receive the reply message, but when the request causes changes in the state of the device or call, other interested applications may also receive event-related messages.

TAPI guarantees that a reply message is made for every request that operates asynchronously, unless the application shuts down TAPI (by calling **lineShutdown** or **phoneShutdown**) before the reply is received.

Information Returned by Functions

An application receives two kinds of information as a result of a function call: the function's return value, and values written to data locations specified by the function's arguments.

If the function's return value is zero, the application knows that the function has completed synchronously. In this case, any values written as a result of the function call are reliable and can be used immediately. However, if the return value is positive, the function has not yet completed but it will complete asynchronously, at which time TAPI notifies the application by sending it an asynchronous reply message for the function. Once the application receives this message (and the message indicates success), any values returned by the function are considered to be reliable. However, before the message is received, the application should consider these values suspect and should not use them. Also, because asynchronous reply messages can take varying lengths of time to be sent, the application may not receive them in the same order in which it called their functions. This is why an application must retain the request IDs of its requests in progress so that it can identify and correctly respond to incoming asynchronous reply messages.

Example Illustrating the Programming Model

Consider a Win32-based application that can make either voice or data (modem) calls. Although these calls could be made simultaneously if a telephone line were in place for each device (the telephone and the modem), assume that there is only one line, so calls are placed one at a time.

For this discussion, a line is defined as a physical telephone line leading from the wall to the telephone company's switch, and a line device (such as a fax or modem) is a local device on the telephone line. Also, this example is restricted to POTS; the telephone line is a standard, two-wire twisted-pair cable that carries an analog signal and constitutes a single channel.

The line devices attached to the computer are visible to the application as instances of the line device class, which is defined by TAPI. The physical telephone is visible to the application as an instance of the phone device class. This application therefore must be able to execute two types of calls: voice and data. One strength of the TAPI programming model is the way its abstraction into classes exploits the similarities between these different types of calls.

For a more in-depth discussion of this process and related ones, see TAPI Applications.

Determining the Call Type (Media Mode)

Before an application uses any telephony services, it needs to interact with the user to know what kind of call to make. To do this, the standard Win32 API functions are used to build the menus or dialogs needed to gather the user input that tells your application what to do. In this example, the user specifies the transmission of data, so the application will make a call that transmits a specified file to another user.

Initializing TAPI

Before placing a call, an application must establish a means of communication between itself and TAPI. The application must select a telephony event notification mechanism, and specify this in a call to the **lineInitializeEx** function (see the description of this function for details on the available notification mechanisms). One of the values **lineInitializeEx** returns is the number of line devices available to the application. In this example, that number is one, and the line's ID in Telephony's zero-based scheme is 0. The application must establish this communication link (with **lineInitializeEx**) regardless of the type of call to be placed or received.

Obtaining a Line

Next, the application needs to obtain a handle to a capable telephone line. The application opens the line with **lineOpen**, but before doing this, it must make sure that the line can support the desired type of call. Invoking the **lineGetDevCaps** function returns that information to the application in a data structure of type **LINEDEVCAPS**. If data calls were not supported, this fact could be reported to the user in a dialog box. The application does not need to use **lineGetDevCaps** before every call, because a line's capabilities should remain static. If the local telephonic configuration (as expressed in .INI files) changes, TAPI notifies applications, which can then call **lineGetDevCaps** to see what has changed.

One of the values returned by **lineGetDevCaps** (as a field in the **LINEDEVCAPS** structure) is the number of addresses assigned to the specified line device. In this example, a single device has a single address.

Ownership of a call is a type of privilege. Applications obtain owner or monitor privilege to new incoming calls by specifying the desired privilege as a parameter of **lineOpen**. The privilege with which a line is opened remains in effect for subsequent calls used by that application on that line. An application always has owner privileges on calls it creates. When the application opens a line to place calls (as opposed to taking inbound calls) it invokes **lineOpen** with the privilege LINECALLPRIVILEGE_NONE, which insulates the application from incoming calls while allowing outgoing calls. The other privileges used with the **lineOpen** function are only for incoming calls.

The LINEOPENOPTION_SINGLEADDRESS option is available when using the **lineOpen** function to allow the application to specify that handles for new calls (either monitor or owner handles) should be delivered to the application only if the address on which the call appears matches the address provided as a function parameter. This is extremely useful when different addresses on a line are designated for calls of different media modes.

Placing the Call

Once the application has opened the line device, it places the call with **lineMakeCall**, specifying the address (phone number and area code) in the *lpszDestAddress* parameter and the media mode (datamodem, in this case) in the *lpCallParams* parameter. This function returns a positive "request ID" if the function will be completed asynchronously, or a negative error number if an error has occurred. Negative return values describe specific error states. LINEERR_CALLUNAVAIL, for example, means that the line is probably in use (someone else already has an active call). If dialing completes successfully, messages are sent to the application to inform it about the call's progress. Applications typically use these messages to display status reports to the user.

Later, when the **lineMakeCall** function has successfully set up the call, the application receives a LINE_REPLY message (the asynchronous reply to **lineMakeCall**). At this point there is not necessarily a connection to the remote station, just an established call at the local end—perhaps indicated by a dial tone. This LINE_REPLY message informs the application that the call handle returned by **lineMakeCall** is valid.

TAPI's programming model treats data calls similarly to voice calls, as shown by the fact that the same function is used to make calls of both types. If LINEBEARERMODE_DATA is specified in a field of the *lpCallParams* parameter of **lineMakeCall**, the call is set up to send data. Speech transmission can be chosen by using a different value. And if NULL is specified, a default 3.1 kHz voice call is established, which can support the speech, fax, and modem media modes.

> **Note** TAPI should not be used for fax transmissions. Instead, use the functions available through MAPI, the Microsoft Messaging API.

As the call is placed, it passes through a number of states, each of which results in a LINE_CALLSTATE message sent to the application. These states include dialtone, dialing, ringback, and, if connection succeeds, LINECALLSTATE_CONNECTED. (To see the complete list of call states, see the **LINECALLSTATUS** structure.) After the message indicating the connected state is received, the application can begin sending data.

Sending Data

If the line is available (not busy) and the connection is established, the data can be sent. The application accomplishes this by giving control back to the user, who, using a dialog box, specifies the file to send and initiates data transmission. Though TAPI functions continue to manage the opened line and the call in progress, actual transmission is started and controlled by non-TAPI functions. In this case, for example, the Comm API of the Win32 SDK could be used to control the media stream.

If the application were setting up a speech call, its actions would be similar. Once the call is established, the duty of data transmission is transferred outside of TAPI to the people who wish to speak, although the line and call continue to be monitored by the application using TAPI functions.

Ending the Call

When the modem transmission is finished, the application receives a LINE_CALLSTATE message, which informs it that the state of a line device has changed. In this example, a remote disconnect has occurred. The application disconnects the call at the local end (it "goes on-hook") with **lineDrop**. Alternatively, the application itself may choose to end the call by invoking **lineDrop** before receiving the remote-disconnect message.

Here are the steps that might be used to end a call, close the line, and leave TAPI:

1. The application calls **lineDrop**, which places the call in the IDLE state. The call still exists, and the application still has its handle. Now the application can examine the call-information record, if desired.
2. The application calls **lineDeallocateCall** to release the call handle for the finished call. The call no longer exists.
3. If the application expects no more calls on the line, it uses **lineClose** to close the line. At this point, there will be no more incoming or outgoing calls on that line.
4. The application invokes **lineShutdown** to end the use of TAPI's functions for the current session.

Synchronous/Asynchronous Operation

The interactive nature of telephony requires that TAPI be a real-time operating environment. Many of TAPI's functions are required to complete quickly and return their results to the application synchronously. Other functions (such as dialing) may not be able to complete as quickly and therefore operate asynchronously. Any given operation always completes either synchronously or asynchronously, and both types of operation are explained in the following topics. A list of all TAPI functions, which states whether each one completes synchronously or asynchronously, appears in Quick Function Reference.

Synchronous Functions

An operation that completes synchronously performs all of its processing in the function call made by the application. The function returns different values depending on its success or failure:

- **Synchronous Success**. If the request or service corresponding to the function has been carried out successfully, the function returns zero, indicating success. Any values written as a result of the function call are reliable and can be used immediately.
- **Synchronous Failure**. If the function detects an error and the request is not carried out, a negative error number is returned to identify the error.

Asynchronous Functions

An operation that completes asynchronously performs part of its processing in the function call made by the application and the remainder of it in an independent execution thread after TAPI has returned from the function call. To ensure completion of the call's processing, the service provider vectors the request to another active entity in the system—such as a LAN server, add-in hardware, a switch, or a network—and then returns to the application. At this time, either a negative error result or a positive request ID is returned to the application.

At the time of asynchronous completion (the service provider has received an interrupt from the hardware, meaning that a message must be delivered), the service provider calls TAPISRV.EXE and reports that "Event X has just taken place. Deliver an appropriate message to all concerned applications." When TAPISRV.EXE receives this message, it forwards the message the TAPI dynamic-link library, in the application's process, which in turn posts a window message, signals an event handle, or posts to an IO completion port, according to the message notification scheme selected by the application in **lineInitializeEx** or **phoneInitializeEx**.

When the asynchronous portion of the operation completes, a LINE_REPLY (or PHONE_REPLY) message is sent to the application. This message contains, as one of its parameters, the request ID returned by the function call. This request ID allows the application to determine which original request has completed. (Applications should remember the request IDs of all their requests in progress so that reply messages can be properly handled.) A second parameter to the LINE_REPLY (or PHONE_REPLY) message is the asynchronous return value. This is either a negative value (for an error) or zero if the operation completed successfully. For asynchronous operations, any of the return values may be returned as part of the function return or as the *dwParam2* parameter in the _REPLY message. The value 0, which indicates success, will only be returned in the LINE_REPLY message, and never as the function's return value.

The initialize functions (**lineInitializeEx** and **phoneInitializeEx**) tell TAPI how to send these messages to the application.

> **Note** In some cases, if a multithreaded application calls an asynchronous function from a thread other than the thread from which the application initialized the line or phone device, the application may receive the LINE_REPLY or PHONE_REPLY message before the asynchronous function has returned. In such cases, the application should save the message parameters and wait until the asynchronous function returns and the request ID is known before processing the message.

The Meaning of SUCCESS

When an operation returns a SUCCESS indication (either synchronously upon function return for synchronous operations, or asynchronously through a LINE_REPLY or PHONE_REPLY message for asynchronous operations), the following is assumed to be true:

- The function has successfully progressed to a point that is defined by the API on a function-by-function basis. After that point has been reached, either the operation is completely done, or it will be in a state such that independent state messages will inform the application about subsequent progress.

  For example, a service provider's implementation of **lineMakeCall** should return SUCCESS no later than when the call enters the proceeding call state. Ideally, the provider should indicate SUCCESS as soon as possible, such as when the call enters the dial tone call state (if applicable). Once SUCCESS has been returned to the application, LINE_CALLSTATE messages will inform the application about the progress of the call. Service providers that delay returning the **lineMakeCall** SUCCESS indication, say, until after dialing is complete, must be aware that this places that provider at a disadvantage because the usability at the application level may be severely limited. For example, it would not be possible for a user to cancel the call setup request in progress until after dialing is complete and all digits had been sent to the switch.
- Functions that return information (such as **lineGetCallInfo**) return SUCCESS only when the requested information is available to the application. Functions that return handles (to lines or calls), can return SUCCESS only after the handle is valid. No messages should be sent about that line or call prior to the SUCCESS indication of the function that caused its creation. The service provider is responsible for suppressing such messages.
- Functions that enable certain permanent conditions (such as **lineMonitorDigits**) return SUCCESS only after the condition is enabled, not when the condition is removed again (for example, not when all digit monitoring has completed).
- Call-control functions (such as **lineHold** or **lineSetupTransfer**, but not **lineMakeCall**) return SUCCESS when the operation is completed. Some telephone networks do not provide acknowledgment (positive or negative) about the completion of certain requests made by service providers. In such situations, the service provider must decide upon success or failure of the request. Therefore, SUCCESS may indicate that the service provider has initiated actions to fulfill the request, but not necessarily anything more. For example, the provider may receive no affirmative acknowledgment to its request from the switch, although it has already sent a success message to the application.

TAPI Applications

This section explains what you need to know to program basic telephonic functionality using the line device class functions.

Establishing a Link

Before an application can call the functions of Win32 Telephony for using a line device, it must take the following steps:

- Initialize the TAPI environment and TAPI's functions with an initialization function. Invoking this function also informs the application of the number of line devices available.
- Negotiate the API version, and if necessary, negotiate the Extensions version.

An application must take the preceding steps for each line device it intends to use.

Applications should not invoke **phoneInitializeEx** without subsequently opening a phone (at least for monitoring). If the application is not monitoring and not using any devices, it should call **phoneShutdown** so that memory resources allocated by the TAPI dynamic-link library can be released if unneeded, and library itself can be unloaded from memory while not needed.

Opening Lines

After having obtained the capabilities of a line, an application must open the line device before it can access telephony functions on that line. (Because a *line device* is an abstraction of a *line* as defined by Telephony, *opening a line* and *opening a line device* can be used interchangeably.) When a line device has been opened successfully, the application receives a handle for it. The application can then use that line to take inbound calls, make outbound calls, or monitor call activities on the line for logging purposes.

To open a line device for any purpose—monitoring or control—the application calls the function **lineOpen**. (Later, when the application is finished with the line device, it can close it with **lineClose**.)

The function **lineOpen** can be invoked in one of two ways:

- A *specific* line device is selected by means of its line-device ID (the parameter *dwDeviceID*). The **lineOpen** request will open the specified line device. Applications interested in handling inbound calls typically use specific line devices because the application has been notified which line is carrying or is expected to carry the inbound call. When a line device has been opened successfully, the application is returned a handle representing the open line.
- The application can specify that it wants to use *any* line device that has certain properties. In this case, the application uses the value LINEMAPPER instead of a specific line-device ID as a parameter for **lineOpen**. The application also specifies which properties it needs on the call in parameters to **lineOpen**. The function opens any available line device that supports the specified call parameters. This attempt, of course, may fail. If successful, the caller can determine the line-device ID by calling **lineGetID**, specifying the handle (*lphLine*) to the open line device returned by **lineOpen**.

An application that has successfully opened a line device can use it to make an outbound call except when the line supports only inbound calls.

Selecting One or More Lines

An application can open one or more lines for various purposes. For example, it can open one line for monitoring calls and another line for making outgoing calls. If several lines are available, the application can choose to open any or all of them. To decide which of several line devices to use, determine the capabilities of each one with **lineGetDevCaps**. This tells whether the line supports the functionality needed by the calls to be made or received—such as their required media mode. This function is also used to get the name of the line.

Because opening a line merely means obtaining a handle to the line (*hLine*) with a given privilege, an application can obtain more than one handle to the same line. In other words, an application can open the same line many times, also called opening different *instances* of the line. For example, a line may simultaneously be opened once for monitoring calls, a second time for accepting incoming calls as their owner, and a third time for making outgoing calls.

After choosing a suitable line (or lines), the application uses **lineOpen**, either specifying a certain line or using LINEMAPPER, as explained in the previous section.

Specifying Media Modes

The ability of an application to deal with inbound calls or to be the target of call handoffs on a line is determined by the value used for the *dwMediaModes* parameter of the **lineOpen** function. With this function, the application indicates its interest in monitoring calls or receiving ownership of inbound calls of one or more specific *media modes*, or of *any* (unspecified) media mode, as described in the following cases:

- An inbound call of a *certain* media mode is given to the application that has opened the line device for that particular media mode. A single application may specify multiple flags simultaneously to handle multiple media modes.
- An application that wants to handle calls for which the actual media mode present has not yet been determined would turn on the *unknown* media bit as a parameter of **lineOpen**.
- An application that wants to handle calls of any media mode would indicate this capability by turning on *all* of the media bits that are supported on the line (which can be obtained from **LINEDEVCAPS**).

All applications that have opened a line device in any mode are notified about certain general statuses and events occurring on the line device or its addresses. These include the line being taken out of service, the line going back into service, the line being under maintenance, an address coming in use or going idle, and an open or close operation being executed on the line. An application that does not care about a certain message can use the **lineSetStatusMessages** function to filter the message. Most such status messages are disabled by default; an application would need to call **lineSetStatusMessages** to enable them.

The media modes specified with **lineOpen** add to the default value for the provider's media mode monitoring for initial inbound call type determination. This means that the *dwMediaMode* settings of all applications with the line open are **OR**ed together, and that union becomes the default media detection on the line. The **lineMonitorMedia** function modifies the mask that controls LINE_MONITORMEDIA messages but does not affect the default media detection enabled on new incoming calls. It is necessary for an application using **lineMonitorMedia** to call it to establish media monitoring on every new call in which it is interested.

Requesting Call Privileges

In addition to media mode, an application can specify the call *privileges* it wants for the calls provided to it. With privileges, an application specifies whether it wants to monitor calls or own them. For an inbound call, only one application is selected as the owner, although all applications with monitor interest in the call are also notified about the incoming call. The usual privileges an application specifies are summarized in the following list:

- If the application only wants to monitor calls, it specifies LINECALLPRIVILEGE_MONITOR. The application will also receive monitor handles to outgoing calls placed by other applications (an application receives owner handles for outgoing calls it places itself). It will also receive MONITOR handles for calls it places itself on other instances of the same line.
- If the application wants to make outbound calls only, it specifies LINECALLPRIVILEGE_NONE. An application that has NONE selected will not be automatically informed of incoming calls. However, it can also become aware of calls on the line with LINE_ADDRESSSTATE(numCalls) or LINE_LINEDEVSTATE(numCalls) messages. It can then call **lineGetNewCalls**..
- If the application wants to accept incoming calls of a specific media mode (or modes), it specifies LINECALLPRIVILEGE_OWNER and one or more relevant LINEMEDIAMODE_ settings.
- If the application is willing to control unclassified calls (incoming calls of as-yet unknown media mode), it specifies LINECALLPRIVILEGE_OWNER and LINEMEDIAMODE_UNKNOWN.
- In other cases, the application should specify the media mode it is interested in handling and set *dwPrivilege* to LINECALLPRIVILEGE_OWNER.

An application that wants to be informed of all calls on the line regardless of whether it can become an owner on the call can set both the LINECALLPRIVILEGE_OWNER and LINECALLPRIVILEGE_MONITOR bits. It will get call handles with owner privileges for incoming calls for which it is the highest priority application for the highest priority media mode on the call, and monitor privileges for all other incoming and outgoing calls.

An application that has successfully opened a line device can always initiate calls using **lineMakeCall**, **lineUnpark**, **linePickup**, **lineSetupConference** (with a NULL *hCall* parameter), as well as use **lineForward** (assuming that doing so is allowed by the device capabilities, line state, and so on).

Application Priority

Conflicts can arise if multiple applications open the same line device for the same media mode. These conflicts are resolved with a priority scheme by which the user assigns relative priorities to the applications. This is usually done through a Control Panel utility or a Preferences menu in a telephonic application. Note the following points about this mechanism:

- Only the highest priority application for a given media mode ever receives ownership (unsolicited) of a call of that media mode.
- Although ownership is usually received when an inbound call first arrives or when a call is handed off, any application (including a lower priority one) can later acquire ownership by using the function **lineGetNewCalls** or **lineGetConfRelatedCalls**.

The user can assign relative priorities to the modules (the applications and the DLLs) that use TAPI. The resulting configuration information is stored in the registry.

For in-depth information about the way applications receive calls in a multi-application environment, see Multiple-Application Programming.

Using lineOpen

An application can open a number of lines as well as negotiate API and extension versions. The application can call the **lineOpen** function with LINECALLPRIVILEGE_MONITOR privilege, meaning that it will only monitor, not own, incoming calls on all the lines opened.

An application could open a line with the intent of *owning* incoming calls by specifying LINECALLPRIVILEGE_OWNER as the privilege and a media mode other than NONE. The application could actually specify a number of media modes in this parameter by **OR**-ing the bit flags for each of the desired media modes. In that case, the application would be notified of incoming calls in any of the specified media modes, and it receives those calls as their owner. (Actually, another application that is also registered to receive calls of that media mode would receive the call instead, if it has a higher priority as designated in the registry.) This notification arrives in a call-state message that specifies, among other information, which line is carrying the incoming call. For example, by specifying LINEMEDIAMODE_INTERACTIVEVOICE, the application would be notified of incoming calls of the interactive voice media type (voice calls with a person on the local end of the line).

Receiving Information

An application receives information in two ways: solicited and unsolicited. Solicited information is requested by the application through a function call such as **lineGetDevCaps** or **lineGetAddressCaps**. Unsolicited information arrives in the form of messages—most importantly call-state messages. Often, the two mechanisms are used together, as when an application receives a LINE_CALLSTATE message, after which it checks the information contained in the **LINECALLINFO** structure by calling **lineGetCallInfo**.

An application can call **lineGetDevCaps** to learn more about available lines. The application determines the names of the lines and the number of addresses on those lines. (An important factor in the configuration of lines and addresses is the way the service provider chooses to map lines and addresses. Though the application has no control over this mapping, it can determine the details of the mapping by calling functions such as **lineGetDevCaps**.) Later, using this information, the application could allow the user to choose which line (and address) to use for an outgoing call, restricting the lines it displays (in a dialog box, for example) to those that support a specific media mode. As an example, an application designed to be used only for faxing may choose to let the user select only lines that support fax transmission.

Call **lineGetAddressCaps** to obtain information for a given address. The application can use the names of the addresses to let the user choose them in a popup menu, but other information is also reported, such as whether caller-ID is supported, what kinds of call states can be produced, and how many active calls can exist on that address.

Changes in the Status of a Line Device

The status of a line device can change for many reasons, some as a result of requests submitted by the local application, and some as a result of actions performed by the switch or by the application (or person) at the other end of the connection.

In either case, an application is notified about these changes with the LINE_LINEDEVSTATE message, which indicates the status item (the attribute of the line device) that has changed. The application can choose the line status items for which it wants to be notified using the function **lineSetStatusMessages**. The messages controlled by invoking **lineSetStatusMessages** are LINE_LINEDEVSTATE and LINE_ADDRESSSTATE.

In addition, an application can determine the current status of an address by calling **lineGetAddressStatus**, which returns its information in a structure of the type **LINEADDRESSSTATUS**. It can also see the complete status of the specified open line device by calling **lineGetLineDevStatus**, which returns its information in a structure of the type **LINEDEVSTATUS**.

Receiving Calls

After an application has opened a line device and, while doing so, registered a privilege other than *none*, and a media mode, it is notified when a call arrives on that line. Specifically, applications that have the line open with LINECALLPRIVILEGE_MONITOR will receive a LINE_CALLSTATE message for every call that arrives on the line. An application that has opened the line with LINECALLPRIVILEGE_OWNER receives a LINE_CALLSTATE message only if it has become an owner of the call or is the target of a directed handoff. In this notification, TAPI gives the application a handle to the incoming call, and the application keeps this handle until the application deallocates the call.

> **Note** To assist in object-oriented implementations of TAPI, in versions 0x00020000 and greater TAPI initially sends a LINE_APPNEWCALL message (instead of a LINE_CALLSTATE message) to the application to notify it of a new call handle.

Applications are informed of call arrivals and all other call-state events with the LINE_CALLSTATE message. This message provides the call handle, the application's privilege to the call, and the call's new state. For an unanswered inbound call, the call state is *offering*. An application can invoke **lineGetCallInfo** to obtain information about an offering call before accepting it. This function call also causes the call information in the **LINECALLINFO** data structure to be updated. By knowing the call state and other information, the application can determine whether the call needs to be answered.

The call information stored in **LINECALLINFO** includes, among other things, the following items:

- *bearer mode, rate* This is the bearer mode (voice, data) and data rate (in bits per second) of the call, for digital data calls.
- *media mode* The current media mode of the call. *Unknown* is the mode specified if this information is unknown, and the other set bits indicate which media modes might possibly exist on the call. For more information, see Multiple-Application Programming.
- *call origin* Indicates whether the call originated from an internal caller, an external caller, or an unknown caller.
- *reason for the call* Describes why the call is occurring. Possible reasons are:
  - Direct call
  - Transferred from another number
  - Busy−forwarded from another number
  - Unconditionally forwarded from another number
  - The call was picked up from another number
  - A call completion request
  - A callback reminder

  The reason for the call is given as *unknown* if this information is not known.
- *caller-ID* Identifies the originating party of the call. This can be in a variety of (name or number) formats, determined by what the switch or network provides.
- *called-ID* Identifies the party originally dialed by the caller.
- *connected-ID* Identifies the party to which the call was actually connected. This may be different from the called party if the call was diverted.
- *redirection-ID* Identifies to the caller the number towards which diversion was invoked.
- *redirecting-ID* Identifies to the diverted-to user the party from which diversion was invoked.
- *user-to-user information* User-to-user information sent by the remote station (ISDN).

The LINE_CALLSTATE message also notifies monitoring applications about the existence and state of outbound (and inbound) calls established by other applications or established manually by the user−for

example, on an attached phone device (if the telephony hardware and the service provider support monitoring of actions on external equipment). The call state of such calls reflects the actual state of the call as follows: An inbound call for which ownership is given to another application is indicated to the monitor applications as initially being in the *offering* state. An outbound call placed by another application would normally first appear to the monitoring applications in the *dialtone* state.

The fact that a call is offered does not necessarily imply that the user is being alerted. Once alerting (ringing) has begun, a separate LINE_LINEDEVSTATE message is sent with a *ringing* indication to inform the application. It may be necessary, in some telephony environments, for the application to accept the call (with **lineAccept**) before ringing starts. The application can determine whether or not this is necessary by checking the LINEADDRCAPFLAGS_ACCEPTTOALERT bit.

Depending on the telephony environment, not all the information about a call may be available at the time the call is initially offered. For example, if caller ID is provided by the network between the first and second ring, caller ID will be unknown at the time the call is first offered. When it becomes known shortly thereafter, a LINE_CALLINFO message notifies the application about the change in party-ID information of the call.

Incoming Calls and Line Privileges

An application cannot refuse ownership of a call for which it receives an owner handle. Whether the call is delivered to the application with owner or monitor privileges is decided before the call arrives—at the time the application opens the line on which the call is established by the remote caller.

If the application opens the line with **lineOpen** with the parameter *dwPrivilege* set to LINECALLPRIVILEGE_MONITOR, it automatically receives a handle with monitoring privileges for all incoming calls on the line. It can then choose to become an owner by calling **lineSetCallPrivilege**. The fact that it indicated MONITOR when it opened the line does not prevent it from later becoming an owner with **lineSetCallPrivilege** or by originating a call with **lineMakeCall** (an application is always an owner of calls it places regardless of the privilege specified with **lineOpen**).

When an incoming call has been offered to an application and the application is an owner of the call, the application can answer the call with **lineAnswer**. Once the call has been answered, its call state typically transitions to *connected*, at which time information can be exchanged over the call.

An application can receive handles to incoming calls only for monitoring. You can modify the application (specifically, the parameters for **lineOpen**) to change the privileges with which it initially opens lines.

Securing a Call

If a new call arrives while another call exists on the line or address, similar notification and call information may be supplied following the same mechanism as for any incoming call. If an application does not want any interference by outside events for a call from the switch or phone network, it should secure the call. Securing a call can be done at the time the call is made with a parameter to **lineMakeCall**, or later (when the call already exists) with **lineSecureCall**. The call will be secure until the call is disconnected. Securing a call may be useful, for example, when certain network tones (such as those for call waiting) could disrupt a call's media stream, such as fax.

Logging Call Information

An application can call the function **lineGetCallInfo** to obtain information about a call. Although this function fills the **LINECALLINFO** structure with a large amount of data, applications need to maintain other items, such as the start and stop time of the call.

Developers of applications that log call information should note the following guidelines when designing those applications:

- Free the call's handle (*hCall*) when the call goes idle—that is, when a LINECALLSTATE_IDLE message is received for the call. At any point in the call's existence prior to its deallocation, monitoring applications can retrieve information about the call.
- To keep the call's log sheet complete, log the fact that the call has gone idle.
- Some applications may also need to update the user interface to show that important events have occurred, such as the fact that a fax is being received.

For more information about call logging, see Multiple-Application Programming.

Establishing a Call

Once an application has determined that a given line offers the needed set of capabilities, and then opens that line, it can access telephony functions for either incoming or outgoing calls on the line. The usual way to place a call on that line is to invoke **lineMakeCall**, specifying the line handle and a dialable destination address.

Address Translation

Applications can provide users with location independence and take advantage of calling-card information managed by Win32 Telephony by storing telephone numbers in the Canonical address format. Before a Canonical address can be used in placing a call, it must be converted (translated) into the *Dialable* address format using the **lineTranslateAddress** function.

To do this, the **lineTranslateAddress** function starts by examining the settings in the registry to find the user's location, including the country and area code. It then produces a valid dialing sequence by removing unnecessary portions of the number (such as the country code or area code) and adding other digits such as a long distance prefix or a digit used to dial out of a local PBX.

To avoid inadvertent misdialing, such as if the user has changed locations but has not yet informed Win32 Telephony of the change, an application may want to present the output of this function call to the user in a dialog box. The user can then confirm the translated address or change it if it is incorrect.

Toll Lists

In some locations in North America, all calls placed to the local area code are local calls. In other locations, some calls placed to the local area code are long distance, and need a "1" to be dialed. The first three digits of the address (the prefix) determine whether or not a call within the local area code is a toll call.

A *toll list* is a list of prefixes in the local area code whose addresses must be dialed as long distance addresses, and are assessed long distance charges. With Win32 Telephony, a toll list can be built in one of two ways:

1. The user can add or remove prefixes manually with the Telephony Control Panel.
2. The user can add or remove prefixes dynamically after a telephone call to that prefix fails for one of these reasons: When the call was dialed, the "1" prefix was missing and necessary or present and unnecessary. This dynamic process works as follows:

   The application knows by the value of the LINETRANSLATERESULT_INTOLLLIST and LINETRANSLATERESULT_NOTINTOLLLIST bits in the **LINETRANSLATEOUTPUT** structure (returned by **lineTranslateAddress**) whether an address with the dialed prefix is already in the toll list. The application can then let the user add or remove (whichever applies) this prefix from the toll list. Adding and removing are both performed using the **lineSetTollList** function.

Dialing the Call

The **lineMakeCall** function first attempts to obtain a *call appearance* on an address on the line, then waits for a dial tone, and finally dials the specified address. A call appearance is a connection to the switch over which a call can be made. Once the connection is established, the call appearance exists, even if no call is placed. After the call is established, the call appearance remains in existence until the call transitions to the idle state. If calls controlled by other applications exist on the line, these calls would normally have to be on hold, and would typically be forced to stay on hold until the application either drops its call or places it on hold. If dialing is successful, a handle to a call with *owner* privileges is returned to the application.

Before invoking **lineMakeCall**, an application can set up parameters for the call and store them in the data structure **LINECALLPARAMS**. A pointer to this structure is then used as a parameter of **lineMakeCall**. In the fields of **LINECALLPARAMS**, the application can specify the quality of service requested from the network as well as a variety of ISDN call setup parameters. If no **LINECALLPARAMS** structure is supplied to **lineMakeCall**, a default POTS voice-grade call is requested with a set of default values. However, it is a good idea to use **LINECALLPARAMS** so that monitoring applications can report this call information (such as the identification of the called party) accurately.

The call's origination address also appears in **LINECALLPARAMS**. Using this field, the application can specify the address on the line where it wants the call to originate. It can do so by specifying an address ID, though in some configurations it is more practical to identify the originating address by its directory number.

> **Note** Do not mix function calls of the Telephony API with the functions of Assisted Telephony. The actions requested by **lineMakeCall** would happen automatically with the Assisted Telephony function calls **tapiRequestMakeCall**. But once an application has reached this state by using the calls of the Telephony API, it makes no sense to revert to an Assisted Telephony function call (such as **tapiRequestMakeCall**), because doing so would cause TAPI to repeat already performed actions. At this stage, therefore, simply calling **lineMakeCall** causes less overhead.

Once dialing is complete and the call is being established, it passes through a number of different states. These states (the progress of the call) are provided to the application with LINE_CALLSTATE messages. This mechanism lets the application track whether the call is reaching the called party. It is important that every telephony application base its behavior on the information received in these messages, and not on any other assumptions about a call's state. An application must not assume that a requested state change has occurred until notification of that state change arrives. Note that it can be helpful to display user-friendly interpretations of call states as indicators of a call's progress, especially for calls expected to pass through states slowly.

If special call setup parameters are to be taken into consideration, the application must supply them to **lineMakeCall**. Call setup parameters are required for actions such as the following:

- Requesting a special bearer mode, bandwidth, or media mode for the call
- Sending user-to-user information (with ISDN)
- Securing the call
- Blocking sending of caller ID to the called party
- Taking the phone offhook automatically at the originator and/or the called party

Calling Card Information

The importance of a *calling card* rests in the different dialing procedures required by the various calling cards. An application provides TAPI with the information it needs to display a dialog box, from which the user can choose a calling card. (The calling card numbered 0 means "use the default dialing rules for the country you are in.") The routine returns, among other information, the user's location (country and area code), the number of calling cards registered for this user in the registry, and the preferred calling card for that location. This calling card information is applied by the **lineTranslateAddress** function and does not happen automatically through **lineMakeCall**.

Typically, an application would prepare a menu choice for the user, such as whether to make a call with the default carrier, to override the default carrier and use a given calling code, or simply to use another specific dialing sequence.

Using Multiple Addresses Simultaneously

The dialable number format (dialing formats are described in Line Devices Overview) allows multiple destination addresses to be supplied at once. This ability can be useful if the service provider offers some form of inverse multiplexing by setting up calls to each of the specified destinations and then managing the information stream as a single high-bandwidth media stream. The application perceives this group of calls as a single call because it receives only a single call handle representing the aggregate of the individual phone calls.

It is also possible to support inverse multiplexing at the application level. To do this, the application would set up a series of individual calls and synchronize their media streams.

Delayed Dialing

The application can also use **lineMakeCall** to allocate a call appearance or to dial just part of the full number. Later, it can complete dialing using **lineDial**. When the number provided is incomplete, dialing some of the digits may be delayed by placing a ";" (semicolon) at the end of the number. The **lineDial** function is used in cases in which the application needs to send address information to the switch on an existing call, such as dialing the address of a party to which the call will be transferred.

> **Note** An application should make sure that *incremental dialing* (providing the number in small pieces) is supported before attempting to use it. This support is indicated by the LINEADDRCAPFLAGS_PARTIALDIAL bit in the *dwAddrCapFlags* field in the **LINEADDRESSCAPS** structure, which is returned by **lineGetAddressCaps**.

The main reasons for an application to use delayed dialing are if the **?** character appears in a dialable address or if the service provider does not support one or more of the call progress detection control characters. These characters, which can occur in a dialable address, are **W** (wait for dial tone); **@** (wait for quiet answer); and **$** (wait for calling-card prompt tone). These and all other characters used in address strings are discussed in greater detail in Line Devices Overview.

The provider indicates which "wait for" dial string modifiers it supports in the following bits in the *dwDevCapFlags* field within the **LINEDEVCAPS** structure returned by **lineGetDevCaps**:

- LINEDEVCAPFLAGS_DIALBILLING
- LINEDEVCAPFLAGS_DIALQUIET
- LINEDEVCAPFLAGS_DIALDIALTONE

The **?** can be placed in the string (either directly by the application or by the address translator with the function **lineTranslateAddress**) if it is known that the user needs to listen for an undetectable tone before dialing can proceed. Every provider should treat **?** as requiring the dial string to be "rejected."

The **lineTranslateAddress** function returns bits, in the *dwTranslateResults* field of the **LINETRANSLATEOUTPUT** structure, that indicate whether any of the four potentially offending modifiers occur in the dialable string output from that translation operation. These bits give the application an idea of whether the dialable string might need to be scanned for unsupported modifiers:

- LINETRANSLATERESULT_DIALBILLING
- LINETRANSLATERESULT_DIALQUIET
- LINETRANSLATERESULT_DIALDIALTONE
- LINETRANSLATERESULT_DIALPROMPT

If the application tries to send an unsupported modifier or a **?** to the provider, it receives an error indicating which offending modifier occurred first within the string:

- LINEERR_DIALBILLING
- LINEERR_DIALQUIET
- LINEERR_DIALDIALTONE
- LINEERR_DIALPROMPT

The application can choose to pre-scan dialable strings for unsupported characters. Or it can pass the "raw" string from **lineTranslateAddress** directly to the provider as part of **lineMakeCall** (or **lineDial** or any other function that passes a dialable address as a parameter) and then let the service provider generate an error to tell it which unsupported modifier occurs first in the string.

When the application is told (or finds) that an unsupported dial modifier is in the dialable string, it must

take the following steps:

1. Locate the offending modifier in the string.
2. Isolate the characters occurring in the string to the left of the offending modifier.
3. Append a semicolon to the end of the partial string.
4. Reissue the dialing command using the partial string.
5. Prompt the user to listen for the audible tones indicating when it is OK to proceed with dialing.
6. Reissue the remainder of the dialable string (the portion following the offending modifier).

Note that in step 6 it is possible for another error to occur, because it is possible for multiple unsupported characters to occur within a single dialable string. Therefore, the application should repeat this process to dial the number in stages.

Tracking Asynchronous Requests

The function **lineMakeCall** is one of many functions that operate asynchronously. If an application manages only one line, it can use one *state record*, a struct containing all the information needed to track one outstanding asynchronous request. State records contain information such as the Request ID, the type of request, and pointers to allocated data that may need to be freed later.

But some applications have to manage more than one line—for example, to manage more than one outgoing call at the same time. To do so, they need to track the different asynchronous requests possible on each of those lines, and must therefore create an array of state records, one for each of the outstanding asynchronous operations (such as multiple invocations of **lineMakeCall**).

When a reply arrives that shows an asynchronous function has completed, the application matches the incoming Request ID with a Request ID in the array, and does whatever that specific call needs at that point. You can design one function to handle the reply that indicates completion of the outstanding asynchronous **lineMakeCall** request. The success or failure of the asynchronous request is recorded in the **requestResult** field of the **state** struct.

Call Handle Manipulation

When an application makes a call, a handle to the call with owner privileges is returned to the application. When the application is notified about an inbound call, it is given a handle to the call with either owner privilege or monitor privilege, depending on the privilege previously requested with **lineOpen**. It can also receive a handoff from another application, in which case it would receive owner privilege.

An application's call handle and associated privileges remain valid until the application takes an explicit action to change them or if it receives a LINE_CLOSE message, which closes the line. In this case, all handles to calls on the line instantly become invalid.

After a call reverts to the *idle* state, the application is still allowed to read the call's information structure and status. When the application has no further use for the call (and its information), it should deallocate the call handle by invoking **lineDeallocateCall**, which is discussed in following sections.

For more information about call logging and handing off calls to other applications, see Multiple-Application Programming.

Dropping Calls

To terminate a call, the application uses **lineDrop** on the call. This has the effect of hanging up on (disconnecting) the call, which makes it possible to make another call on the line. The **lineDrop** function is also used to abandon a call attempt in progress. If the remote party disconnects a call, the local application receives a LINE_CALLSTATE message with a call state of *disconnected*. If the local application disconnects a call, the call becomes *idle*, but its handle is not automatically deallocated (the application must call **lineDeallocateCall**). An application should check the **dwCallFeatures** field in **LINECALLSTATUS** to determine whether or not it is legal to invoke **lineDrop** at a particular time.

Deallocating Call Handles

A call handle remains valid after the call has been dropped. This enables applications to use operations such as **lineGetCallInfo** to retrieve call information for logging purposes. Once an application knows it has all the information it needs about a call and it has received a LINE_CALLSTATE(LINECALLSTATE_IDLE) message, it should call **lineDeallocateCall** to free system-allocated memory related to the call. The application must itself free memory that it allocated for its own purposes.

The way to free an idle call is to deallocate its handle with **lineDeallocateCall**. The application's duty to free a call is independent of the reason the call went idle. That is, the handle must be deallocated whether it was the local or the remote application that dropped it. If an application is the owner of the call, it can deallocate the call's handle only if the call is in the idle state. If monitoring the call, it can deallocate the call handle at any time.

It is better to process the LINE_CALLSTATE(LINECALLSTATE_IDLE) message (and all other call-state notifications) consistently in one location regardless of its cause.

Reclaiming Memory Resources

The TAPI dynamic-link library allocates memory for each call for each application that has a handle to the call. It is likely that service providers will allocate memory to hold call information as well. Deallocation of an application's call handle allows the library and the service provider to reclaim these memory resources. An application's handle for a call becomes void after a successful deallocation.

An application's attempt to deallocate the handle of a non-idle call for which it is the only owner will fail. The application should either first hand off ownership and change its privilege to monitor, or simply try to change its privilege to monitor (in case there are other owners) or clear the call by dropping it—which places the call into the *idle* state—and then deallocate its handle.

Closing Line Devices

After an application is finished using a line device, it should close the device by calling **lineClose** on the line-device handle. After the line has been closed, the application's handle for the line device is no longer valid. A LINE_LINEDEVSTATE message is sent to other interested applications to inform them about the state change on the line.

In certain environments, it may be desirable for a line device that is currently open by an application to be forcibly reclaimed (possibly by the use of some control .) from the application's control. This feature can be used to prevent a single misbehaved application or user from monopolizing a line. It is also used when the user wants to reconfigure the line parameters, and has told the service provider directly through its Setup function in the Telephony Control Panel that the provider should forcibly close the line. When this occurs, an application receives a LINE_CLOSE message for the open line device that was forcibly closed.

Exiting Telephony

The **lineShutdown** function disconnects the application from the Telephony API. If this function is called when the application has lines open or calls active, the call handles are deleted and the equivalent of a call to the **lineClose** function is automatically performed on each open line. (It is better for applications to explicitly close all open lines before invoking **lineShutdown**.) If shutdown is performed while asynchronous requests are outstanding, those requests are canceled.

An application that has registered as an Assisted Telephony request recipient should de-register itself by calling **lineRegisterRequestRecipient**, using the value FALSE for the *bEnable* parameter.

Setting a Terminal for Phone Conversations

The user's desktop computer may have access to multiple devices that can be individually selected and used to conduct interactive voice conversations. One of these devices is the telephone itself, complete with lamps, buttons, display, ringer, and a voice I/O device (handset, speakerphone, or headset). The user's computer may also have a separate voice I/O device (such as a headset, or microphone/speaker combination attached to a sound card) for use with phone conversations. TAPI enables the user to select where to route the information sent by the switch over the line, address, or call. The switch normally expects this destination to be one of its phone sets, and sends ring requests, lamp events (for stimulus phones), display data, and voice data as appropriate.

The phone in turn sends hookswitch events, button press events (for stimulus phones), and voice data back to the switch. The *line* portion of TAPI makes lamp events, display events, and ring events available, either as *functional* return codes to TAPI's various operations or as unsolicited functional call-status messages sent to the application. TAPI's implementation is responsible for mapping the functional API level to the underlying stimulus or functional messages used by the telephony network. In functional telephony environments, TAPI's functions are mapped to the functional protocol.

Modes of Operation: Functional and Stimulus

The *functional* mode of operation differs from the *stimulus* mode in the way meaning is attributed to events. For example, a given telephone has a button labeled "Transfer." When this button is pressed, one of two things can happen: the phone can send a message to the switch stating that the Transfer button was pressed, or it can send a message stating that "button number 18" was pressed. In the functional model, the button's function is indicated. It allows more flexibility in the phone hardware, because the switch doesn't need to know anything about the layout of the buttons, but the telephone will likely be more expensive, because it has more intelligence.

The stimulus model means that the event is simply indicated in a more raw, hardware fashion, such as by button number—even down to separate button-up and button-down events. In a stimulus-based system, telephones can cost less, but more intelligence is required in the switch so that it can recognize different types of telephones and translate their buttons into features. The stimulus model can provide more flexibility because different people can configure their phone buttons to mean different things through switch programming rather than by changing the phone itself.

Event Routing

Although not described in Supplementary Line Functions, event routing is a part of the supplementary line services and is not a basic function.

With the **lineSetTerminal** function, the application can control or suppress the routing of specified low-level events (exchanged between the switch and the station) to a device. With **lineSetTerminal**, the application specifies the terminal device to which these events (such as line, address, or call media-stream events) are routed.

The routing of the different classes of events can be individually controlled, allowing separate terminals to be specified for each event class. Event classes include lamps, buttons, display, ringer, hookswitch, and media stream.

For example, the media stream of a call (voice, for example) can be sent to any transducer device if the service provider and the hardware is capable of doing so. In general, a *transducer* means the same as what is referred to as a *hookswitch* device in the Telephony Phone API—something that has a microphone and a speaker. Ring events from the switch to the phone can be mapped into a visual alert on the computer's screen or they can be routed to a phone device. Lamp events and display events can be ignored or routed to a phone device (which appears to behave as a normal phone set). Finally, button presses at a phone device may or may not be passed to the line. In any case, this routing of low-level signals from the line does not affect the operation of the line portion of TAPI, which always maps low-level events to their functional equivalent. To determine the terminals a line device has available (and their capabilities), consult the line device's capabilities with **lineGetDevCaps**.

Assume initially that the application has suppressed the routing of all events (with **lineSetTerminal**), and the user selects a headset as the current I/O device. An incoming call sends a LINE_CALLSTATE message, and a LINE_LINEDEVSTATE message with the *ringing* indication. Because routing of all events is suppressed, ring events are not routed to the phone, so ringing is suppressed. Instead, the application notifies the user with a pop-up dialog box and a system beep in the headset.

The user decides to answer the call. Because the user's current I/O device is the headset, the telephony application invokes **lineSetTerminal** on the incoming call to route the call's media to the headset and answer the call. The application may also invoke **lineSetTerminal** to route lamp and display information events to the phone set so that it will behave as usual.

As a second example, assume that an incoming call is alerting at the user's computer. Instead of selecting the answer option with the mouse, the user decides to just pick up the phone's handset to answer the call. The offhook status at the phone sends a message to the application. The application can interpret this status as a request by the user to select the phone handset to conduct the conversation. The application then invokes **lineSetTerminal** to route the voice data on the call to the phone set.

Service Dependencies

Take care to ensure accurate listing of service dependencies among TAPI (specifically, the Telephony Service−TAPISRV.EXE), other service applications that use TAPI, and telephony service providers (TSP) that use other services.

The installation program for the service application or telephony service provider must record these dependencies with the Service Control Manager.

Note Failure to list TAPI as a dependency of the service application or failure to list another service as a dependency of TAPI can result in the system hanging.

List "Telephony Service" as a dependency of any service application that initializes a TAPI line or phone function. When TAPI is a dependency of a service application, the installation program must include "Telephony Service" in the list of service names passed to the *lpDependencies* parameter of the **CreateService** function.

When another service is activated by the TSP during the service provider startup (during **TSPI_providerEnumDevices**, **TSPI_lineNegotiateAPIVersion**, or **TSPI_providerInit**), the service started by the TSP must be listed as a dependency of the "Telephony Service." As a service that starts dynamically, TAPI starts all TSPs during its startup, and it is critical for the Service Control Manager to know when any service provider starts another service during TSP startup.

Call the **QueryServiceConfig** function to determine the existing configuration of "Telephony Service," including dependencies. If the service or services started by the TSP are not already included in the dependencies of the "Telephony Service", add the necessary items to the dependency list and call **ChangeServiceConfig** to update the dependencies.

For additional information about changing a service configuration, see Changing a Service Configuration in the *Microsoft Win32 Programmer's Reference.*

Multiple-Application Programming

Win32 Telephony applications may be designed to cooperate with each other, or they may act independently of one another. These TAPI applications will at times operate with non-TAPI applications built to support TAPI functionality, such as media-stream control applications. All these applications must be able to work together, or at least function independently in a cooperative way. To achieve this, TAPI defines mechanisms that let applications coordinate their telephony and phone activities while maintaining a high degree of flexibility.

The roles played by Telephony's major components are described in various topics of this section. Application writers can not only learn about TAPI's functioning from these sections, but can apply that knowledge directly when designing TAPI applications. For example, the Unknown application (defined in the following section) performs specific duties in media-mode probing and call handoffs. It is important to note and understand these duties before writing an Unknown application.

Event-Driven Environment

Like all Win32 applications, Win32 Telephony applications operate in the event-driven model. In Telephony, the most important events are call-state transitions. The service provider and TAPI dynamic-link library report call states to applications for particular calls. Interested applications, which know the previous state of the call, could infer call-state transitions from call states.

All running applications receive information—call-state messages—about all the calls in which they are interested. At times, several applications will have interest in the same call or calls, as monitors or owners of those calls. Incoming call-state events often cause applications to take actions on calls, and because those actions sometimes involve a shared call, one application must react to the knowledge that another has taken a certain step. Examples include the case where one application shows interest in owning (having control of) a call currently owned by your application, or drops a call co-owned by your application.

The Telephony system was designed to minimize *race* situations—in which the timing of competing function calls from different applications makes a difference. Awareness of the principles and the guidelines described in this section should help minimize possible competition.

Definitions

The following concepts are important for understanding the material presented in this section:

- **Initial media modes**. On a network other than ISDN, service providers usually do not know the media mode of an arriving call. For such calls, the service provider indicates a number of *initial* media modes, from which the correct one is eventually selected during the next step—the probing process. The initial media mode(s) identified by the service provider as being possible on the call will be reflected in the **dwMediaMode** field of the **LINECALLINFO** data structure, which the application can obtain by calling **lineGetCallInfo** after the initial LINE_CALLSTATE message announcing a new call is received.
- **Call control**. Having *control* of a call means that the application has received a LINE_CALLSTATE message stating that it has become an owner of the call. With this event, the application acquires a handle to the call with owner privileges. A way for a monitoring application to obtain call control (ownership) is to call **lineSetCallPrivilege** to set its call privilege to owner.
- **Call monitor**. An application that has a handle to a call with monitor privileges is a *monitor* of that call. Such an application cannot control the existence or other aspects of the call, but it can record (log) facts about the call. The application can also reset its call privilege to owner, thus becoming an owner of the call, in the event that the application determines (by monitoring media modes or other events) that it should take control.
- **Call owner**. An application that has control of a call is an *owner* of that call. An application can become an owner of a call in several ways, all discussed in this section. A call can have several owners simultaneously, although the usual situation is for only one application to be the owner of a call.
- **Probing**. *Probing* is the sending of signals on the phone line as an attempt to determine an incoming call's media mode. This search for the media mode is conducted only by applications. Some service providers may also be configured to do some amount of probing automatically to narrow down the initial media modes reported on a new call.
- **The Unknown application**. An application that has opened a line requesting ownership privilege for calls of as-yet undetermined, or UNKNOWN media mode, is referred to as the *unknown* application. (The LINEMEDIAMODE_UNKNOWN bit is set in the **dwMediaMode** field of the **LINECALLINFO** structure, along with other bits indicating other modes that are potentially present on the call.) An application that does this may actually be capable of handling calls of a number of different media modes. Alternatively, it may simply act as a traffic director, passing calls on to other applications that can use calls of the specific media modes.

Call Ownership

The mechanism with which applications control calls is based on the concept of *ownership*. At any given time, one or more applications can own a call. While an application has ownership of a call, it is allowed to manipulate the call in ways that affect the state of the call. An application that does not own a call (but has a handle to it) is a *monitor* of the call and is prevented from manipulating it. It can only perform status- and information-query operations on that call. While one or more applications are owners of a call, still other applications can be monitoring the call.

Ownership of a call is assigned to applications according to the following rules:

- An application that makes an outgoing call is the initial sole owner of that call. Other applications monitoring the line will be informed of the outgoing call at the time the first LINE_CALLSTATE message is received. Usually, this notification occurs when dial tone is initially detected.
- Ownership of an incoming call is assigned to one application only. This assignment avoids the situation in which, depending on timing, different applications may seize control at different times, causing unpredictable results.
- An application that is currently an owner of a call can pass ("hand off") ownership to another application that has the call's line open. While handing off ownership of a call, the original owner application can specify the new media type of the call. When the handoff succeeds, the original application remains an owner of the call, and it can then choose to either deallocate its handle (if it is no longer interested in the call), change to being a monitor (using **lineSetCallPrivilege**), or remain an owner (although doing so is discouraged). The original application's privilege is *not* automatically changed by **lineHandoff**. More information on the two types of call handoffs (*directed* and *media-mode*) can be found later in this chapter.
- If a target application for the handoff is found, and if it is already a co-owner of the call, it will see no effect caused by the handoff, although it will receive a LINE_CALLSTATE message. This message repeats the fact that it is an owner to alert it that another application has explicitly asked it to take control of the call. The application initiating the handoff is informed about the success of the handoff.
- If there is no target application for the requested handoff and the call is active, an error is returned. No handoff takes place.
- Handing off a call between applications never affects the state of the physical call as perceived by the switch or the service provider.
- An application that does not have (but wants) ownership of a call may request ownership. The application can select calls based on a number of criteria, ranging from all calls on a particular line or address (a phone number assigned to the line, using **lineGetNewCalls**), to calls related to a specified call (using **lineGetConfRelatedCalls**). An application that calls **lineGetNewCalls** or **lineGetConfRelatedCalls** will always receive a monitor handle. If it wants to become an owner of a call it receives, it must then call **lineSetCallPrivilege**. If it determines that it is not interested in one or more of the calls to which it receives handles using **lineGetNewCalls** or **lineGetConfRelatedCalls**, it must call **lineDeallocateCall** for each such handle to release the internal resources maintained to track the call ownership.
- Any application that asks for ownership receives it; any application that is offered ownership cannot refuse it. An application that becomes an owner through a handoff actually becomes a co-owner of the call. When the call is initially presented by the provider, the initial owner is the sole owner of the call.
- The originally owning applications are informed about the existence of every new owner. Monitoring applications are informed as well.

Note that with *co-owned calls* (calls simultaneously owned by more than one application), no protection is offered to prevent the applications from interfering with each other. For this reason, maintaining ownership after a handoff or after ownership is taken by another application is discouraged.

Because media streams are not managed by the Telephony API, call handoff does not handle the handoff

of the call's media stream. Media-stream handoff must be carried out using commands from an appropriate media-control API or directly coordinated between the applications involved.

Handling Incoming Calls

When multiple applications are running simultaneously, an appropriate one must be found to become the initial owner of each incoming call. In general, incoming calls reach their destination, or *target* application, in two or three steps: First, the service provider learns of the new call and passes it to the TAPI dynamic-link library which gives the call to the appropriate application. Finally, applications conduct probing, if necessary, which can cause the call to be handed off between applications one or more times. These steps are described in the following topics. Sometimes applications perform further probing, a case which is also covered in the following topics.

Duties of the Service Provider

The service provider determines the media modes.

Determining Initial Media Modes

When a service provider learns of the appearance of a call, its first task is to determine the call's media mode to the best of its ability. (It has received ringing voltage on a POTS line or, in the case of EPBX or ISDN, a protocol message indicating that a call is incoming.) It may be able to tell the single correct media mode or it may only be able to narrow down the possibilities to a certain few. These first media mode settings are called *initial media modes*, and the following are the considerations used for setting initial media mode bits:

- **Service provider setup** The service provider has been configured to work with only a single media mode or certain media modes.
- **Hardware limitations** Hardware limitations are usually reflected in the service provider's configuration, but the media modes could be further restricted by a particular card in use.
- **Call to lineOpen** Possible media modes are limited by what applications have requested in their invocation of the **lineOpen** function. TAPI combines all of the media modes requested by applications and indicates the sum of them to the service provider in a call to **TSPI_lineSetDefaultMediaDetection**. For example, a telephony device and its service provider may be able to handle Group 3 fax calls, but if no application is running to handle such calls, the provider would know not to bother with probing for fax or reporting fax calls to TAPI. (The TAPI dynamic-link library does not automatically launch an application to handle a particular type of incoming call.)
- **Caller ID and direct Inward Dialing** With Direct Inward Dialing (DID) at the called address, the switch supplies the service provider with the digits that were dialed (the called address). The service provider can be configured to associate particular called addresses with particular media modes. Likewise, it could associate calls from particular numbers as being associated with particular media modes, although this is much less commonly used.
- **Distinctive ringing** The ring pattern of the incoming call may match a predetermined pattern (of several possible at the called address) that is reserved for calls of a certain media mode. If, for example, the incoming call is using ring pattern 2, the service provider knows it to be a fax call (based on configuration information supplied by the user).
- **ISDN** On an ISDN network, the provider may analyze the call's protocol frames to determine the media mode. If the call is indicated as a 3.1 kHz Voice call, it is still possible that the actual media mode on the call is analog data modem, Group 3 fax, text telephone, or any of several other voiceband modulated signals, in addition to human voice; it is only with digital data signals that the media mode would necessarily be clearly defined—for example as Group 4 fax—at call setup time in ISDN.
- **Auto answer and probe** Some providers give the user an option to let the service provider *autoanswer* the call and do some of the probing itself. In this process, TAPI gives the call the correct application with the correct media mode already identified.

These tools may be enough to make a final and accurate determination of the media mode. In any case, when the service provider passes the new call to TAPI, it sends a LINE_CALLSTATE message and includes in the message all that it knows about the call's media mode(s). The following topics give details on the possible cases.

Known Media Mode

When the service provider knows the media of the call unambiguously, one flag is set in **dwMediaMode** in **LINECALLINFO**. The media mode cannot be the single bit LINEMEDIAMODE_UNKNOWN, which is a different scenario. TAPI gives ownership of the call to the highest priority application that has opened a line for this media mode. It also gives call handles with monitor privileges to all other monitor applications on the line.

Unknown Media Mode

Even when the service provider does not know the exact media mode of the call, it might still know which media modes are possible. In this case, the service provider sets a combination of likely media mode bit flags, including LINEMEDIAMODE_UNKNOWN and passes the call to TAPI. The service provider sets these bits both in the **dwMediaMode** field of the **LINECALLINFO** record and in the *dwParam3* parameter of the first LINE_CALLSTATE message it sends to TAPI.

The service provider considers only the media modes for which applications have opened the line with owner privileges (it becomes aware of these media modes through the **TSPI_SetDefaultMediaDetection** call) and which it is capable of handling. TAPI informs the provider about the union of all the lines that have been opened with a specified media mode. The provider can use this union to enable only the appropriate media mode detections for which applications are interested. If no applications have opened the line for ownership, the provider will not consider any media modes. Incoming calls are still delivered to TAPI, but no initial owner is possible. In this case, monitoring applications will still be informed of the call, and if none of them changes their privilege to owner and answers the call, the call will remain unanswered.

Duties of the TAPI Dynamic-Link Library

The TAPI dynamic-link library does not perform probing; that is, it does not send signals on phone lines in order to determine a call's media mode. TAPI does try to deliver the call to an application that can do probing. The way in which TAPI gives calls to applications is determined by several factors, the most important of which are the media mode bits that have been set and the running applications that can (or cannot) handle calls of those media modes.

TAPI's behavior can be divided into two main scenarios: one media mode bit is set, and the UNKNOWN bit is set. These cases are described in the following topics.

Only One Media Mode Bit Is Set

If only one media mode bit (*not* the UNKNOWN bit) has been set in the **dwMediaMode** field of the **LINECALLINFO** data structure, TAPI distributes calls by following a consistent procedure based on the current state of the system and on information saved by the user in the registry. These are the steps it takes:

- The TAPI dynamic-link library is notified by the service provider that a call is arriving.
- The TAPI library uses the information in the HandoffPriorities section of the registry to know which applications have been listed—possibly through a Preferences option in the application's user interface—as being interested in calls having the incoming call's media mode.
- The first such application listed, reading left to right, is the highest priority application. If that application is currently running and has the arriving call's line open for that media mode, it is given ownership of the call. If it is not running or it does not have that line open, TAPI again uses the information in the registry to find an interested application in the correct state, and it gives the call to it.
- If none of the applications listed in the registry are in the proper state, TAPI looks for other applications that are currently executing and have the line open for that media mode (though they are not listed in the registry). The relative priority among these unlisted applications is arbitrary and not necessarily associated with the sequence in which they were launched or opened the line.
- Every application that has the line open for monitoring also receives a handle to a call, and any of them could step up, claim ownership (by calling **lineSetCallPrivilege**), and answer the call. However, this behavior could result in *race conditions* and unpredictable call handling, and is therefore discouraged.
- If no application becomes an owner of the call, the call is eventually dropped. Calls can be dropped by TAPI only if no owner is found for the call and the call state is not *idle* or *offering*. The calling party can also drop the call. (On an ISDN network, this event becomes known when a "call-disconnect" frame is received.) If the call is not explicitly dropped, it can go idle after the expiration of a timeout based on the absence of ringing. (The service provider would need to assume that the call has been dropped by the calling party, and implement the timeout.) Because there were no applications that could take the call successfully, this situation usually means that the incoming call reached a wrong number.

The UNKNOWN Bit Is Set

If the LINEMEDIAMODE_UNKNOWN bit is on in *param3* of the first LINE_CALLSTATE message delivered by the service provider, the call is treated differently depending on whether an application prepared to accept calls of unknown media type has opened the line. These two possible cases (an Unknown application is running, or is not) are described in this section.

An Unknown Application Is Running

If at least one Unknown application has opened the line, the TAPI dynamic-link library gives an ownership handle for the incoming call to the highest priority Unknown application. It also passes monitoring handles to the other applications that have the line open for monitoring. The Unknown application receives a LINE_CALLSTATE message with *dwParam3* set to owner.

This Unknown application can then try to perform media determination itself, or use the assistance of the other media applications, allowing them to perform probes for their media mode(s), if appropriate. The Unknown application can pass the call to another media application using **lineHandoff**. The Unknown application would examine **dwMediaMode** in **LINECALLINFO** to determine the possible remaining candidate media. In doing so, it uses the highest priority media to determine the initial handoff target. It calls **lineHandoff**, specifying the single highest priority destination media mode as the target.

The following is the default priority of media modes, listed in order from first tried to last tried when used during media-type handoffs.

Order	Media Mode
1	LINEMEDIAMODE_INTERACTIVEVOICE
2	LINEMEDIAMODE_DATAMODEM
3	LINEMEDIAMODE_G3FAX
4	LINEMEDIAMODE_TDD
5	LINEMEDIAMODE_G4FAX
6	LINEMEDIAMODE_DIGITALDATA
7	LINEMEDIAMODE_TELETEX
8	LINEMEDIAMODE_VIDEOTEX
9	LINEMEDIAMODE_TELEX
10	LINEMEDIAMODE_MIXED
11	LINEMEDIAMODE_ADSI

Automated voice is a media mode that has no meaningful distinction with interactivevoice at this level, and is therefore not listed.

If the handoff fails, the Unknown application should clear that media mode flag in the **dwMediaMode** member of **LINECALLINFO**. This action moves the probe for the call one step closer to a final determination of the media mode. If the handoff indicates TARGETSELF, it means that the Unknown application is the highest priority application for the media mode for which it was trying to hand off the call, so it should go ahead and do the probing itself.

If the handoff indicates SUCCESS, it means that a different application is the highest priority application for the media mode for which the call was being handed off. The Unknown application should deallocate the call handle or change to being a monitor while the new owner has control and proceeds with probing.

The receiving application controls the call. If the probe is successful, it should set the correct media mode bit. If the probe fails, the application should clear the failed media mode bit in **LINECALLINFO** and hand the call off to the next highest priority application. If no more media mode bits are set, the handoff fails, because no suitable owner application exists for the call.

Eventually, the media mode may be identified through monitoring or successful probing, though the UNKNOWN bit still may be set in **dwMediaMode** in the data structure **LINECALLINFO**. In this case, the application that has received the call cannot be sure that it is the highest priority application for the identified media mode. It is now the duty of that application to ensure that the call goes to the highest priority application. To do so, it follows these steps:

- It calls **lineSetMediaMode**, which writes into the **dwMediaMode** field of the call to turn off the UNKNOWN bit and specify the newly identified media mode bit.
- It calls **lineHandoff** to return the call to TAPI. The TAPI dynamic-link library is not explicitly specified in this command, but rather a media-type handoff is performed, through which the TAPI library knows that it must look for other applications to find the highest priority application for that media mode.
- If this application is itself the highest priority application for this media mode, it receives a LINEERR_TARGETSELF return value (for the **lineHandoff** function call). This error means "No, you already are the highest priority application for that media mode." The application never loses control of the call, and it continues handling the call normally. If the **lineHandoff** succeeds, then there was a higher priority application for the identified media mode, and the application that called **lineHandoff** should deallocate its handle or change to being a monitor while the highest-priority application handles the call.

As long as the UNKNOWN bit is still on, the receiving application still does not know that the highest-priority media mode is present on the call, so it still needs to probe for it. It only considers the media mode to be present if the UNKNOWN bit is off—only then can it use the call as a call of that media mode.

Using Media Priorities While Probing

Unknown applications should take care to use default priorities given in the table in the preceding topic, An Unknown Application Is Running, when probing for applications to take calls of unknown media modes. One reason to do this is to protect human callers from hearing unpleasant fax or modem signals. If, for example, both the INTERACTIVEVOICE and the G3FAX bits are set in LINEMEDIAMODE_, a human caller may still be on the other end of the line. The application should wait to start probing for a fax (with a fax tone) until it is sure that the call is not a voice call. The way to be sure is to probe first for voice, which occurs automatically if following the order stated in the default media-mode list.

However, while probing for high-priority media modes, it is a good idea to turn media monitoring on. This feature, invoked by calling **lineMonitorMedia**, detects signals that indicate other media. For example, one application may be playing an outgoing "leave a message" voice message while the incoming call starts sending a fax "calling" tone and waits for a handshake. In order not to lose the fax call, the local application needs to be monitoring for this tone while playing the voice message. Determining the lower-priority media (the fax call) while actively probing for the higher-priority media (voice) is not only a safer method—it helps prevent the loss of a call—it is efficient because it can shorten the probing process.

No Unknown Application is Running

If no Unknown application has the line open, the TAPI dynamic-link library itself assumes the role of a simplistic Unknown application. The TAPI library first passes an owner handle for the call to the highest priority application that is registered for the highest-priority media mode for which a media mode flag is set in the **dwMediaMode** member of **LINECALLINFO**. If there is no such application, the media mode flag is cleared, and TAPI tries the highest priority application for the next highest-priority media mode. If there is such an application, it can try to make a determination for the highest priority media mode in **dwMediaMode** in **LINECALLINFO**.

If no application is found to become the initial owner of a call, the call remains in the *offering* state until a monitor application becomes an owner through **lineSetCallPrivilege**, or until the call is abandoned by the calling party and is transitioned (by the service provider) to the *idle* state, at which time all monitoring applications deallocate their handles to the call.

Duties of the Media Application

The media mode application that receives a call as a handoff target first checks the bit flags of **dwMediaMode** in **LINECALLINFO**. If only a single media mode flag is set, the call is officially of that media mode, and the application can act accordingly.

If the UNKNOWN and other media mode flags are set, the media mode of the call is officially UNKNOWN, but is assumed to be one of the media modes for which a flag is set in **LINECALLINFO**. The application should now probe for the highest priority media mode.

If more than one bit is set in **LINECALLINFO** and the call has not been answered, the application must perform a **lineAnswer** to continue probing. If the call has already been answered, the application can continue probing without having to first answer the call.

If the probe succeeds (either for the highest-priority media mode or for another one), the application should set **dwMediaMode** in **LINECALLINFO** to the single media mode that was recognized. If the actual media mode is this expected media mode, the application can act accordingly. Otherwise—if it makes a determination of another media mode—it must first attempt to hand off the call in case it is not the highest priority application for the detected media mode.

If the probe fails, the application should clear the flag for that media mode in **dwMediaMode** in **LINECALLINFO**, and hand the call off to the Unknown application. It should also deallocate its call handle or revert back to monitoring the call. At this point, the fate of the call is determined by the steps described under the preceding topics of this section—depending on whether or not an Unknown application is active.

If the attempt to hand off the call to the UNKNOWN application fails, this means that no unknown application is running. It is then the responsibility of the application that currently owns the call to attempt to hand it off to the next-highest-priority media mode (while leaving the UNKNOWN bit turned on in **dwMediaMode** in **LINECALLINFO**). If that handoff fails, the application should turn off that media bit, and attempt the next higher-priority bit, until the handoff succeeds or all of the bits are off except for the UNKNOWN bit.

If none of the media modes were determined to be the actual one, only the UNKNOWN flag will remain set in **dwMediaMode** in **LINECALLINFO** at the time the media application attempts to hand the call off to UNKNOWN. The final **lineHandoff** invocation will fail if the application is the only remaining owner of the call. This failure informs the application that it should drop the call and then deallocate the call's handle. At this point, the call is abandoned.

Receiving Incoming Calls

Once the target application has been determined, it is given the call. The following topics discuss the target application as it accepts and answers calls.

Accepting and Answering Calls

On a POTS network, the only reason for an application to call **lineAccept** is to inform other applications that it has accepted responsibility to present the call to the user. Similarly, on an ISDN line, the effect of accepting a call is to make other applications aware that some application has accepted responsibility for handling the call.

On an ISDN network, accepting a call also informs the switch that the application will present the call to the user (by alerting the user for example, by ringing or by popping up a dialog box). If the LINEADDRCAPFLAGS_ACCEPTTOALERT bit is set, the application *must* perform a **lineAccept** on the call or the call will not ring. If the application fails to call **lineAccept** quickly enough (the timeout may be as short as three seconds on some ISDN networks), the network will assume that the station is powered off or disconnected and act accordingly, such as by deflecting the call (if Forward on No Answer is activated) or sending a disconnect message to the calling station.

Accepting a call is not the same as answering a call. Answering calls, in POTS, simply means to go offhook. On an ISDN line, it means to tell the switch to place the call in a connected state. Before answering, there is no physical connection for the call between the switch and the destination, though the call is connected from the caller to the switch.

Sometimes a call has already been answered when a new application takes control of it. This can occur, for example, when one application discovers that it is not the highest priority application for a call of a given media mode, and it hands the call off. If the first application has already answered the call, the receiving application takes control of an answered call. It should treat the call normally—that is, as if it had answered the call itself. Another example is when a user instructs an application to operate on an existing call. In this case, the application seizes the call. Again, it should treat the call as if it had answered it.

Waiting a Minimum Number of Rings

The **lineGetNumRings** function can be used by any application to determine the number of times an inbound call on the given address should ring before the call is to be answered. Waiting a certain number of rings allows callers to be spared the charge of a call connection if it seems that the call will not be answered by the desired party (usually a person). This feature is sometimes called *toll-saver* support. Applications can use the functions **lineGetNumRings** and **lineSetNumRings** in combination to provide a mechanism to support toll-saver features for multiple independent applications.

Any application that receives a handle for a call in the offering state and a LINE_LINEDEVSTATE ringing message should wait a number of rings equal to the number returned by **lineGetNumRings** before answering the call in order to honor the toll-saver settings across all applications. The function **lineGetNumRings** returns the minimum number of rings any application has specified with the function **lineSetNumRings**. Because this number may vary dynamically, an application should call **lineGetNumRings** each time it has the option to answer a call—that is, when it is the owner of a call still in the *offering* state. A separate LINE_LINEDEVSTATE ringing message is sent to the application for each ring cycle.

If the service provider is set to auto-answer calls, it answers after a certain number of rings. Service providers do not have access to the minimum-ring information established by **lineSetNumRings**, and therefore will make their own determination of when to automatically answer an incoming call. When a call has been so answered by a service provider, it will be initially delivered to the owning application already in the *connected* state, so the application will not need to be concerned with counting rings or with answering the call.

Taking Ownership of a Call

In general, when one application learns that another application wants ownership of a call, it simply relinquishes ownership of the call to that other application. Although there can be many co-owners of a call, it should be a transitory state for there to be multiple owners.

In one specific case, it is valid for an application to actively take ownership of a call owned by another application. This is when the application is instructed to do so by the user—perhaps through a user interface. For example, a fax application may be instructed by a user to break into that same user's existing voice call and use the call to send a fax. In this case, the fax application takes ownership from the previous owner, the application that was controlling the voice call.

An application can forcibly become owner of a call by taking the following steps:

- Obtain a handle to the call with monitor privilege. If the desired call is one for which the application does not yet have a handle, it should request a handle with **lineGetNewCalls**. If the application is already a co-owner of the call and wants to become sole owner, it should start by calling **lineSetCallPrivilege** with the parameter *dwCallPrivilege* set to LINECALLPRIVILEGE_MONITOR. This action, which relinquishes ownership of the call (temporarily, in this case), is seen by other applications as the departure of an owner.
- Call **lineSetCallPrivilege** with the parameter *dwCallPrivilege* set to LINECALLPRIVILEGE_OWNER for the call. Other applications see a new owner coming on line by receiving a LINE_CALLINFO message stating that the number of owners has increased and the number of monitors has changed; the bit LINECALLINFOSTATE_NUMOWNERINCR is on. These applications should yield the call to the new owner but there is no guarantee that they will do so. If the other existing owners do relinquish ownership, the new owner can proceed with what it intended to do on the call.

Note There is no way to shield a call from another application's attempt to become an owner of it, nor is there any reason to do so. Once an application is informed that another application has become an owner, it should draw its activities on the call to an orderly close, and then relinquish ownership, because such changes in ownership are almost always done at the explicit direction of the user.

Relinquishing a Call

An application can relinquish ownership of a call by invoking **lineSetCallPrivilege** to change to a monitor application, or simply by using **lineDeallocateCall** to indicate that it has no further interest in the call. If the application is the sole owner of the call and cannot hand off ownership to another application, TAPI will not permit it to change to being a monitor or to deallocate its call handle—in this situation, the application has no choice but to drop the call.

Control of the Media Stream

An application that has just obtained a call may not immediately receive control of the media stream, and may need to wait until the previous owner application relinquishes it. Though this may take time, any application with control of the media stream should transfer control when it sees that a new owner has come on line (the number of owners has increased).

The procedures for transferring control of an active media stream differ for every media-control API. The API may allow only one application to have a media-stream device (such as a COM port used for data transfer) open. In this case, it is important that the current owner relinquish control of a media stream device *before* handing off the call. But with some other types of media such as WAV audio, several applications (and several devices) can have the media stream open at the same time. This makes it unnecessary to close the media stream before the handoff, and perhaps not at all.

Call Handoffs

After an application has acquired ownership of the call, ownership can be transferred to another application. Why would this be necessary? Normally, to allow the call's media mode to be changed. In this case, the highest priority application for the new media mode should take and handle the call. Media mode changing usually occurs because of one of the following causes.

User command. Through a user interface or through window messages, the application learns that the local user wants to change media mode. For example, the user has told the new target application (which is not yet an owner) to obtain an existing voice call for transmitting data. The target application must now take control of the call. In this case, the current owner notices the number of owners increase, and then relinquishes its control of the call. Alternatively, the user could instruct the current owner of the call to hand it off to an application that can handle the new media mode.

Media mode change. The service provider can detect a media mode change with **lineMonitorMedia**. As an example of this, the local application is playing a recorded voice message to the caller. During this message, the caller spontaneously decides to transmit a fax calling tone, and the local application can respond accordingly by changing the media mode to fax and, if necessary, handing the call off to a fax application. Another way this can work is for a monitoring application to enable media mode monitoring, and, when the media mode in which it is interested is detected on a call, it can request ownership of the call. This mechanism makes it unnecessary for every application to monitor every call for every media mode.

Remote party command. The remote party can interactively indicate a change in media modes during an existing call. For example, the local application is using **lineMonitorDigits** to monitor DTMF input by the remote caller. Through this monitoring, the caller indicates, for example, that a fax is about to be sent. Other ways the caller can control local applications is with commands received on other data connections and through ISDN user-user information messages.

A call handoff will have one of these outcomes:

- The call is given to another application (SUCCESS),
- The handing-off application is itself the target (TARGETSELF),
- The handoff fails (TARGETNOTFOUND).

If the application that is receiving the handed-off call already has a call handle to the call, this old call handle is used. Otherwise a new call handle is created. In either case, the application ends up with owner privileges to the call. Whenever the handing-off application is not the same as the target application, the target is informed about the handoff in a LINE_CALLSTATE message with *dwParam3* set to LINECALLPRIVILEGE_OWNER, as if it were receiving a new call.

> **Note** The LINE_CALLSTATE parameter *dwParam3* is set to owner only if the LINE_CALLSTATE message is being sent to an application that is the initial owner of a new call, is the target of a handoff, or was previously a monitor or an owner of the call. The parameter *dwParam3* can be set to monitor only if the LINE_CALLSTATE message is sent to the application when it is presented a new call for monitoring. In all other cases (such as when the application already has a handle to the call, and its ownership state is not being changed), *dwParam3* is set to 0.

If the current owner application is told to change modes, it does so by handing off the call to an application used for the target media mode. The two types of call handoffs are described in the following topics.

Directed Handoffs

A *directed handoff* takes place when the target application is known by name to the original application. This situation would occur, for example, among a set of applications written by the same vendor. Control of directed handoffs can usually be configured by the user. With such a handoff, the call is given to the specified application if it has opened the line on which the call exists. The media mode specified at the time the application opened the line is ignored. One common example is a voice call followed by fax transmission in the same call. Directed handoff would most often be used by applications from the same developer that are linked in other ways as well.

Directed handoff may also be used in future versions as part of the process of arbitrating multiple applications waiting for incoming calls of the same media mode, with the selection of the application to handle the call being based on data-link or higher level protocol detection rather than media mode. An example of its use would be an incoming data modem line with applications such as remote takeover, bulletin board, remote network access, and remote e-mail access all waiting for calls simultaneously.

Media Mode Handoffs

A *media mode handoff* takes place when there is a new, targeted media mode, usually when the owning application determines that the media mode needed for the call is not present or is about to change. The process for a media-dependent handoff can be a probing process if the UNKNOWN bit is on, and is virtually the same as for the initial assignment of a call to an application. The difference is the fact that **lineHandoff** can have only one media-mode bit set.

Because only a single media mode bit can be specified, the call is given to the highest priority application for that media mode. However, it is possible that more than one media mode is to be considered for the handoff. In this case, the handing-off application should specify the highest-priority of the possible media modes as a parameter for **lineHandoff**. If an applications specifies the UNKNOWN bit when performing a media-mode handoff and the handoff fails, this means that no Unknown application is currently running. The handing-off application should then try to hand the call off to the highest priority application registered for the next higher media mode.

The receiving application is now responsible for the call. It now probes for the call's actual media mode. If the call's media mode matches that handled by the application, it must ensure that it is the highest-priority application registered for that media mode. If so, it keeps the call and processes it normally. If not, it hands the call off to another application registered for that media mode.

If, however, the probe for that media mode fails, the application probes again, attempting the remaining media-mode possibilities. It determines these by examining the **dwMediaMode** field in the **LINECALLINFO** structure. But first, using the **lineSetMediaMode** function, the owning application turns off the bit for the current (disproved) media mode in the **dwMediaMode** field.

This process of probing and handing off continues, and the remaining media modes are eliminated one by one. Along the way, one of the applications may see that the media mode it handles is on the call, and the handoff is successful. The application should now perform a final **lineSetMediaMode** to set the correct media mode and clear all other media-mode bits. This informs other interested applications of the correct media mode. These other applications receive a LINE_CALLINFO message stating that the call's media mode has changed. To determine the correct media mode, they invoke **lineGetCallInfo** and examine the **dwMediaMode** member in the **LINECALLINFO** structure.

It is the responsibility of the owning application to cycle through media modes to find the highest-priority application. TAPI does this cycling only on the initial incoming call to find the first owner. It does not do it when **lineHandoff** is called.

To sum up the media-mode handoff process, TAPI *does not* check for other media bits during **lineHandoff**. TAPI only attempts to hand off to the *single* media mode indicated in the parameter to **lineHandoff**. It is up to the application to turn off the bit corresponding to the media mode that failed to hand off, and to try other media modes until the handoff succeeds or all the possible media modes are exhausted. If it gets to a point where all of the bits are off except for UNKNOWN, it must abandon the call by calling **lineDrop** and then deallocating the handle.

Logging Calls

Logging a call simply means actively recording the actions and states of a monitored call into a file on a disk. (An application that owns a call is also a monitor of that call.) Though limited call logging is performed by the call manager provided with Win32 Telephony, any application can be programmed to log a call that it is monitoring.

TAPI facilitates logging by allowing applications to monitor calls and by collecting information from other applications that are controlling calls. TAPI writes the information it collects into the **LINECALLINFO** structure, which can then be read by applications.

Sources of Log Information

Information useful to call logging (which is saved by TAPI) originates in the following sources, and is placed in the **LINECALLINFO** data structure.

- In an application's call of the initialization function, **lineInitializeEx**, it provides information such as the application's name. An application should ensure that the information it provides with the initialization command is accurate so that its activities are reflected correctly in the call log.
- The **LINECALLPARAMS** data structure, used in the function **lineMakeCall**, stores information such as the name of the called party, the originating address, and the destination address. This structure is passed to TAPI by any application that originates a call.
- The service provider supplies many items that can be logged such as the caller's ID.
- The LINE_CALLSTATE messages that pertain to a given call are an important source of logging information, indicating whether the call successfully connected or was abandoned because of a busy signal, no answer, network congestion, or other causes.

The start time, end time, and duration of a call are not recorded by TAPI. A logging application that wants to log this information must record the time by checking the system clock when certain LINE_CALLSTATE messages (such as CONNECTED, DISCONNECTED, and IDLE) are received, and then derive the related time log information.

Using the LINECALLINFO Data Structure

The **LINECALLINFO** structure stores a large amount of information about a call, and is thus an important source of data. Generally, a logging application reads from **LINECALLINFO** and write this information into its log file.

A separate **LINECALLINFO** structure exists for every inbound and outbound call. Information in this structure is obtained by an application with **lineGetCallInfo**. An application typically reads the information from **LINECALLINFO** at the following times:

- When the application first receives a handle for a call in the LINE_CALLSTATE message.
- Each time the application receives notification in a LINE_CALLINFO message that LINECALLINFO has changed. Whenever any part of the structure changes, a LINE_CALLINFO message is sent to the application indicating which information item has changed.

Both the LINE_CALLSTATE message and the LINE_CALLINFO message also supply the call's handle as a parameter.

Much of the information held in **LINECALLINFO** remains fixed for the duration of the call. Information about a call that changes dynamically, such as call progress status, is available in the **LINECALLSTATUS** structure, which is returned with the function call **lineGetCallStatus**. Other information needed by logging applications that is not stored in **LINECALLINFO** is call start time, call stop time, and the call's duration, which it determines by checking the system time when the corresponding LINE_CALLSTATE messages are received.

LINECALLINFO stays intact after the call is disconnected, so the logging application can later read it in order to write additional information into the log. **LINECALLINFO** remains available only until the last application that had a handle for the call (owner and monitor handles) deallocates its handle.

Deallocating a Call

If an application is finished with a call and another application wants the call, the call can be handed off. But if no other applications want to take ownership, there is nothing to do but deallocate the call's handle. This is done with **lineDeallocateCall**. A call handle is no longer valid after it has been deallocated.

In contrast, dropping (disconnecting) a call puts the call in the *idle* state, which means that the local end of the connection is on hook. If the other end of the connection drops the call, the call transitions to the *disconnected* state, not the *idle* state. Typically, once an application receives a call-state message indicating the *disconnected* state, it immediately drops the call, causing it to become *idle*. Although the call is in the *idle* state, any handles to it held by applications remain valid until they are deallocated. If the call was never answered (the local end never went off hook), it may revert to the *idle* state without being dropped.

An application cannot deallocate a call if it is the sole owner and the call's state is not *idle*. This is because TAPI tries to ensure that there is always at least one owner for every active call. If the application is the sole owner and the call is not *idle*, the error message LINEERR_INVALCALLSTATE is returned. If an application needs to circumvent this restriction, it can do so by dropping the call first (with **lineDrop**) and then deallocating its handle. This prevents an application from deallocating its handle which would result in a call disconnect. By making the application do an explicit drop, it can inform the user (in a dialog box) that the call is about to be disconnected.

If releasing the ownership handle results in the call's having no more handles, TAPI calls the service provider function **TSPI_lineCloseCall**. When this function is invoked on a call that is not yet idle, it is up to the service provider to drop the call.

> **Note** An application that has monitor privileges for a call can always deallocate its handle for the call. Deallocating a call does not affect the call state of the physical call, but it does release the internal resources (memory) related to the call.

An application should deallocate the handle to a call it owns in these two cases:

- **Idle call state**. If an application receives a LINE_CALLSTATE message indicating that the call has transitioned to the *idle* state, and has already gathered all the information it needs about the call, it should deallocate the call handle immediately.
- **Handoff**. The application has handed off the call (or has otherwise relinquished call ownership to another application) or has set its call privilege to monitor, and has no interest in monitoring or logging the call.

Failure to deallocate call handles in a timely way can result in system failure and lost calls due to unnecessary consumption of memory and other resources.

Closing Lines

An application should close a line it has open in the following cases:

- **Before exiting**. An application should always close all lines it has open before it becomes inactive.
- **For non-TAPI applications**. TAPI applications should cooperate with non-TAPI applications that use media stream devices such as COM ports. If your TAPI device is a serial device accessed through a COM port (such as a modem) and the line is open, the service provider needs to have the COM port open. But with the COM port open, non-TAPI applications and console applications are prevented from accessing the COM port. Therefore, a TAPI application should open the line (and keep it open) only if it is waiting for incoming calls or it is actively engaged in placing an outgoing call.
- Non-Telephony communications applications may need to share resources with Telephony applications.

The data structure **LINEDEVCAPS** contains a capabilities bit (LINEDEVCAPFLAGS_CLOSEDROP) that tells whether closing a line while a call is still active causes calls on the line to be dropped. If TRUE, the service provider drops (clears) all active calls on the line when the last application having that line open closes it using the **lineClose** function. If the bit is set to FALSE, the service provider does not drop active calls on the line; instead, it leaves these calls active and under the control of an external device or devices, such as phones.

Therefore, an application can examine **LINEDEVCAPS** to detect in advance whether closing a line will cause this active call to be dropped. The application should warn all appropriate users that the call is about to be dropped, perhaps by displaying an OK/Cancel dialog box that lets the user keep the line open.

For example, if a desktop computer and a phoneset are both connected directly to an analog line (in a party-line configuration), the service provider should set the flag to FALSE, as the offhook phone would automatically keep the call active even after the computer powers down.

As another example, a user is speaking on the phone on a call owned by an active application. The user decides to leave the office for the day and shuts down the system. The operating system in turn shuts down the active telephony application, which attempts to close the lines it has open. Whether the call is automatically hung up depends on whether the LINEDEVCAPFLAGS_CLOSEDROP bit in **LINEDEVCAPS** is set or not and whether the phone is offhook.

If other applications are monitoring the call, the service provider will not even be informed that one application has closed the line. It is only when the last application that has a handle to the call closes the line that the service provider is informed (with **TSPI_lineClose**). At that point, it is up to the service provider to handle any remaining calls. If the service provider is required to drop the calls, it should do so but it should first warn applications about this requirement with the LINEDEVCAPFLAGS_CLOSEDROP flag.

Closing an application should ideally perform the following cleanup tasks:

- **Dispose** of all calls.
- **Close** all open lines and phones.
- **Shut down** the usage of the Telephony API.

Failure to do so may leave calls in indeterminate states.

Learning About Existing Calls at Application Startup

Applications can become aware of existing calls on the line at the time of application startup with **lineGetNewCalls**. This function gives the application handles (with monitor privilege) to all the calls on the line or address for which it did not already have handles.

An example of the usefulness of this mechanism is when a user starts a fax application during a voice call, having decided to send a fax. The new fax application needs to discover the existing call (using **lineGetNewCalls**) to request ownership of the call in order to send the fax.

Media Mode Updating

A call's media mode as stated in **dwMediaMode** in **LINECALLINFO** and a media mode message (LINE_MONITORMEDIA) sent from the service provider to the application can and often will differ. An application could use **lineMonitorMedia** to enable monitoring for particular media modes, and use any resulting LINE_MONITORMEDIA messages as guidance in determining what the calling party might be probing for or what the Unknown application should probe for first. LINE_MONITORMEDIA messages report events on the line (such as probes being received from the remote station) and not "hard" decisions about media modes. When an application determines that a call involves a particular media mode, it calls **lineSetMediaMode** to update **dwMediaMode** in **LINECALLINFO** and inform other applications of this update.

Communication Between Applications

Applications can communicate with each other by writing to and reading from a field in a specified call's information record, the **LINECALLINFO** data structure. With **lineSetAppSpecific**, any owner application can write to the application-specific field called **dwAppSpecific**. It is uninterpreted by the Telephony API or any of its service providers. This field's usage is entirely controlled by applications.

The field can be read from the **LINECALLINFO** record returned by **lineGetCallInfo**. However, **lineSetAppSpecific** must be used to set the field so that changes become visible to other applications. When this field is changed, all other applications with call handles are sent a LINE_CALLINFO message with an indication that the **dwAppSpecific** field has changed.

Special Cases in TAPI

The following topics describe situations that could occur in multiple-application environments.

A Call Without an Owner

What happens to an incoming call if no applications have the line open with owner privilege? First, TAPI informs all monitoring applications on the line about the call. If no monitoring application switches to owner privilege to answer the call, the service provider eventually drops the call.

The following steps describe what normally happens if a line has only monitoring applications, a call comes in, and no applications answer the call. Assume that the service provider is not configured to answer new calls by itself.

- The service provider passes the call handle to TAPI.
- TAPI passes monitor handles to the monitor applications.
- TAPI determines that the call is in the *offering* (not *connected*) state and lets the call "sit."
- The remote party eventually hangs up, and the call reverts to the *idle* call state because it was never connected.
- The monitors are notified that the call is *idle* and deallocate their handles if they have not already done so.
- The last deallocation causes a **TSPI_lineCloseCall**.

Although the call in this example was never answered, its appearance and disappearance may have been significant to the applications monitoring the line. On a network that offers caller ID, a user may want to screen incoming calls, recording who has called without necessarily answering every call. Monitoring applications can help accomplish this without ever needing to answer a call.

The Call is Answered Elsewhere

This scenario is similar to a call without an owner, but the user answers the phone elsewhere on the line, and this answer is detected by the provider through the hardware. The call is reported as being in the *connected* state, though it is not connected to any application on the local computer. The call eventually goes *idle*, at which point each application (monitor as well as all owner applications) must deallocate its handle.

The following steps describe what occurs in a scenario in which a line has only monitoring applications, the service provider is configured to not answer new calls by itself, and a call comes in:

- The service provider passes the call handle to TAPI.
- TAPI passes monitor handles to the monitor applications.
- TAPI determines that the call in the *offering* (not *connected*) state and lets the call "sit."
- The user picks up a ringing downline phone, and the call becomes *connected*.
- All monitoring applications are sent a corresponding LINE_CALLSTATE message.
- The user eventually hangs up the downline phone, and the call reverts to the *idle* state.
- The monitor applications are notified the call is *idle* and deallocate their handles, if they have not already done so.
- The last deallocation causes a **TSPI_lineCloseCall**.

Auto-Answer by the Service Provider

In this scenario, the provider is set up to auto-answer a new call. The provider answers it and determines the call's media mode. It then informs TAPI of the call. Because no applications decide to take ownership, TAPI cannot pass the call to any application.

A line has only monitoring applications. The service provider is configured to auto-answer all new calls. At this point, a call comes in.

These are the steps:

- The service provider passes the call handle to TAPI.
- TAPI passes monitor handles to the monitor applications (the call is in the *connected* state).
- TAPI realizes that the call is *connected*, but there are, so far, no potential owners.
- TAPI must do a **TSPI_lineDrop** to cause the call to revert to *idle*.
- The monitor applications are notified that the call is idle and deallocate their handles if they have not already done so.
- The last deallocation causes a **TSPI_lineCloseCall**.

This example illustrates an important point for writers of monitoring applications. It is not a good idea to wait monitor for incoming calls you later want to own. It is better to open the line as an owner, because there might not be enough time to change your privilege from monitor to owner before TAPI drops the call.

Call States and Events

A connection is not fully established until both parties are communicating. To reach that point, the establishment of the call goes through several stages, as does the clearing (termination) of the call. A call's events cause it to transition through *call states* as it comes into existence, is used to exchange information, and terminates. These call-state transitions result from both solicited and unsolicited events. A solicited event is one caused by the application controlling the call (as when it invokes TAPI operations), while unsolicited events are caused by the switch, the telephone network, the user pressing buttons on the local phone, or the actions of the remote party. Some operations on line devices, addresses, and calls may first require that the line, address, or call upon which they operate be in certain specific states.

Different call states indicate that connections exist to different parts of the switch. For example, a *dial tone* is a particular state of a switch that means the computer is ready to receive digits.

Whenever a call changes state, TAPI reports the new state to the application in a message. This programming model, therefore, is one in which the application reacts to the events reported to it, as opposed to a rigid call-state model. In other words, call-state notification tells the application what the call's new state is, instead of reporting the occurrence of specific events and assuming that the application will be able to deduce the transitions between two states.

Call State Definitions

Some of the call states and events defined by TAPI are exclusive to inbound or outbound call processing, while others occur in both cases. Several of these call states provide additional information that can be used by the application. For example, the *busy* state signifies that a call cannot be completed because a resource between the originator and the destination is unavailable, as when an intermediate switch has reached its capacity and cannot handle an additional call. Information supplied with the *busy* state includes *station busy* or *trunk busy*. Station busy means that the destination's station is busy (the phone is offhook), while trunk busy means that a circuit in the switch or network is busy. The call states defined by TAPI are listed below.

Call State	Description
idle	This corresponds to the "null" state: No activity exists on the call, which means that no call is currently active.
offering (inbound)	When the switch informs the computer of the arrival of a new incoming call, that call is in the offering state. Note that *offering* is not the same as causing a phone or computer to ring. When a call is offered, the computer is not necessarily instructed to alert the user. *Example:* An incoming call on a shared call appearance is offered to all stations that share the appearance, but typically only the station that has the appearance as its primary address is instructed to ring. If that station does not answer after some amount of time, the bridging stations may be instructed to ring as well.
accepted (inbound)	An application has taken responsibility for an incoming call. In ISDN, the *accepted* state is entered when the called party equipment sends a message to the switch indicating that it is willing to present the call to the called person; this has the side effect of alerting the users at both sides of the call: the caller's and the called party's. An incoming call can always be immediately answered without first being separately accepted.
dial tone (outbound)	Indicates that the switch is ready to receive a dialable number. In most telephony environments, this state is entered when audible dial tone is detected by the line device. Additional information includes:
−n*ormal dial tone*	The "normal" or everyday dial tone, usually a continuous tone.
−s*pecial dial tone*	A special dial tone is often used to signal certain conditions such as message-waiting. This is usually an interrupted dial tone.
dialing (outbound)	The originator is dialing digits on the call.

	The dialed digits are collected by the switch.
proceeding (outbound)	The call is proceeding through the network. This occurs after dialing is complete and before the call reaches the dialed party, as indicated by ringback, busy, or answer.
special info (outbound)	The call is receiving a special information signal, which precedes a prerecorded announcement indicating why a call cannot be completed. Such announcements can be of these types:
–*no circuit*	A no-circuit or emergency announcement.
–*customer irregularity*	This typically means that the dialed number is not correct.
–*reorder*	A reorder or equipment-irregularity announcement.
busy (outbound)	The call is receiving a busy signal. Busy indicates that some resource is not available and the call cannot be normally completed at this time. Additional information consists of:
–*station busy*	The station at the other end is off-hook.
–*trunk busy*	The network is congested. This usually produces a "fast busy" signal.
ringback (outbound)	The station to be called has been reached, and the destination's switch is generating a ring tone back to the originator. A ringback means that the destination address is being alerted to the call.
connected (inbound and outbound)	Information is being exchanged over the call.
on hold (inbound and outbound)	The call is currently held by the switch. This frees the physical line, which allows another call to use the line.
conferenced (inbound and outbound)	The call is a member of a conference call and is logically in the connected state (to the conference bridge). A call in the *conferenced* state refers to a conference call (in the *connected*, *onHold*, ... state).
on hold pending conference (inbound and outbound)	The conference call is currently on hold and waiting for the user to add another party.
on hold pending transfer (inbound and outbound)	The call is on hold in preparation of being transferred.
disconnected (inbound and outbound)	The call has been disconnected by the remote party.
unknown (inbound and outbound)	The call exists, but its state is currently unknown. This may be the result of poor call progress detection by the service provider. A call state message with the call state set to unknown may also be generated to inform the TAPI DLL about a

new call at a time that that the actual call state of the call is not exactly known.

Although under normal circumstances an outbound call is likely to transition to *connected* through a number of intermediate states (*dial tone*, *dialing*, *proceeding*, *ringback*), other paths are often possible. For example, the *ringback* state may be skipped, as when a hot phone (or other non-dialed phone) transitions directly to *connected*.

An application should always process call-state event notifications. Call-state transitions valid for one switch or configuration may be invalid for another. For example, consider a line from the switch that (using a simple Y-connector) physically terminates both at the computer and at a separate phone set, creating a party line configuration between the computer and the phone set. The computer termination and, therefore, the application using TAPI, may not know of the activities on the line handled by the phone set. That is, the line may be in use without the service provider being aware of it. An application that wants to make an outbound call will succeed in allocating a call appearance from the API, but this results in sharing the active call on the line. In this case, blindly sending a DTMF dial string without first checking for a dial tone may not result in intended (or polite) behavior.

Obtaining Call State Information

Using the function **lineGetCallStatus**, an application can receive complete call status information for the specified call as a data structure of type **LINECALLSTATUS**. The function **lineGetCallInfo** returns mostly constant information about a call as a data structure of the type **LINECALLINFO**. The message LINE_CALLSTATE is sent unsolicited to an application to notify it about changes in a call's state.

The call-information data structure maintained for each call contains an application-specific field that applications can read from and write to. This field is not interpreted by TAPI. Applications can use it to tag calls in application-specific ways. Writing to this field so that the change is visible to other applications can be done with the function **lineSetAppSpecific**.

Supplementary Line Functions

In Win32 Telephony, *supplementary* functions are functions whose form and functionality has been defined by the API description, but which are not required in basic Telephony. They are functions that developers of telephony applications and service providers may choose to implement to suit the design of their custom products. That is, in contrast to Basic Telephony functions, Supplementary Telephony functions are optional.

This section describes the Supplementary functions used with TAPI-defined line devices. For convenience, they are grouped into general functional categories.

Bearer Mode and Rate

The notion of *bearer mode* corresponds to the quality of service requested from the network for establishing a call. It is important to keep the concept of bearer mode separate from that of media mode. The *media mode* of a call describes the type of information that is exchanged over a specific call of a given bearer mode. As an example, the analog telephone network (PSTN) provides only 3.1 kHz voice-grade quality of service—this is its bearer mode. However, a call with this bearer mode can support a variety of different media modes such as voice, fax, or data modem. In other words, media modes require certain bearer modes. TAPI manages the bearer modes only by passing the bearer mode parameters on to the network. Media modes are fully managed through the appropriate media stream APIs, although some limited support is provided in TAPI.

The bearer mode of a call is specified when the call is set up, or is provided when the call is offered. With line devices able to represent channel pools, it is possible for a service provider to allow calls to be established with wider bandwidth. The *rate* (or bandwidth) of a call is specified separately from the bearer mode, allowing an application to request arbitrary data rates.

The bearer modes defined in TAPI are:

- **Voice**, which is regular 3.1 kHz analog voice service; bit integrity is not assured.
- **Speech**, which is G.711 speech transmission on the call.
- **Multiuse**, as defined by ISDN.
- **Data**, which is unrestricted data transfer; the data rate is specified separately.
- **Alternate speech and data**, which is the alternate transfer of speech and unrestricted data on a call (ISDN).
- **Non-call-associated signaling**, which provides a clear signaling path from the application to the service provider.

Although support for changing a call's bearer mode or bandwidth is limited in networks today, TAPI provides an operation to request a change in the bearer mode or the data-rate parameters of an existing call. This function is **lineSetCallParams**.

Call Monitoring

Call monitoring includes media, digit, and tone monitoring, as described in the following topics.

Media Monitoring

When a call is in the *connected* state, information can be transported over the call. A call's media mode provides an indication of the type of information (for example, its data type, or higher-level protocol) of this media stream. TAPI allows applications to be provided with a notification about changes in a call's media mode. The notification provides an indication of the call's new media mode. The service provider decides how it wants to make this determination. For example, the service provider could use signal processing of the media stream to determine the media mode, or it could rely on distinctive ringing patterns assigned to different media streams, or on information elements passed in an out-of-band signaling protocol. Independent of how the media mode determination is achieved, the application is simply informed about media mode changes on an existing call.

The media modes defined by TAPI include:

- **Unknown**. The media mode of the call is not currently known—the call is unclassified.
- **Interactive voice**. Voice energy was detected on the call, and the call is handled as an interactive voice call with a person at the application's end.
- **Automated voice**. Voice energy was detected on the call, and the call is handled as a voice call but with no person at the application's end, such as with an answering machine application.
- **Data modem**. A modem session on the call. Current modem protocols require the called station to initiate the handshake. For an inbound data modem call, the application can typically make no positive detection. How the service provider makes this determination is its choice. For example, a period of silence just after answering an inbound call can be used as a heuristic to decide that this might be a data modem call.
- **G3 fax**. A group 3 fax session on the call.
- **G4 fax**. A group 4 fax session on the call.
- **TDD**. The call's media stream uses the Telephony Devices for the Deaf protocol.
- **Digital data**. A digital data stream of unspecified format.
- **Teletex, Videotex, Telex, Mixed**. These correspond to the telematic services of the same names.
- **ADSI**. An Analog Display Services Interface session on the call. ADSI enhances voice calls with alphanumeric information downloaded to the phone and the use of soft buttons on the phone.

An application can enable or disable *media monitoring* on a specified call with **lineMonitorMedia**. The application specifies which media modes it is interested in monitoring, and when media monitoring is enabled, the detection of a media mode change causes the application to be notified with the LINE_MONITORMEDIA message. This message provides the call handle on which the media mode change was detected as well as the new media mode.

There is a distinction between the media mode of a call as reported by **lineGetCallInfo** and the media mode event reports by LINE_MONITORMEDIA messages. A call's media mode is determined exclusively by owner applications of the call and is not automatically changed by media monitoring events. The one exception is the initial media mode determination that can be performed by the TAPI dynamic-link library to select the first owner of a call. One could argue that in this case, the library is the owner of the call.

Default media mode monitoring is performed for the media modes for which the line device has been opened. This allows an inbound call's media mode to be determined before the call is handed to an application based on what the application demands. The scope of the media monitoring of a call is bound by the lifetime of the call. Media monitoring on a call ends as soon the call *disconnects* or goes *idle*.

An application can obtain device IDs for various device classes associated with an opened line by calling **lineGetID**. This function takes a line handle, address, or call handle and a device class description. It returns the device ID for the device of the given device class that is associated with the open line device, address, or call. If the device class is "tapi/line," then the device ID of the line device is returned. If the device class is "mci/wave," then the device ID of an mci waveaudio device is returned (if supported),

which allows activities such as the recording or playback of audio over the call on the line.

The application can use the returned device ID with the corresponding media API to query the device's capabilities and subsequently open the media device. For example, if your application needs to use the line as a waveform device, it must first call **waveInGetDevCaps** and/or **waveOutGetDevCaps** to determine the waveform capabilities of the device. The typical waveform data format supported by telephony in North America is 8-bit m-law at 8000 samples per second, although the wave device driver can convert this sample rate and companding to other more common multimedia audio formats.

To subsequently open a line device for audio playback using the waveform API, an application calls **waveOutOpen**. The implementation of **waveOutOpen** is device specific, and there are a variety of options for implementing this function.

Digit Monitoring

Digit monitoring monitors the call for digits. TAPI allows digits to be signaled according to two methods (digit modes):

- **Pulse**. Digits are signaled as pulse or rotary sequences. For detection, these pulses manifest themselves as nothing more than sequences of audible clicks. Valid pulse digits are '0' through '9'.
- **DTMF**. Digits are signaled as DTMF (Dual Tone Multiple Frequency) tones. Valid DTMF digits are '0' through '9', 'A'. 'B', 'C', 'D', '*', and '#'. Both the beginning and the down edge of DTMF digits can be monitored.

An application can enable or disable digit monitoring on a specified call with **lineMonitorDigits**. When digit monitoring is enabled, detected digits cause the application to be notified with the LINE_MONITORDIGITS message. This message provides the call handle on which the digit was detected as well as the digit value and the digit mode. The scope of digit monitoring is bound by the lifetime of the call. Digit monitoring on a call ends as soon as the call *disconnects* or goes *idle*.

Tone Monitoring

Tone monitoring monitors the media stream of a call for specified tones. A tone is described by its component frequencies and cadence. An implementation of the API may allow several different tones to be monitored simultaneously. An application can tag each tone to be able to distinguish the different tones for which it requests detection.

An application can enable or disable tone monitoring on a specified call with **lineMonitorTones**. With this function, the application indicates which tones to detect on a specified call. When tone monitoring is enabled, detected digits cause the application to be notified with the LINE_MONITORTONE message. This message provides the call handle on which the tone was detected as well as the application's tag for the tone.

The scope of tone monitoring is bound by the lifetime of the call. Tone monitoring on a call ends as soon the call *disconnects* or goes *idle*.

Note The monitoring of tones, digits, or media modes often requires the use of resources of which the service provider can only have a finite amount. A request for monitoring can be rejected if resources are not available. For the same reason, an application should disable any unnecessary monitoring.

Media Control

An application can request the execution of a limited set of media-control operations on the call's media stream triggered by telephony events. Although an application is encouraged to use the media API specifically defined for the media mode, media control can yield a significant performance improvement for client/server implementations. The **lineSetMediaControl** function is used to set up a call's media stream for media control by allowing an application to specify a list of tuples specifying a telephony event and the associated media-control action. The telephony events that can trigger media-control activities are:

- **Detection of a digit**. The application provides a list of specific digits and the media-control action that each of them triggers.
- **Detection of a media mode**. The application provides a list of media modes and the media-control actions that a transition into the media mode triggers.
- **Detection of a specified tone**. The application specifies a list of tones and the media-control action that each tone detection triggers.
- **Detection of a call state**. The application specifies a list of call states and the media-control action that each transition to the call state triggers.

The media-control actions listed below are defined generically for the different media modes. Not all media streams can provide meaningful interpretations of the media-control actions. The operations should map well to audio streams:

- **Start** starts the media stream.
- **Reset** resets the media stream.
- **Pause** stops or pauses the media stream.
- **Resume** starts or resumes the media stream.
- **Rate up** increases the rate (speed) of the media stream by an implementation-defined amount.
- **Rate down** decreases the rate of the media stream by an implementation-defined amount.
- **Rate normal** returns the rate to normal.
- **Volume up** increases the volume (amplitude) of the media stream.
- **Volume down** decreases the volume of the media stream.
- **Volume normal** returns the volume to normal.

The scope of media control is bound by the lifetime of the call. Media control on a call ends as soon the call *disconnects* or goes *idle*. Only a single media-control request can be outstanding on a call across all applications.

Digit Gathering

Besides enabling digit monitoring and being notified of digits one at a time, an application can also request that multiple digits be collected in a buffer. Only when the buffer is full or when some other termination condition is met is the application notified. Digit gathering is useful for functions such as credit card number collection. It is performed when an application calls **lineGatherDigits**, specifying a buffer to fill with digits. Digit gathering terminates when one of the following conditions is true:

- The requested number of digits has been collected.
- One of multiple termination digits is detected. The termination digits are specified to **lineGatherDigits**, and the termination digit is placed in the buffer as well.
- One of two timeouts expires. The timeouts are a first digit timeout, specifying the maximum duration before the first digit must be collected, and an inter-digit timeout, specifying the maximum duration between successive digits.
- Digit gathering is canceled explicitly by **lineGatherDigits** again with either another set of parameters to start a new gathering request or by using a NULL digit buffer parameter to cancel.

When terminated for any reason, a LINE_GATHERDIGITS message is sent to the application that requested the digit gathering. Only a single digit gathering request can be outstanding on a call at any given time across all applications that are owners of the call.

Digit gathering and digit monitoring can be enabled on the same call at the same time. In that case, the application will receive a LINE_MONITORDIGITS message for each detected digit and a separate LINE_GATHERDIGITS message when the buffer is sent back.

Generating Inband Digits and Tones

Once a call is in the *connected* state, information can be transmitted over it. Two functions are provided that allow end-to-end inband signaling between the application and remote station equipment such as an answering machine. One function is **lineGenerateDigits**, which generates inband digits on a call, signaling them over the voice channel. Digits can be signaled as either rotary/pulse sequences or as DTMF tones. The other function is **lineGenerateTone**, which enables the application to generate one of a variety of multifrequency tones inband (over the media stream). This generates telephony tones, such as ringback, beep, and busy, as well as arbitrary multi-frequency, multi-cadenced tones.

Only one digit or tone generation can be in progress on a call at any one time. When digit or tone generation completes, a LINE_GENERATE message is sent to the application that requested the generation. In the case where multiple digits are generated, only a single message is sent back after all digits have been generated. Calling **lineGenerateDigits** or **lineGenerateTone** while digit or tone generation is in progress will abort the generation currently in progress and send the LINE_GENERATE message to the application whose generation was aborted with a *cancel* indication.

Call Operations

Call operations include acceptance, rejection, redirecting, holding, forwarding, parking, pickup, and completion. These operations are described in the following topics.

Call Accept, Reject, and Redirect

In environments like ISDN, call offering is separate from alerting. In fact, after a call has been offered to an application, a time window exists during which the application has a number of options:

- It can immediately answer the call using **lineAnswer**.
- It can accept the call using **lineAccept**, which initiates alerting to both the caller (as *ringback*) and the called party (as *ring*).
- It can reject the offering call using **lineDrop**, which reverts the offering call to the *idle* state.
- It can redirect the call using **lineRedirect**, which deflects the offering call to another address. The call reverts to the *idle* state.

Call Hold

Most PBXs can associate multiple calls with a single line. A call can be placed on *hard hold*, which frees the user's line/address to make other calls. An application can place a call on hard hold by calling **lineHold**. An application can retrieve a call on hard hold by calling **lineUnhold**.

Hard hold is different from a *consultation hold*. A call is automatically placed on consultation hold, for example, when a call is prepared for transfer or conference.

Call Transfer

TAPI provides two mechanisms for call transfer: *blind transfer* and *consultation transfer*.

- In blind transfer (or single-step transfer), an existing call is transferred to a specified destination address in one phase using **lineBlindTransfer**.
- In a consultation transfer, the existing call is first prepared for transfer to another address using **lineSetupTransfer**. This places the existing call on consultation hold, and identifies the call as the target for the next transfer-completion request. The **lineSetupTransfer** function also allocates a consultation call that can be used to establish the consultation call with the party to which the call will be transferred. The application can dial the extension of the destination party on the consultation call (using **lineDial**), or it can drop and deallocate the consultation call and instead activate an existing held call (using **lineUnhold**), if supported by the switch.

While the initial call is on consultation hold and the consultation call is active, the application can toggle between these calls using **lineSwapHold**.

Finally, the application completes the transfer in one of two ways using **lineCompleteTransfer**:

- Transfer the call on transfer hold to the destination party. Both calls will revert to the *idle* state.
- Enter a three-way conference. A new call handle is created to represent the conference and this handle is returned to the application.

In version 0x00020000 and greater, applications can use the "one step transfer" feature of many PBXs (a single button press to establish a consultation transfer) using LINECALLPARAMFLAGS_ONESTEPTRANSFER with **lineSetupTransfer**.

Call Conference

Conference calls are calls that include more than two parties simultaneously. They can be set up using either a switch-based conference bridge or an external server-based bridge. Typically, only switch-based conferencing will allow the level of conference control provided by the API. In server-based conference calls all participating parties dial into the server which mixes all the media streams together and sends each participant the mix; there may be no notion of individual parties in the conference call, only that of a single call between the application and the bridge server.

A conference call can be established in a number of ways, depending on device capabilities:

- A conference call can begin as a regular two-party call, such as a call established with **lineMakeCall**. Once the two-party call exists, additional parties can be added, one at a time. Calling **lineSetupConference** prepares a given call for the addition of another party, and this action establishes the conference call. This operation takes the original two-party call as input, allocates a conference call, connects the original call to the conference, and allocates a consultation call whose handle is returned to the application.

  Note The capabilities of the addressed line device can limit the number of parties conferenced in a single call and whether or not a conference starts out with a normal two-party call.

  The application can then use **lineDial** on the consultation call to establish a connection to the next party to be added. The **lineDrop** function can be used to abandon this call attempt. The third party is added with the **lineAddToConference** function, which specifies both the conference call and the consultation call.
- A three-way conference call can be established by resolving a transfer request for three-way conference. In this scenario, a two-party call is established as either an inbound or outbound call. Next the call is placed on transfer hold with the **lineSetupTransfer** function, which returns a consultation call handle. After a period of consultation, the application may have the option to resolve the transfer setup by selecting the three-way conference option which conferences all three parties together in a conference call with **lineCompleteTransfer** with the *conference* option (instead of the *transfer* option). Under this option, a conference call handle representing the conference call is allocated and returned to the application.
- A conference call may need to be established with **lineSetupConference** without an existing two-party call. This returns a handle for the conference call and allocates a consultation call. After a period of consultation, the consultation call can be added with **lineAddToConference**. Additional parties are added with **linePrepareAddToConference** followed by **lineAddToConference**.

To add parties to an existing conference call, the application uses **linePrepareAddToConference**. When calling this function, the application supplies the handle of an existing conference call. The function allocates a consultation call that can later be added to the conference call and returns a consultation call handle to the application. This conference call is then placed on conference hold. Once the consultation call exists, it can be added to the existing conference call with **lineAddToConference**.

Once a call becomes a member of a conference call, the member's call state reverts to *conferenced*. The state of the conference call typically becomes *connected*. The call handle to the conference call and all the added parties remain valid as individual calls. LINE_CALLSTATE events can be received about all calls. For example, if one of the members disconnects by hanging up, an appropriate call-state message can inform the application of this fact; such a call is no longer a member of the conference.

As is the case with call transfer, the application can toggle between the consultation call and the conference call using **lineSwapHold**.

Use the call handle for the member calls to later remove the call from the conference. Do this by calling **lineRemoveFromConference** on the call handle. This operation is not commonly available in its fully general form. Some switches may not allow it at all, or only allow the most recently added party to be

removed. The line's device capabilities describe which type of **lineRemoveFromConference** is possible.

In version 0x00020000 and greater, applications can use the "no hold conference" feature of PBXs by using the LINECALLPARAMFLAGS_NOHOLDCONFERENCE option with **lineSetupConference**; this feature allows another device, such as a supervisor or recording device, to be silently attached to the line.

Removing a Party

When canceling the consultation call to the third party for a conference call or when removing the third party in a previously established conference call, the service provider (and switch) may release the conference bridge and revert the call back to a normal two-party call. If this is the case, *hConfCall* will transition to the *idle* state, and the only remaining participating call will transition to the *connected* state.

Call Park

Two forms of call parking are provided: *directed call park* and *non-directed call park*. In directed call park, the application specifies the destination address where the call is to be parked. This roughly behaves like a call transfer to the destination address, but it doesn't alert or time-out as a transfer would.

In non-directed call park, the switch returns to the application the address where it parked the call. In either case, the function **linePark** is used to park a call. A parked call can later be retrieved with **lineUnpark**. The application specifies the park address to **lineUnpark** which returns a call handle to the unparked call. Appropriate LINE_CALLSTATE messages will be sent to the application as the call is reconnected.

Call Forwarding

Forwarding affects the treatment by the switch or network of incoming calls destined for a given address. The application can specify call forwarding conditions based on the origin of the call (internal, external, selective based on caller ID); the status of the address (busy, no answer, unconditionally); and the destination address where calls are to be forwarded. When the specified conditions are met for an incoming call, the switch deflects the incoming call to the specified destination number. Because the switch performs the forwarding action, the application will typically not know when a call has been forwarded.

The **lineForward** function provides a combination of call forwarding (by setting call-forwarding requests) and a do-not-disturb feature. The **lineForward** function can also cancel any or all of the forwarding requests currently in effect. Some switches require that a call be established to the forwarding address for call forwarding to be initiated. On such systems, **lineForward** allocates a consultation call and returns the handle for it to the application. The consultation call can be used as any other call. After the connection is established, forwarding confirmation is received from the switch, the call is dropped (using **lineDrop**), and forwarding is in effect. A LINE_ADDRESSSTATE message with a *forwarding* indication informs the application about changes in an address' forwarding status.

Note It may be impossible for a service provider to know at all times what forwarding is in effect for an address. Forwarding can be canceled or changed in ways that make it impossible for a service provider to be informed of this fact.

Call Pickup

Call pickup allows an application to answer a call that is alerting at another address. The application invokes **linePickup** by identifying the target of the pickup and is returned a call handle for the picked-up call. There are several ways to specify the target of the pickup request. First, specify the address (extension) of the alerting party. Second, if no extension is specified and the switch allows it, the application can pick up any ringing phone in its pickup group. Third, some switches require a group ID to identify the group to which the ringing extensions belongs.

After the call has been picked up, it is diverted to the application and the application is sent appropriate LINE_CALLSTATE messages for the call. An application can invoke **lineGetCallInfo** for information about the picked-up call, if provided by the switch.

Some key telephone systems support a *transfer through hold* capability on bridged-exclusive call appearances. In this scheme, a call is owned exclusively by a particular phone when it is active, but when the call is on hold it can be picked up on any phone that has an appearance of the line. In versions 0x00020000 and greater, an application can use the **linePickup** function with a NULL target address for this purpose, similar to how the function is used to pick up a call waiting call on an analog line. LINEADDRFEATURE_PICKUPHELD indicates the existence of the capability (in **LINEADDRESSCAPS**) and when it can be invoked (in **LINEADDRESSSTATUS**).

Call Completion

When making an outbound call, the unavailability of certain resources can prevent the call from reaching the *connected* state, as when the destination party is busy or doesn't answer. Unavailable resources include trunk circuits as well as the destination party's station. The **lineCompleteCall** function lets the application specify how it wants to complete a call that cannot be completed normally—this is called "placing a call-completion request." The application has the following options:

- **Camp on** to queue the call until the call can be completed.
- **Call back** requests the called station to return the call when the station returns to idle. Answering the call-back can automatically reinitiate (redial) the connection request.
- **Intrude** allows the application to barge in to the existing call.
- **Message** (also known as "leave word calling") allows the application to send one of a small number of predefined messages to the destination. These messages can be text shown on the phone's display, a voice message left for the user, and so forth.

A call completion request can be canceled with **lineUncompleteCall**. Multiple call completion requests can potentially be outstanding for a given address at any one time. To identify individual requests, the implementation returns a completion ID. When a call completion request completes and results in a new call, the call completion ID is available in the **LINECALLINFO** data structure returned by **lineGetCallInfo**. Canceling a call completion request in progress also uses this call completion ID.

In versions 0x0002000 and greater, the LINECALLREASON_CAMPEDON bit allows a service provider to indicate when a new call was camped on to an address.

Extended Line Functions

Extended Line Services (or device-specific line services) include all service-provider defined extensions to the API. The API defines a mechanism that enables service-provider vendors to extend TAPI using device-specific extensions. The API only defines the extension mechanism, and by doing so provides access to device-specific extensions, but the API does not define their behavior. Behavior is completely defined by the service provider.

TAPI consists of scalar and bit-flag constant definitions, data structures, functions, and messages. Procedures are defined that enable a vendor to extend most of these as follows.

For extensible scalar data constants, a service-provider vendor can define new values in a specified range. As most data constants are DWORDs, typically the range 0x00000000 through 0x7FFFFFFF is reserved for common future extensions, while 0x80000000 through 0xFFFFFFFF are available for vendor-specific extensions. The assumption is that a vendor would define values that are natural extensions of the data types defined by the API.

For extensible bit-flag data constants, a service-provider vendor can define new values for specified bits. As most bit-flag constants are DWORDs, typically a specific number of the lower bits are reserved for common extensions while the remaining upper bits are available for vendor-specific extensions. Common bit flags are assigned from bit zero up; vendor-specific extensions should be assigned from bit 31 down. This provides maximum flexibility in assigning bit positions to common extensions versus vendor-specific extensions. A vendor is expected to define new values that are natural extensions of the bit flags defined by the API.

Extensible data structures have a variably sized field that is reserved for device-specific use. Being variably sized, the service provider decides the amount of information and the interpretation. A vendor that defines a device-specific field is expected to make these natural extensions of the original data structure defined by the API.

Two functions, **lineDevSpecific** and **lineDevSpecificFeature**, and two related messages, LINE_DEVSPECIFIC and LINE_DEVSPECIFICFEATURE, provide a vendor-specific extension mechanism. The **lineDevSpecific** function and associated LINE_DEVSPECIFIC message enable an application to access device-specific line, address, or call features that are unavailable with the Basic or Supplementary Telephony Services. The parameter profile of the **lineDevSpecific** function is generic in that little interpretation of the parameters is made by the API. The interpretation of the parameters is defined by the service provider and must be understood by an application that uses them. The parameters are simply passed through TAPI from the application to the service provider. An application that relies on device-specific extensions will not generally work with other service providers; however, applications written to the Basic and Supplementary Telephony Services will work with the extended service provider.

For convenience, a more specialized escape function is also provided. It is similar to **lineDevSpecific**, but places interpretation on some of the parameters. This more specialized function is **lineDevSpecificFeature**, a device-specific escape function to allow sending switch features to the switch. The message LINE_DEVSPECIFICFEATURE is the device-specific message sent to the application as an indication of features sent to the switch. This function and its associated message allow an application to emulate button presses at the line's feature phone. As feature phones and the meanings of their buttons are vendor-specific, feature invocation using **lineDevSpecificFeature** is also vendor specific.

As mentioned earlier, there is no central registry for manufacturer IDs. Instead, a unique ID generator (EXTIDGEN) is made available.

Passthrough Mode

When a call is active in LINEBEARERMODE_PASSTHROUGH, the service provider gives direct access to the attached hardware for control by the application. This mode is for use by applications desiring temporary direct control over asynchronous modems, accessed through the Win32 Communication functions, for the purpose of configuring or using special features not otherwise supported by the service provider, such as facsimile (Class 1, 2, and so on). This bearer mode is supported by the Universal Modem Driver (UNIMODEM) service provider.

Service providers that support LINEBEARERMODE_PASSTHROUGH indicate it in the **dwBearerModes** member of the **LINEDEVCAPS** structure. When LINEBEARERMODE_PASSTHROUGH is indicated, the Unimodem service provider will also include in the DevSpecific area of the **LINEDEVCAPS** structure the registry key used to access information about the modem associated with the line device, in the following format:

```
struct {
    DWORD dwContents;   // Set to 1 (indicates containing key)
    DWORD dwKeyOffset;  // Offset to key from start of this
                        // structure (not from start of
                        // LINEDEVCAPS structure). 8 in
                        // our case.
    BYTE rgby[...];     // place containing null-terminated
                        // registry key.
}
```

For example:

```
    00000001 00000008 74737953 435c6d65  ........System\C
    65727275 6f43746e 6f72746e 7465536c  urrentControlSet
    7265535c 65636976 6c435c73 5c737361  urrentControlSet
    65646f4d 30305c6d xx003030 xxxxxxxx  Modem\0000.
```

This registry key could then be opened using this function:

```
RegOpenKey(HKEY_LOCAL_MACHINE, pszDevSpecificRegKey, &phkResult)
```

Passthrough mode is invoked most often using the **lineMakeCall** function, by setting the LINEBEARERMODE_PASSTHROUGH bit in the **dwBearerMode** member of the **LINECALLPARAMS** structure pointed to by the *lpCallParams* parameter. When this is done, the service provider will open the serial port to the modem and immediately place the call into LINECALLSTATE_CONNECTED. The application can then use the **lineGetID** function with the device class "comm/datamodem" to obtain an open Win32 file handle to read from and write to the comm port.

Passthrough mode can be invoked in response to an incoming call as well. Generally, applications will invoke passthrough mode while the call is in LINECALLSTATE_OFFERING, before the call has been answered. Instead of calling **lineAnswer**, the applications calls **lineSetCallParams**, passing LINEBEARERMODE_PASSTHROUGH as the *dwBearerMode* parameter. When this is done, as with **lineMakeCall**, the call will immediately be placed into LINECALLSTATE_CONNECTED by the service provider, and the application can obtain a handle to the open port using **lineGetID**. **lineSetCallParams** can be called when the call is in LINECALLSTATE_OFFERING, LINECALLSTATE_ACCEPTED, or LINECALLSTATE_CONNECTED.

Passthrough mode is normally terminated by calling **lineDrop** on the call handle obtained from **lineMakeCall** or the first LINE_CALLSTATE message (if the call was an incoming call). The service provider will close the port, and restore the modem to its default state. The application must call **CloseHandle** on the handle it received from **lineGetID**.

Passthrough mode can also be terminated by calling **lineSetCallParams** with the *dwBearerMode* parameter set to LINEBEARERMODE_VOICE. The media mode set by **lineSetMediaMode** is presumed to be in effect. If LINEMEDIAMODE_DATAMODEM is active, the service provider will take over the call as though it was a data modem call already in progress; if **lineDrop** is subsequently called, the service provider will issue the appropriate modem commands or interface state changes to drop a data call.

Quality of Service

As Asynchronous Transfer Mode (ATM) networking emerges into the mainstream of computing, and support for ATM has been added to other parts of the Microsoft® Windows® operating system, TAPI also supports key attributes of establishing calls on ATM facilities. The most important of these from an application perspective is the ability to request, negotiate, renegotiate, and receive indications of Quality of Service (QOS) parameters on inbound and outbound calls.

QOS information in TAPI is exchanged between applications and service providers in **FLOWSPEC** structures which are defined in Windows Sockets 2.0.

Applications request QOS on outbound calls by setting values in the flowspec fields in the **LINECALLPARAMS** structure. The service provider will endeavor to provide the specified QOS, and fail the call if it cannot; the application can then adjust its parameters and try the call again. Once a call is established, an application can use the **lineSetCallQualityOfService** function to request a change in the QOS; a new bit, LINECALLFEATURE_SETQOS, lets applications determine when this function can be called.

The QOS applicable to inbound or active calls can be obtained by using **lineGetCallInfo** and examining the flowspec fields. A bit in the LINE_CALLINFO message, LINECALLINFOSTATE_QOS, lets applications know when QOS information for a call has been updated.

Support for QOS is not restricted to ATM transports; any service provider can implement QOS features.

Call Center Control

You can use TAPI to manage call centers and other elements of telephony network infrastructure (such as IVR and voice mail servers) through third-party call control. The following topics describe the TAPI features that make this control easier.

Modeling of a Call Center

Service providers can expose each resource on the PBX as a line device and possibly an associated phone device. Terminals which support multiple call appearances would do so through multiple addresses, just as in first-party call control. In fact, the third-party view of a device is identical to the first-party view; applications on the server can see and control all of the first-party devices, whereas an individual client PC connected to the server would only be able to see those devices which are made visible to it through access controls administered by TAPISRV.EXE on the server. Resources other than terminals can also be modeled as line devices. For example, an ACD queue or route point would be modeled as a line device that could have many active calls; an IVR server, voice mail server, or set of predictive dialing ports could also be modeled as a line device that supports multiple calls.

Within this model, the status of the addressed device and calls associated with it can be monitored though existing TAPI messages such as LINE_LINEDEVSTATE, LINE_ADDRESSSTATE, LINE_CALLSTATE, and LINE_CALLINFO, and details obtained through functions such as **lineGetLineDevStatus**, **lineGetAddressStatus**, **lineGetCallStatus**, and **lineGetCallInfo**. Whenever a TAPI object is operated upon through a third-party application running on the server, the result is identical to what would have occurred if the same object had been similarly operated on by a first-party application running on a client PC associated with that device. Status indications sent by the server service provider controlling the switching fabric (or switch) are delivered both to applications running on the server and to those running on associated, authorized clients.

Stations

Station sets being monitored through a third-party link are modeled as a line device and possibly an associated phone device. The line device can have multiple addresses, if the modeled terminal supports more than one directory number (DN). Multiple call appearances on the same DN can be modeled as a single address that supports multiple calls.

Calls between two stations on the switch have two call handles, one giving the call view from the first station (on its line device), and the other giving the call view from the second station (on its line device). For example, a third-party **lineMakeCall** placed by an application on the server would be directed to the line device associated with the station from which the call is to be dialed; a call handle would be created on that line, on the address specified in **LINECALLPARAMS** (thereby giving control over which DN is used on a phone that supports multiple DNs). When the call is offered to the destination address, a new call handle showing a call in *offering* state is created; applications would know that it was another view of the same call by the **dwCallID** member in **LINECALLINFO** being equal for both calls. Both calls would go *idle* when the call was dropped; a call could be dropped from the third-party application by doing a **lineDrop** on either of the call handles.

Predictive Dialing

Predictive dialing is an application that typically runs on a call center telephony server. It uses a list of phone numbers, often obtained from a database, to attempt outbound calls; when a call is *completed*, the call is automatically assigned to an agent for handling. The application can make use of a *predictive dialing port* on a switch, which is a device that can make outbound calls and has special abilities (through DSP, and so on) to detect call progress tones and other audible indications of call state. When a call is made on a predictive dialing port, it will typically be automatically transferred to another device on the switch when the call reaches a particular state or upon detection of a particular media mode; this target device might be a queue for agents handling outbound calls.

Applications identify a device as having predictive dialing capability by the LINEADDRCAPFLAGS_PREDICTIVEDIALER bit in the **dwAddrCapFlags** member in **LINEADDRESSCAPS**. The **dwPredictiveAutoTransferStates** member in **LINEADDRESSCAPS** indicates the states upon which the predictive dialing port can be commanded to automatically transfer a call; if this member is zero, it indicates that automatic transfer is not available, and that it is the responsibility of the application to transfer calls explicitly upon detecting the appropriate call state (or media mode or other criteria). Preferably, switch manufacturers will make available both automatic and manual transfer, and allow applications to select the preferred mechanism, but service providers would have to model the behavior of legacy equipment. A single predictive dialing port (line device/address) can support making several outbound calls simultaneously, as indicated by the **dwMaxNumActiveCalls** member in **LINEADDRESSCAPS**. Predictive dialing capability can also be made available on *any* device, using a shared pool of predictive dialing signal processors, which are bridged onto the line being dialed upon request.

When the **lineMakeCall** function is used on a line capable of predictive dialing (a port with the LINEADDRCAPFLAGS_PREDICTIVEDIALER set) and predictive dialing is requested using LINECALLPARAMFLAGS_PREDICTIVEDIAL, then the call is made in a predictive fashion with enhanced audible call progress detection. Additional fields and constants are defined in the **LINECALLPARAMS** structure passed in to **lineMakeCall** to control the behavior of the predictive dialing port. The **dwPredictiveAutoTransferStates** member indicates the line call states which, upon entry of the call into any of them, the predictive dialing port should automatically transfer the call to the designated target (the bits must be a proper subset of the supported auto-transfer states indicated in **LINEADDRESSCAPS**); the application can leave the field set to 0 if it desires to monitor call states itself and use **lineBlindTransfer** to transfer the call when it reaches the desired condition. The application must specify the desired address to which the call should be automatically transferred in the variable field defined by the **dwTargetAddressSize** and **dwTargetAddressOffset** members in **LINECALLPARAMS**.

Applications can also set a timeout for outbound calls so that the service provider will automatically transition them to a disconnected state if they are not answered. This is controlled using the **dwNoAnswerTimeout** member in **LINECALLPARAMS**.

Call Queues and Route Points

A call queue or route point is a special address within the switch where calls are temporarily held pending action. This characteristic is represented by the bits LINEADDRCAPFLAGS_QUEUE and LINEADDRCAPFLAGS_ROUTEPOINT in the **dwAddrCapFlags** member in **LINEADDRESSCAPS**. All calls appearing on such an address are awaiting action by the application, and there can be default actions that take place (for example, transfer to an agent or trunk) if the application takes no action within a defined period of time. The application must be configured by the system administrator so that it knows what actions it should take regarding calls appearing on each queue or route point address, and the amount of time available to decide on the action to take.

Applications can determine the number of calls pending in a queue or route point using **lineGetAddressStatus**. The **lineGetCallInfo** function can be used to obtain information such as calling ID, called ID, inbound or outbound origin, and so on, and used by the application to make decisions on call handling; calls can be redirected, blind-transferred, dropped, and so on, or just allowed to automatically pass out of the queue to a destination. A call goes to LINECALLSTATE_DISCONNECTED if it is abandoned. Calls go *idle* when they leave the queue; **lineGetCallInfo** can be used to read the redirection ID to determine where they were transferred.

Some switches allow calls in a queue or on hold to receive particular treatment such as silence, ringback, busy signal, music, or listening to a recorded announcement. The **lineSetCallTreatment** function allows the application to control the treatment. The structure delimited by the **dwCallTreatmentListSize** and **dwCallTreatmentListOffset** members in **LINEADDRESSCAPS** allows applications to determine the supported treatments. The **dwCallTreatment** member in **LINECALLINFO** indicates the current treatment, and a LINE_CALLINFO message with LINECALLINFOSTATE_TREATMENT indicates when this changes. The LINECALLFEATURE_SETTREATMENT bit in the **dwCallFeatures** member in **LINECALLSTATUS** indicates when changing the treatment by the application is permitted. The LINECALLTREATMENT_ set of constants defines a limited set of predefined call treatments; service providers can define many more.

ACD Agent Monitoring and Control

Monitoring and control of ACD agent status on stations is supported through these functions: **lineGetAgentCaps**, **lineGetAgentStatus**, **lineGetAgentGroupList**, **lineGetAgentActivityList**, **lineSetAgentGroup**, **lineSetAgentState**, and **lineSetAgentActivity**. The LINE_AGENTSTATUS message is used to indicate when agent information has changed.

These controls are associated with an *address* instead of a line because many ACD systems are implemented with different ACD queues associated with buttons on the phone terminal (and separate call appearances). Also, ACD agent phones can often have separate call appearances for personal calls.

Architecturally, ACD functionality should be implemented in a server-based application. The client functions mentioned above, rather than mapping to the telephony service provider, are conveyed to a server application which has registered (using an option of **lineOpen**) as a handler for such functions. The LINE_PROXYREQUEST message is used to signal to the handler application when a request has been made; it calls the **lineProxyResponse** function to return results and data. Handler applications can also call **lineProxyMessage** to generate LINE_AGENTSTATUS messages when required. In the case of a legacy PBX or stand-alone ACD which implements ACD functionality itself, the telephony service provider for the switch must include a proxy server application that accepts the requests and routes them (possibly using **lineDevSpecific** functions or a private interface) to the service provider, which routes them to the switch.

Call Data

In a call center environment, applications may need to accumulate data about a call (such as IVR input of account numbers) that is desirable to have available to all agents and applications that handle the call. The variable-sized field, bounded by the **dwCallDataSize** and **dwCallDataOffset** members in the **LINECALLINFO** structure, gives the telephony application a way to provide to the service provider data to be passed along with a transferred call and made visible to other applications that are monitoring the call (either on the same PC, or, through the server, on other PCs). The LINECALLINFOSTATE_CALLDATA message indicates whenever this field changes. The **lineSetCallData** function allows an application that owns the call to set this data; LINECALLFEATURE_SETCALLDATA indicates when changing the data is permitted. The **dwMaxCallData** member in **LINEADDRESSCAPS** indicates the maximum number of bytes permitted in this field. Initial call data to be attached to a call can be passed to the service provider in **LINECALLPARAMS**.

Station Status Control

There are three major station status functions that need control: message waiting lights, forwarding, and do not disturb. Forwarding and Do Not Disturb are controllable through the existing **lineForward** function (which is address-specific), and queried using **lineGetAddressStatus**. The LINEDEVSTATUSFLAGS_MSGWAIT bit in the **dwDevStatusFlags** member of **LINEDEVSTATUS** indicates the status of the message waiting light on the device, and a LINEDEVSTATE_MSGWAITON or LINEDEVSTATE_MSGWAITOFF message is sent to indicate when the state changes. The **lineSetLineDevStatus** function allows the message waiting light to be controlled without having to implement a TAPI phone device just for that purpose. The LINEFEATURE_SETDEVSTATUS bit (in the **dwLineFeatures** member of **LINEDEVCAPS** and **LINEDEVSTATUS**) indicates when it can be called, and the **dwSettableDevStatus** member of **LINEDEVCAPS** allows the application to detect which of the device status settings can be controlled from the application. In addition to allowing the message waiting feature to be controlled, it also allows the device's Connected, Inservice, and Locked status to be set, to the extent that these are supported by the switch or other hardware. Calls to this function result in appropriate LINE_LINEDEVSTATE messages being sent to reflect the new status.

Call State Timer

Currently, all timing of calls is left up to applications. This can be quite burdensome if the application is monitoring a large number of calls, and if multiple applications were present, possibly on multiple servers, it would be necessary for them to all maintain timers on the same calls. It therefore makes more sense for call state timing to be handled by the server.

The **tStateEntryTime** member in **LINECALLSTATUS** allows timing of calls in states to be reported. The member (of type **SYSTIME**) indicates the time at which the current state was entered.

Media Event Timers

Many applications depend on the timing relationship between media events (for example, DTMF digits received) in order to determine the nature of a requested operation. For example, in a voice mail application, two consecutive DTMF "1" digits may mean "back up two segments" or "replay from beginning of message", depending on how much time elapsed between the two digits. In a client/server environment, if the DTMF detection is being performed on a separate processor from the one on which the application is running, latency in the local area network makes it very likely that the timing relationship between media events will be skewed, with the result that these timing-based differences could be lost or become unreliable.

To resolve this issue, several TAPI messages can be timestamped. Because it is the *relative* timing between these events that is important, the "clock time" of the event is not important, and sub-second timing is involved, these timestamps use the millisecond-resolution "time since Windows started" returned by the **GetTickCount** function. Applications must be aware that this is the tick count on the server (or machine where the service provider directly managing the hardware is running), and is not necessarily the same machine on which the application is executing; thus, the timestamps in these TAPI messages can only be compared to *each other*, and not to the value returned by **GetTickCount** on the processor on which the application is running.

The TAPI messages which can be timestamped are: LINE_GATHERDIGITS, LINE_GENERATE, LINE_MONITORDIGITS, LINE_MONITORMEDIA, and LINE_MONITORTONE. The tick count will be inserted into *dwParam3* of these messages. If timestamping is not supported by the service provider (which is indicated by the service provider setting *dwParam3* in these messages to 0), then TAPI itself will insert the tick count into *dwParam3* of all of these messages (it can be skewed somewhat, but less than if the application did the same after the messages had traversed an interprocess communication scheme).

Line Devices Overview

A line device is a physical device such as a fax board, a modem, or an ISDN card that is connected to an actual telephone line. Line devices support telephonic capabilities by allowing applications to send or receive information to or from a telephone network. A line device is the logical representation of a physical line device, one of the two device classes supported by TAPI. This section describes line devices and explains how to use the line functions that access these devices.

What is a Line Device?

A line device is a physical device such as a fax board, a modem, or an ISDN card that is connected to an actual telephone line (although the device may not be physically connected to the computer on which the telephony application is running). Line devices support telephonic capabilities by allowing applications to send or receive information to or from a telephone network. A line device contains a set of one or more homogeneous channels that can be used to establish calls.

Within TAPI applications, a line device is the logical representation of a physical line device. Although "line" often connotes something with two endpoints, it is possible to abstract a line device to a single point because TAPI views it only as a point of entry to the line that leads to the switch.

{ewc msdncd, EWGraphic, bsd23548 0 /a "SDK.WMF"}

Although the three lines in the preceding illustration are composed of different hardware and used for different functions, they are abstracted to the same device type and governed by the same rules. The telephone represents not a phone device but a line device used for voice calls. When using this line device for incoming or outgoing calls, the application would also need to open and control an instance of the phone-device class, which is described in detail in later sections.

TAPI requires that every TAPI-capable line device support all of Basic Telephony. If an application needs to use capabilities beyond those of Basic Telephony (namely Supplementary or Extended Telephony), it must first determine the line device's capabilities, which can vary according to network configuration (client versus client/server), hardware, service-provider software, and the telephone network. The **lineNegotiateAPIVersion** function allows the application to identify the set of Extended capabilities supported on a line device, and the **lineNegotiateExtVersion** function allows for different versions of that set to be used. The function **lineGetDevCaps** returns the telephonic capabilities implemented through the use of the Supplementary and Extended (if any) TAPI functions of a given line device in a data structure of the type **LINEDEVCAPS**.

Lines, Channels, and Addresses

In POTS, exactly one channel exists on a line, and this is used exclusively for voice. With ISDN, at least three (and as many as 30 or more) channels can exist on a line simultaneously. Currently, most TAPI functionality involves POTS applications that handle a single line using its single channel because ISDN hardware is not yet widespread. In POTS, an application that wants to transmit data would communicate over one line, and a voice application would communicate over another line—both of these applications could use the same line, but if so, not at the same time.

In general, one line has exactly one address (telephone number). In cases where lines carry two or more channels, each channel can have its own address, which means that a line has as many addresses as it has channels. TAPI assigns Address IDs to these different addresses to make it easier to manipulate them.

Multiple Addresses on a Single Channel

Some installations support the assignment of more than one address to a single channel. On POTS lines, multiple addresses are made possible by various systems, such as DID (direct inward dialing) or distinctive ringing, which are extra-fee services provided by the telephone company.

Many large corporations use DID for incoming calls. Before a call is connected, its destination extension number is signaled to the PBX, which causes the extension to ring instead of the operator's phone. An example of distinctive ringing in a private home would be if the parents used one address, the children another, and a fax machine a third. Because only one line connects the house to the telephone network, all phones ring when a call appears, but the ring pattern will be different depending on the number dialed by the calling party. With distinctive ringing, the people know who the incoming call is meant for, and the fax machine answers its calls by recognizing its own ringing style.

In ISDN, the various B channels might not have separate addresses. Because these B channels might be on the same address, it is the service provider (and not the application or a person who has dialed the number) that assigns calls to these channels.

ISDN Subaddresses

Subaddressing is a capability provided on ISDN lines that allows more information than just a single telephone number to be used when dialing. This additional information can specify an individual telephone extension to ring or, in a computing environment, a particular application to be alerted. Other parameters that can be passed can describe the required aspects of a requested modem connection, such as rate and timing.

Addresses and Address Identifiers

Each line device is assigned one or more addresses. An address corresponds to a telephone directory number, and it is actually assigned twice: First, by the telephone company at the switch, and second, by the user while configuring the local system. If a telephone number is changed at the switch, the user will normally need to assign the new number at the local system, although some systems can be sophisticated enough to perform the reassignment without human control.

After addresses have been assigned to lines, TAPI assigns address IDs to addresses. An address ID is a number between 0 and the number of addresses on the line minus one. Because each address depends on its line to exist, the address's ID is meaningful only in the context of the associated line device. For this reason, an address name consists of not only the address ID, but also an identifier of the line. It serves as a kind of shorthand, an easy way for programmers to identify addresses.

Address Configurations

The relationship of an address to a line (and to other local addresses) is known as its configuration. The network or switch can configure address-to-line assignments in several ways. The main types of address configurations recognized by TAPI are:

1. **Private**. The address is assigned to one line device only. An inbound call for this address is offered (the switch informs the desktop computer of an incoming call) at one line device only.
2. **Bridged**. A bridged address is a single address assigned to more than one line device. (Different switch vendors have different names for address bridging, such as multiple appearance directory number (MADN), bridged appearance, or shared appearance.) An incoming call on a bridged address will be offered on all lines associated with the address. The network of lines connected together is known as the bridge. Different variations of bridged behavior are possible:
   - **Bridged-Exclusive**. Connecting one of the bridged lines to a remote party causes the address to appear "in use" to all other members of the bridge.
   - **Bridged-New**. Connecting one of the bridged lines to a remote party does not preclude the other lines from using the bridged address to answer or make calls. However, a new call appearance is allocated to another of the connected lines.
   - **Bridged-Shared**. If one line is connected to a remote party, other bridged lines that use the address automatically enter into a multi-party conference call on the existing call.
3. **Monitored**. The line indicates the busy or idle status of the address, but the line cannot use the address for answering or making calls.

Address-Related Functions and Messages

Different line devices can have different capabilities and so can their addresses. Additionally, switching features and capabilities can be different for different addresses. An application calls the function **lineGetAddressCaps** to determine the telephony capabilities of each address and then receives this information in a data structure of the type **LINEADDRESSCAPS**. In a similar way, an application can call **lineGetDevCaps** for a line device to determine the number of addresses assigned to the line, as well as other information.

TAPI's device-query capability and status and event reporting mechanisms give an application the information it needs to manage the different kinds of bridged-address arrangements. For example, the application can determine whether a bridged station has answered a call by tracking the status changes and call-state event changes on the address. (For more information about call states, see Call States and Events.)

Normally, addresses on a line device are identified by their address IDs. However, TAPI lets applications that make outgoing calls use alternate address types for the originating address, such as dialable format (see the following section), or naming mechanisms specific to a given service provider. This can be accomplished through API extensions that are based on switch-assigned station IDs. A useful function for this purpose is **lineGetAddressID**, which retrieves the ID of an address specified in an alternate format.

Address Formats

Each address ID corresponds to a directory number through which incoming calls can be placed to the address. Similarly, making outbound calls typically requires that a directory number—the address—be supplied to identify the party being called.

Storing Numbers in Electronic Address Books

Many users choose to dial people, fax machines, bulletin boards, and other entities by selecting their names from an address book. The actual number that is dialed depends on the geographic location of the user and the way the line device to be used is connected. For example, a desktop computer can have access to two lines, one connected to a PBX, the other to the telephone company's central office. When making a call to the same party, different numbers can have to be used. (To dial through the PBX, for example, the computer may need to dial '9' to "get out," or a different prefix may be needed for a call made through the central office.) Or, a user may make calls from a portable computer and want to use a single, static address book even when calling from different locations or telephony environments. TAPI's address translation capabilities let the user inform the computer of the current location and the desired line device for the call. TAPI then handles any dialing differences, requiring no changes to the user's address book. An application uses the **lineTranslateAddress** function to convert an address from the canonical address format to the dialable address format (see the next section).

A related topic is the handling of international call-progress monitoring, which is the process of listening for audible tones such as dial tone, special information tones, busy signals, and ringback tones to determine the state of a call (its *progress* through the network). Because the cadences and frequencies of call-progress tones vary from country to country, the service provider must know what call progress to follow when making an international outbound call. Therefore, applications specify the destination country code when placing outgoing calls.

Canonical Addresses

The canonical address format is intended to be a universally constant directory number. For this reason, numbers in address books are best stored using canonical format. A canonical address is an ASCII string with the following structure:

+ *CountryCode* Space [**(***AreaCode***)** Space] *SubscriberNumber* | *Subaddress* ^ *Name* CRLF ...

The components of this structure are given in the following table.

Component	Meaning
+	Equivalent to ASCII Hex (2B). Indicates that the number that follows it uses the canonical format.
CountryCode	A variably sized string containing one or more of the digits 0-9. The *CountryCode* is delimited by the following Space. It identifies the country in which the address is located.
Space	Exactly one ASCII space character (0x20). It is used to delimit the end of the *CountryCode* part of an address.
AreaCode	A variably sized string containing zero or more of the digits 0-9. *AreaCode* is the area code part of the address and is optional. If the area code is present, it must be preceded by exactly one ASCII left parenthesis character (0x28), and be followed by exactly one ASCII right parenthesis character (0x29) and one ASCII Space character (0x20).
SubscriberNumber	A variably sized string containing one or more of the digits 0-9. It may include other formatting characters as well, including any of the dialing control characters described in the Dialable Address Format: AaBbCcDdPpTtWw*#!,@$? The subscriber number should not contain the left parenthesis or right parenthesis character (which are used only to delimit the area code), nor should it contain the "|", "^", or CRLF characters (which are used to begin following fields). Most commonly, non-digit characters in the subscriber number would include only spaces, periods ("."), and dashes ("-"). Any allowable non-digit characters which appear in the subscriber number will be omitted from the DialableString returned by the **lineTranslateAddress** function, but will be retained in the DisplayableString.
|	ASCII Hex (7C). If this optional character is present, the information following it up to the next + | ^ CRLF, or the end of the canonical address string is treated as subaddress information, as for an ISDN subaddress.
Subaddress	A variably sized string containing a subaddress.

	The string is delimited by + | ^ CRLF or the end of the address string. During dialing, subaddress information is passed to the remote party. It can be such things as an ISDN subaddress or an e-mail address.
^	ASCII Hex (5E). If this optional character is present, the information following it up to the next CRLF or the end of the canonical address string is treated as an ISDN name.
Name	A variably sized string treated as name information. Name is delimited by CRLF or the end of the canonical address string and can contain other delimiters. During dialing, name information is passed to the remote party.
CRLF	ASCII Hex (0D) followed by ASCII Hex (0A), and is optional. If present, it indicates that another canonical number is following this one. It is used to separate multiple canonical addresses as part of a single address string (inverse multiplexing). For example, the canonical representation of the main switchboard telephone number at Microsoft Corporation would be: +1 (206) 882-8080

Dialable Addresses

The dialable address format describes a number that can be dialed on the given line. A dialable address contains part addressing information and is part navigational in nature. Any input string which does not begin with a "+" character is presumed to be not in canonical format and therefore in dialable address format, and is returned to the application unmodified. A dialable address is an ASCII string with the following structure:

DialableNumber | *Subaddress* ^ *Name* CRLF ...

The components of this structure are given in the following table.

Component	Meaning
DialableNumber	digits and modifiers 0-9 A-D * # , ! W w P p T t @ $? ; delimited by | ^ CRLF or the end of the dialable address string. The plus sign (+) is a valid character in dialable strings. It indicates that the phone number is a fully-qualified international number. Within the *DialableNumber*, note the following definitions: 0-9 A-D * # ASCII characters corresponding to the DTMF and/or pulse digits.
!	ASCII Hex (21). Indicates that a hookflash (one-half second onhook, followed by one-half second offhook before continuing) is to be inserted in the dial string.
P p	ASCII Hex (50) or Hex (70). Indicates that pulse dialing is to be used for the digits following it.
T t	ASCII Hex (54) or Hex (74). Indicates that tone (DTMF) dialing is to be used for the digits following it.
,	ASCII Hex (27). Indicates that dialing is to be paused. The duration of a pause is device specific and can be retrieved from the line's device capabilities. Multiple commas can be used to provide longer pauses.
W w	ASCII Hex (57) or Hex (77). An uppercase or lowercase W indicates that dialing should proceed only after a dial tone has been detected.
@	ASCII Hex (40). Indicates that dialing is to "wait for quiet answer" before dialing the remainder of the dialable address. This means to wait for at least one ringback tone followed by several seconds of silence.
$	ASCII Hex (24). Indicates that dialing the billing information is to wait for a "billing signal" (such as a credit card prompt tone).
?	ASCII Hex (3F). Indicates that the user is to be prompted before continuing with dialing.

	The provider does not actually do the prompting, but the presence of the "?" forces the provider to reject the string as invalid, alerting the application to the need to break it into pieces and prompt the user in-between.
;	ASCII Hex (3B). If placed at the end of a partially specified dialable address string, it indicates that the dialable number information is incomplete and more address information will be provided later. ";" is only allowed in the *DialableNumber* portion of an address.
|	ASCII Hex (7C), and is optional. If present, the information following it up to the next + | ^ CRLF, or the end of the dialable address string is treated as subaddress information (as for an ISDN subaddress).
Subaddress	A variably sized string containing a subaddress. The string is delimited by the next + | ^ CRLF or the end of the address string. When dialing, subaddress information is passed to the remote party. It can be for an ISDN subaddress, an e-mail address, and so on.
^	ASCII Hex (5E), and is optional. If present, the information following it up to the next CRLF or the end of the dialable address string is treated as an ISDN name.
Name	A variably sized string treated as name information. Name is delimited by CRLF or the end of the dialable address string. When dialing, name information is passed to the remote party.
CRLF	ASCII Hex (0D) followed by ASCII Hex (0A). If present, this optional character indicates that another dialable number is following this one. It is used to separate multiple dialable addresses as part of a single address string (for inverse multiplexing).

The **lineTranslateAddress** function and related support functions are used to translate an address from canonical format to dialable format. An application might not use this function to dial a number but it might use it to generate and display for verification a number that could be dialed. Also, it can compute the local time at the destination address from the country code and area code.

The application uses **lineTranslateAddress** to specify both the line device upon which it intends to dial the call and a canonical address, and the function returns the dialable number and the country code. Because the line device can have specific dialing requirements, it is part of the context needed for an accurate translation.

The user's location also plays a role in address translation. Information related to the current location, such as the country code, area code, and outside line access codes is entered by the user through the Telephony applet in the Control Panel. The *Subaddress* and *Name* fields, if present in the address, are unmodified by the translation. Alphabetic characters in the number, such as in 1-800-FOR-TAPI, are not translated by the **lineTranslateAddress** function due to the different standardizations in use in different countries, but they may be translated by applications themselves.

Although an application can use dialable addresses returned by **lineTranslateAddress**, it is not limited to them and can compose its own dialable numbers.

Initialization and Shutdown in TAPI

For an application to use any of TAPI's basic or supplementary line functions, it needs a connection to TAPI through which it can receive messages. The application establishes this connection, using either the **lineInitializeEx** or the **phoneInitializeEx** function. The parameters of these functions allow the application to specify the message notification mechanism the application desires to use. Following are specifics about the initialization process:

- The initialization functions are not device-related. When an application calls an initialization function, TAPI does not act on a line or phone device or an abstraction thereof.
- The first time an initialization function is called in a telephony session, TAPI also sets up the telephony environment. Among the tasks it performs are loading the TAPI dynamic-link library and TAPISRV.EXE, and loading the device drivers (Telephony service providers and ancillary components) specified in the registry. In addition, the communication link described above is established between TAPI and the calling application.
- The INIFILECORRUPT error can be returned if TAPI determines that the registry contains an invalid entry. When this error occurs (in **lineInitializeEx** and **phoneInitializeEx**, or another function), the user should identify and resolve the problem. It may be necessary to rebuild the registry or a portion of it, which can be done through the Telephony Control Panel.

  For example, the LINEERR_NODRIVER ("the driver was not installed") error indicates either that a service provider that was previously installed can no longer be found or that some subsidiary component of a service provider (such as a VxD) cannot be found. When this error is encountered, the application should advise the user to correct the problem with the Driver Setup function within the Telephony Control Panel.
- Although each application needs only one associated with TAPI, it can call an initialization function more than once to specify other message notification path.
- Both **lineInitializeEx** and **lineShutdown** (and the corresponding phone functions) operate synchronously. That is, these functions return a success or failure indication, not an asynchronous Request ID.

Upon completion, the **lineInitializeEx** function returns two pieces of information to the application: an application handle and the number of available line devices.

- The application handle represents the application's usage of TAPI. That is, to TAPI, it represents the application. TAPI functions that use line or call handles (explained later in this section) do not require the application handle, because this handle is derived from the specified line, phone, or call handle.
- The **lineInitializeEx** function also returns the number of line devices available to the application through TAPI. Line devices are identified by their device identifier (device ID). Valid device IDs range from zero to one less than the number of line devices. For example, if **lineInitializeEx** reports that there are two line devices in a system, the valid line-device IDs are 0 and 1.

Once an application is finished calling TAPI's line functions, it calls **lineShutdown** and passes its application handle to terminate its usage of TAPI. This allows TAPI to free any resources assigned to the application.

Calls

Unlike line devices and addresses, calls are dynamic. A call represents a connection between two (or more) addresses. The originating address (the caller) is the address from which the call originates, and the destination address (the called) identifies the remote end point or station with which the originator wishes to communicate.

Zero, one, or more calls can exist on a single address at any given time. A familiar example of multiple calls on a single address is call waiting: During a conversation with one party, a subscriber with call waiting is alerted that another party is trying to call. The subscriber can flash the phone to answer the second caller (which automatically places the first party on hold) and then toggle between the two parties by flashing. In this example, the subscriber has two calls on one address on the line. Because the person at the telephone handset can be talking to only one remote party at a time, only one call is active per line at any point in time. The telephone switch keeps the other calls on hold. With a line able to encompass more than one channel, different configurations can allow multiple active calls on a line at the same time.

Call Handles

TAPI identifies a specific call by means of the call's handle, and TAPI assigns call handles as required. One call handle exists for every call owned or monitored by an application, and an application can obtain call handles in a number of well defined ways. Certain TAPI functions create new calls. As they do so, they return any new call's handle to the application. Sometimes, call handles are provided unsolicited in message sent to the application from TAPI, as is the case with inbound calls or calls being handed off by other applications.

For every call, one handle exists per application—unique call handles are provided to each application by TAPI. This means that different applications with handles to the same call use different handles for it, which limits the scope of a call handle to a single application. In addition, the service provider can assign a unique call ID to a call (unrelated to the call's handle), which is used to track the call across transfers. Whether or not a service provider can assign call IDs to calls is a device capability.

The privileges of an application for a given call are maintained by TAPI and are not the property of an application's handle for the call. (For information about call privileges, see TAPI Applications.) Resources such as memory are allocated dynamically for each call for each application that is given a handle to the call. These resources are not automatically deallocated when the call is dropped as the application may still find it useful to extract information from the call (such as for logging purposes). Therefore, applications must dispose of their call handle when they have finished using it by calling the **lineDeallocateCall** function.

Version Negotiation

Over time, different versions may exist for TAPI, applications, and service providers for a line or phone. New versions may define new features, new fields to data structures, and so on. Version numbers therefore indicate how to interpret various data structures.

To allow optimal interoperability of different versions of applications, versions of TAPI itself, and versions of service providers by different vendors, TAPI provides a simple, two-step version negotiation mechanism for applications. Two different versions must be agreed on by the application, TAPI, and the service provider for each line device. The first is the version number for Basic and Supplementary Telephony and is referred to as the API version. The other is for provider-specific extensions, if any, and is referred to as the extension version. The format of the data structures and data types used by TAPI's basic and supplementary features is defined by the API version, while the extension version determines the format of the data structures defined by the vendor-specific extensions.

Version negotiation proceeds in two phases. In the first phase, the API version number is negotiated and the extension ID associated with any vendor-specific extensions supported on the device is obtained. In the second phase, the extension version is negotiated. If the application does not use any API extensions, it skips the second phase and extensions are not activated by the service provider. If the application does want to use extensions, but the service provider's extensions (the extension ID) are not recognized by the application, the application should skip the negotiation for extension version as well. Each vendor has its own set of legal (recognized) versions for each set of extension specifications it distributes.

The **lineNegotiateAPIVersion** function is used to negotiate the API version number to use. It also retrieves the extension ID supported by the line device, returning zeros if no extensions are supported. With this function call, the application provides the API version range it is compatible with. TAPI in turn negotiates with the line's service provider to determine which API version range it supports. TAPI next selects a version number (typically, although not necessarily, the highest version number) in the overlapping version range that the application, the DLL, and the service provider have supplied. This number is returned to the application, along with the extension ID that defines the extensions available from that line's service provider.

If the application wants to use the extensions defined by the returned extension ID, it must first call **lineNegotiateExtVersion** to negotiate the extension version. In a similar negotiation phase, the application specifies the already agreed-upon API version and the extension version range it supports. TAPI passes this information to the service provider for the line. The service provider checks the API version and the extension version range against its own, and selects the appropriate extension version number, if one exists.

When the application later calls **lineGetDevCaps**, it returns a set of device capabilities for the line that correspond to the results of version negotiation. These include the line's device capabilities consistent with the API version and the line's device-specific capabilities consistent with the extension version. The application must specify both of these version numbers when it opens a line. At that point, the application, the DLL, and the service provider are committed to using the agreed-upon versions. If device-specific extensions are not to be used, the extension version should be specified as zero.

In an environment where multiple applications open the same line device, the first application to open the line device selects the versions for all future applications that want to use the line (service providers do not support multiple versions simultaneously.) Similarly, an application that opens multiple line devices may find it easier to operate all line devices under the same API version number.

Phone Devices Overview

A phone device is one of the two device classes supported by TAPI. This section describes phone devices and explains how to use the TAPI phone functions to access these devices.

The Phone Device

A phone device is a device that supports the phone device class and that includes some or all of the following elements:

- **Hookswitch/transducer**. This is a means for audio input and output. The Telephony API recognizes that a phone device can have several transducers, which can be activated and deactivated (taken offhook or placed onhook) under application or manual user control. TAPI identifies three types of hookswitch devices common to many phone sets:

  Handset The traditional mouth-and-ear piece combination that must be manually lifted from a cradle and held against the user's ear.

  Speakerphone Enables the user to conduct calls hands-free. The hookswitch state of a speakerphone can usually be changed both manually and by the API. The speakerphone can be internal or external to the phone device. The speaker part of a speakerphone allows multiple listeners.

  Headset Enables the user to conduct calls hands-free. The hookswitch state of a headset can usually be changed both manually and by the API.

  A hookswitch must be offhook to allow audio data to be sent to and/or received by the corresponding transducer.
- **Volume Control/Gain Control/Mute**. Each hookswitch device is the pairing of a speaker and a microphone component. The API provides for volume control and muting of speaker components and for gain control or muting of microphone components.
- **Ringer**. A means for alerting users, usually through a bell. A phone device can be able to ring in a variety of modes or patterns.
- **Display**. A mechanism for visually presenting messages to the user. A phone display is characterized by its number of rows and columns.
- **Phone buttons**. An array of buttons. Whenever the user presses a button on the phone set, the API reports that the corresponding button was pressed. Button-lamp IDs identify a button and lamp pair. Of course, it is possible to have button-lamp pairs with either no button or no lamp. Button-lamp IDs are integer values that range from 0 to the maximum number of button-lamps available on the phone device, minus one. Each button belongs to a button class. Classes include call appearance buttons, feature buttons, keypad buttons, and local buttons.
- **Lamps**. An array of lamps (such as LEDs) individually controllable from the API. Lamps can be lit in different modes by varying the on and off frequency. The button-lamp ID identifies the lamp.
- **Data areas**. Memory areas in the phone device where instruction code or data can be downloaded to and/or uploaded from. The downloaded information would affect the behavior (or in other words, program) the phone device.

TAPI allows an application to monitor and control elements of the phone device. The most useful elements for an application are the hookswitch devices. The phone set can act as an audio I/O device (to the computer) with volume control, gain control and mute, a ringer (for alerting the user), data areas (for programming the phone), and perhaps a display, though the computer's display is more capable. The application writer is discouraged from directly controlling or using phone lamps or phone buttons, because lamp and button capabilities can vary widely among phone sets, and applications can quickly become tailored to specific phone sets.

There is no guaranteed core set of services supported by all phone devices as there is for line devices (the Basic Telephony Services). Therefore, before an application can use a phone device, the application must first determine the exact capabilities of the phone device. Telephony capability varies with the configuration (client versus client/server), the telephone hardware, and the service-provider software. Applications should make no assumptions as to what telephony capabilities are available. An application determines the device capabilities of a phone device by calling the **phoneGetDevCaps** function. A phone's device capabilities indicate which of these elements exist for each phone device present in the system and what their capabilities are. Although strongly oriented toward real-life telephone sets, this

abstraction can provide a meaningful implementation (or subset thereof) for other devices as well. Take as an example a separate headset directly connected and controllable from the computer and operated as a phone device. Hookswitch changes can be triggered by detection of voice energy (offhook) or a period of silence (onhook); ringing can be emulated by the generation of an audible signal into the headset; a display can be emulated through text-to-speech conversion.

A phone device need not be realized in hardware, but can instead be emulated in software using a mouse- or keyboard-driven graphical command interface and the computer's speaker or sound system. Such a "soft phone" can be an application that uses TAPI. It can also be a service provider, which can be listed as a phone device available to other applications through the API, and as such is assigned a phone device ID.

Depending on the environment and configuration, phone sets can be shared devices between the application and the switch. Some minor provision is made in the API where the switch can temporarily suspend the API's control of a phone device.

Initialization and Shutdown

For an application to use any of TAPI's 30 supplementary phone functions, it needs a connection to TAPI, through which it can receive messages. The application establishes this connection using the **phoneInitializeEx** function. In this function, the application specifies the notification mechanism by which TAPI informs the application of changes in the state of the phone and of asynchronous completion of phone functions.

The **phoneInitializeEx** function returns two pieces of information to the application: an *application handle*, and the number of phone devices. The application handle represents the application's usage of TAPI. The TAPI functions that use phone handles do not require the application handle, as this handle is derived from the specified phone handle.

The second piece of information returned by **phoneInitializeEx** is the number of phone devices available to the application through the Telephony API. Phone devices are identified by their device identifier (*device ID*). Valid device IDs range from zero to the number of phone devices minus one. For example, if **phoneInitializeEx** reports that there are two phone devices in a system, then valid phone device IDs are 0 and 1. Once an application is finished using the phone functions of TAPI, it invokes **phoneShutdown**, passing its application handle to shut down its usage of TAPI. This allows TAPI to free any resources assigned to the application.

Both **phoneInitializeEx** and **phoneShutdown** operate synchronously. That is, these functions either return a success or failure indication, and never return an asynchronous Request ID.

Opening and Closing Phone Devices

After determining the capabilities of a phone device, an application must open the device before it can access functions on that phone. After a phone device has been successfully opened, the application is returned a handle representing the open phone. A phone device can be opened in different modes, thus providing a structured way of sharing a phone device.

The function **phoneOpen** opens the specified phone device to give the application access to functions on the phone. A phone device is identified to **phoneOpen** by means of its device ID, which is passed as the *dwDeviceID* parameter.

Operating Modes and Privileges

The application can request one of two operating modes when opening a phone device. These modes reflect the privileges the application requests for the device:

- Opening a phone for monitor privileges lets the application determine the status of the phone device. Messages are sent to the application when status changes on the phone device are detected.
- An application that opens a phone device for owner privileges can use operations that modify the state of the phone device. Owner privilege automatically includes full monitor privileges as well. At any time, a given phone device can be open only once for owner privileges, but multiple times for monitor privileges. All **phoneSet** operations require owner privileges, while all **phoneGet** operations require only monitor privileges.

Device IDs

Other TAPI phone functions use a handle to an open phone device to identify the open phone device. The only functions for phone devices that take a phone device ID parameter (as opposed to a phone handle) are the **phoneGetDevCaps** and **phoneOpen** functions. An application can retrieve the phone's device ID by using the function **phoneGetID** with the phone handle as a parameter.

An application can also obtain device IDs for various device classes associated with an opened phone by invoking **phoneGetID**. See Device Classes for device class names.

This function takes a phone handle and a device class description. It returns the device ID for the device of the given device class that is associated with the open phone device. If the device class is "tapi/phone," the device ID of the phone device is returned.

In contrast with line devices, for which the basic line services provide the equivalent of POTS, no minimum guaranteed set of functions is defined for phone devices. While each phone device provides at least the functions and messages described in this section, they do not offer any actual operations on the physical phone device.

Closing the Phone Device

The **phoneClose** function closes the specified phone device. The phone device can also be forcibly closed after the user has modified the configuration of that phone or its driver. If the user wants the configuration changes to be effective immediately (as opposed to after the next system restart), and they affect the application's current view of the device (such as a change in device capabilities), then a service provider can forcibly close the phone device.

These messages can also be sent unsolicited as a result of the phone device being reclaimed in some other manner. An application should therefore be prepared to handle unsolicited PHONE_CLOSE messages. At the time the phone device is closed, any outstanding asynchronous replies pertaining to that device are suppressed.

Hookswitch Devices

A phone device can have multiple *hookswitch* devices. A hookswitch is the switch that connects or disconnects a device from the phone line. On a telephone, for example, this is the switch that is automatically activated when a user lifts the receiver from the cradle to get a new dial tone. The Telephony API defines three types of hookswitch devices for a phone: handset, speakerphone, and headset. Each hookswitch device has a speaker and a microphone component, and operates in one of four hookswitch modes:

- **Onhook**. The hookswitch device is onhook, and both its microphone and speaker are disabled.
- **Microphone only**. The hookswitch device is offhook, its microphone is enabled, and its speaker is mute.
- **Speaker only**. The hookswitch device is offhook, its microphone is mute, and its speaker is enabled.
- **Microphone and speaker**. The hookswitch device is offhook, and both microphone and speaker are enabled.

The **phoneSetHookSwitch** function is used to set the hookswitch mode of one or more of the hookswitch devices of an open phone device. For example, to mute or unmute the microphone or speaker component of a hookswitch device, use **phoneSetHookSwitch** with the appropriate hookswitch mode. The function **phoneGetHookSwitch** can be used to query the hookswitch mode of a hookswitch device of an open phone device.

When the mode of a phone's hookswitch device is changed manually, for example by lifting the handset from its cradle, a PHONE_STATE message is sent to the application to notify the application about the state change. Parameters to this message provide an indication of the change.

The volume of the speaker component of a hookswitch device can be set with **phoneSetVolume**. Volume setting is different from mute in that muting a speaker and later unmuting it will preserve the volume setting of the speaker. The **phoneGetVolume** function can be used to return the current volume setting of a hookswitch device's speaker of an open phone device.

The microphone component of a hookswitch device can also be gain controlled. Gain setting is different from mute in that muting a microphone and later unmuting it will preserve the gain setting of the microphone. Use **phoneSetGain** to set the gain of a hookswitch device's microphone of an open phone device, and **phoneGetGain** to return the gain setting of a hookswitch device's microphone of an opened phone.

When the volume or gain of a phone's hookswitch device is changed, a PHONE_STATE message is sent to the application function to notify the application about the state change. Parameters to this message provide an indication of the change.

Display

The Telephony API provides access to a phone's display. The display is modeled as an alphanumeric area with rows and columns. A phone's device capabilities indicate the size of a phone's display as the number of rows and the number of columns. Both these numbers are zero if the phone device does not have a display. Use **phoneSetDisplay** to write information to the display of an open phone device, and **phoneGetDisplay** to retrieve the current contents of a phone's display.

When the display of a phone device is changed, a PHONE_STATE message is sent to the application function to notify the application about the state change. Parameters to this message provide an indication of the change.

Ring

A single phone may be able to ring with different ring modes. Given the wide variety of ring modes available, ring modes are identified by means of their ring mode number. A ring mode number ranges from zero to the number of available ring modes minus one.

The functions an application would use to control a phone device's ring modes are **phoneSetRing**, which rings an open phone device according to a given ring mode, and **phoneGetRing**, which returns the current ring mode of an opened phone device.

When the ring mode of a phone device is changed, a PHONE_STATE message is sent to the application to notify the application about the state change. Parameters to this message provide an indication of the change.

Phone Buttons

The Telephony API models a phone's buttons and lamps as button-lamp pairs. A button with no lamp next to it or a lamp with no button is specified using a "dummy" indicator for the missing lamp or button. A button with multiple lamps is modeled by using multiple button-lamp pairs.

Information associated with a phone button can be set and retrieved. When a button is pressed, a PHONE_BUTTON message is sent to the application function. The parameters of this message are a handle to the phone device and the button-lamp ID of the button that was pressed. The keypad buttons '0' through '9', '*', and '#' are assigned the fixed button-lamp IDs 0 through 11.

The functions associated with buttons are **phoneSetButtonInfo**, which sets the information associated with a button on a phone device, and **phoneGetButtonInfo**, which returns information associated with a button on a phone device. The PHONE_BUTTON message is sent to an application when a button on the phone is pressed.

The information associated with a button does not carry any semantic meaning as far as TAPI is concerned. It is useful for device-specific applications that understand the meaning of this information for a given phone device, or for display to the user, such as online help.

Lamps

The lamps on a phone device can be lit in a variety of different lighting modes. Unlike ringing patterns, lamp modes are more uniform across phone sets of different vendors. A common set of lamp modes is defined by the API. A lamp identified by its lamp-button ID can be lit in a given lamp mode. A lamp can also be queried for its current lamp mode.

The TAPI functions used for lamps are **phoneSetLamp**, which lights a lamp on a specified open phone device in a given lamp lighting mode, and **phoneGetLamp**, which returns the current lamp mode of the specified lamp.

When a lamp of a phone device is changed, a PHONE_STATE message is sent to the application to notify the application about the state change. Parameters to this message provide an indication of the change.

Data Areas

Some phone sets support the notion of downloading data from or uploading data to the phone device, which allows the phone set to be programmed in a variety of ways. The Telephony API models these phone sets as having one or more download or upload areas. Each area is identified by a number that ranges from zero to the number of data areas available on the phone minus one. Sizes of each area can vary. The format of the data itself is device-specific.

The TAPI **phoneSetData** function downloads a buffer of data to a given data area in the phone device, and the **phoneGetData** function uploads the contents of a given data area in the phone device to a buffer.

When a data area of a phone device is changed, a PHONE_STATE message is sent to the application to notify the application about the state change. Parameters to this message provide an indication of the change.

Status

Most of the get and set operations deal with one component of information only. The **phoneGetStatus** function returns complete status information about a phone device to an application.

As mentioned earlier, whenever a status item changes on the phone device, a PHONE_STATE message is sent to the application function. This message's parameters include a handle to the phone device and an indication of the status item that changed.

An application can use **phoneSetStatusMessages** to select the specific status changes for which it wants to be notified. Correspondingly, **phoneGetStatusMessages** returns the status changes for which the application wants to be notified.

Extended Telephony Phone Functions

The Extended Phone Services (or Device-Specific Phone Services) include all extensions to the Telephony API defined by the service provider. TAPI defines a mechanism that enables service-provider vendors to extend TAPI using device-specific extensions. TAPI defines only the extension mechanism, and by doing so provides access to device-specific extensions. The Telephony API does not define their behavior, which is completely defined by the service provider.

TAPI consists of scalar and bit-flag data constant definitions, data structures, functions, and messages. Procedures are defined that enable a vendor to extend most of these, as described in the following topics.

Scalar Data Constants

For extensible scalar data constants, a service-provider vendor can define new values in a specified range. Because most data constants are DWORDs, the range 0x00000000 through 0x7FFFFFFF is typically reserved for common future extensions, while 0x80000000 through 0xFFFFFFFF is available for vendor-specific extensions. The assumption is that a vendor would define values that are natural extensions of the data types defined by the API.

Bit-Flag Data Constants

For extensible bit-flag data constants, a service-provider vendor can define new values for specified bits. Because most bit-flag constants are DWORDs, a specific number of the lower bits are usually reserved for common extensions, while the remaining upper bits are available for vendor-specific extensions. Common bit flags are assigned from bit zero up, and vendor-specific extensions should be assigned from bit 31 down. This scheme provides maximum flexibility in assigning bit positions to common extensions, as opposed to vendor-specific extensions. A vendor is expected to define new values that are natural extensions of the bit flags defined by the API.

Extensible data structures have a variably sized field that is reserved for device-specific use. Because the field is variably sized, the service provider decides the field's amount of information and interpretation. A vendor that defines a device-specific field is expected to make these natural extensions of the original data structure defined by the API.

Functions and Messages

The **phoneDevSpecific** function and its associated PHONE_DEVSPECIFIC message enable an application to access device-specific phone features that are unavailable through the common Telephony services for phones. In other words, **phoneDevSpecific** is the device-specific escape function that allows vendor-dependent extensions, and PHONE_DEVSPECIFIC is the device-specific message that is sent to the application.

The parameter profile of the **phoneDevSpecific** function is generic in that little interpretation of the parameters is made by the Telephony API. Rather, the interpretation of the parameters is defined by the service provider and must be understood by an application that uses them. The parameters are simply passed through by TAPI from the application to the service provider. An application that relies on device-specific extensions will usually not work with other service providers, but applications written to the common telephony phone services will work with the extended service provider.

Assisted Telephony Overview

Assisted telephony provides very basic telephony functionality to primarily non-telephonic applications. If your application needs extensive telephonic control or is meant to handle incoming calls, you can skip this section.

Call Requests

Assisted Telephony provides telephony-enabled applications with an easy-to-use mechanism for making phone calls without requiring the developer to become fully telephony literate.

The **tapiRequestMakeCall** function requests a voice call between the user and a remote party specified by its phone number. The request is made to TAPI, which passes it to an application that is registered as a recipient of such requests. This recipient is a call-manager application.

After the application has made the request, the call is controlled entirely from the call-manager application because Assisted Telephony applications cannot manage calls. Because the more complex aspects of telephony and all user-interface operations are handled by the call-manager application, telephony-enabled applications need not be modified in any substantial way. In fact, applications that allow this operation to be invoked from their built-in script language may not need to be modified at all.

The **tapiGetLocationInfo** function returns to the application the country code and city (area) code which the user has set in the current location parameters in the Telephony control panel. The application can use this information to assist the user in forming proper canonical telephone numbers, such as by offering these as defaults when new numbers are entered in a phone book entry or database record.

Request Recipients

Two kinds of applications are needed to run Assisted Telephony. Assisted Telephony *clients* are applications that use Assisted Telephony by calling the functions that have the prefix "tapi." An example of such a client application would be a spreadsheet to which a **Dial** menu command or toolbar button is added.

Assisted Telephony *servers* are applications that can execute Telephony API functions that result from another application's call to a "tapi"-prefixed function. To make itself known as an Assisted Telephony server, such an application registers as one using the function **lineRegisterRequestRecipient**.

The functions of Assisted Telephony (which begin with the prefix "tapi") are known as request functions. Assisted Telephony applications that process these requests—Assisted Telephony servers—are called *request recipients*.

Assisted Telephony Requests

Applications that use Assisted Telephony services only initiate requests that are temporarily queued by TAPI. It is the request recipient application that retrieves these requests and executes them on behalf of the Assisted Telephony application. The **tapiRequestMakeCall** function requests the establishment of a voice call. The requesting application does not control the call.

TAPI allows the user to establish different or the same request recipient applications for each of these services. An application becomes a request recipient by registering with **lineRegisterRequestRecipient**, in which TRUE is specified as the value for the parameter *bEnable*. (Specifying FALSE deregisters the application as a request recipient, which it should do when it has determined that its recipient duties are through for the current session.) The application selects which services it wants to handle in the *dwRequestMode* parameter of **lineRegisterRequestRecipient**. A possible value for a request is LINEREQUESTMODE_MAKECALL, to show that the application will handle **tapiRequestMakeCall** requests. If multiple applications register for the same services, a priority scheme is used to allow the user to select which application is preferred for handling requests. This priority scheme is identical to that used for call hand-off and the routing of incoming calls based on a list of filenames in the HandoffPriorities section of the registry.

Processing Assisted Telephony Requests

The process with which requests are delivered and serviced is as follows:

1. When TAPI receives an Assisted Telephony request, it checks for a request recipient, that is, an application currently registered to process that type of request. If there is a request recipient, the request is queued, and the highest-priority application that has registered for that request's service is sent a LINE_REQUEST message. The message notifies the request recipient that a new request has arrived, and it carries an indication of the request's mode.
2. If TAPI cannot find a currently running application to process such a request, it tries to launch an application that has been registered as capable of doing so. This registration information is recorded in the HandoffPriorities section of the registry. TAPI tries to launch applications in the order in which they are listed in the HandoffPriorities section. (See the following step.)

   If no application is currently registered, TAPI examines the list of request-processing applications on the associated entry in the HandoffPriorities section. If the associated line is missing from the file, if there are no applications listed on it, or if none of the applications in the list can be launched, the request is rejected with the error TAPIERR_NOREQUESTRECIPIENT.

   When a request recipient is launched (whether or not it has been launched by TAPI) it is its duty to call **lineRegisterRequestRecipient** during the startup process and register itself as a request recipient.
3. If one or more applications are listed in the entry, TAPI begins with the first listed application (highest priority), and attempts to launch it using the **CreateProcess** function. If the attempt to launch the application fails, TAPI attempts to launch the next application in the list. When any application launches successfully, TAPI simply queues the request and returns a success indication to the application even though the request hasn't yet been signaled to the request recipient.

   Once the request recipient application is launched, it calls **lineRegisterRequestRecipient**, which causes a LINE_REQUEST message to be sent, signaling that the request is queued. If for some reason the launched application never registers, the request remains queued and remains in the queue indefinitely until an application registers for that type of request.
4. If TAPI finds such a registered application already running or successfully launches one, it queues the request, sending a LINE_REQUEST message to the server application, and returns a success indication for the function call to the Assisted Telephony application. This success message states only that the request has been accepted and queued, not that it has been successfully executed.

When the server application is ready to process a request, it calls the function **lineGetRequest**. This lets it receive any information it needs, such as an address to dial. It then processes the request, using the Telephony API functions (such as **lineMakeCall** and **lineDrop**) that would otherwise be used to place the call. Invoking **lineGetRequest** removes the request from TAPI, and the request parameters are copied in an application-allocated request buffer. The size and interpretation of the contents of the buffer depend on the request mode.

The server must ensure that it uses the correct parameters when executing requests. When doing so, these steps are followed:

1. The request recipient first receives a LINE_REQUEST message informing it that requests can exist for it in the request queue. This tells the application to call **lineGetRequest** and keep calling it until the queue is drained (if the request is for making a new call), or to drop an existing call. This message does not contain the parameters for the request, except in the case of a request to drop an existing call.
2. If the request is to make a new call, the Assisted Telephony server uses the **lineGetRequest** function to retrieve the full request, which includes the request's parameters. The server now has all the information it needs, such as the number to dial or the identification of the maker of the request. First, however, the server must allocate the memory needed to store this information.
3. Finally, the server executes the request by invoking the appropriate Telephony API function or set of

functions.

If TAPI cannot launch an application capable of serving as a request recipient, the Assisted Telephony call fails and returns the error TAPIERR_NOREQUESTRECIPIENT.

Notes on Request Recipient Operations

The following information concerns systems on which Assisted Telephony requests are processed:

- The default registry should list a call manager application in the priority list for **tapiRequestMakeCall**. It would be helpful, but not essential, for the call manager application to have a menu option that allows users to set it to the highest priority.
- When an Assisted Telephony recipient application is launched automatically by TAPI and if it is the only TAPI application in the system, this action initializes TAPI. If the Assisted Telephony recipient application initializes and shuts down the line device before registering for Assisted Telephony requests, TAPI is shut down as well, and the Assisted Telephony request is lost. Assisted Telephony requests might also be lost when another TAPI application that is launched performs an initialize and shutdown.

Using Assisted Telephony

You might use assisted telephony in a word-processing application. Consider a word-processing application that has a button with a caption of "George." When the user selects a telephone number in a document and clicks this button, the application sends a request (**tapiRequestMakeCall**) to the call-control application, which dials the number and notifies the user of the call's status.

Device Classes

A *device class* is a group of related physical devices or device drivers through which applications send and receive the information or data that makes up a call. Every device class has a *device class name* that uniquely identifies the class, and provides information about the programming interface and commands that can be used to open and communicate with the devices in the class.

The Telephony application programming interface (TAPI) associates devices from one or more device classes to each line or phone device. You access one of these devices by retrieving the device identifier for the device using the **lineGetID** or **phoneGetID** function. You supply the device class name, and the function returns the specific port name, device name, device handle, or device identifier that you need to open and access the device. The format of the information returned depends on the device class and is described in subsequent topics of this section.

Note The device identifier definitions apply to 16-bit and 32-bit TAPI. In some cases, the data type of a media handle in the device identifier definition may be different from that specified by the Microsoft® Windows® operating system version 3.*x* or Microsoft® Win32® application programming interface. For example, Windows version 3.*x* and Win32 define wave device identifiers with the **UINT** type, but TAPI defines this device identifier with the **DWORD** type. In such cases, you should cast the media handle to the appropriate data type when using it with the Windows version 3.*x* or Win32 API.

You also use device class names with the **lineConfigDialog** and **phoneConfigDialog** functions to enable the user to set configuration options for the given device, with the **lineGetIcon** and **phoneGetIcon** functions to retrieve an icon to represent the given device, and with the **lineGetDevConfig** and **lineSetDevConfig** functions to directly retrieve and set configuration options for the given device.

By default, there are the following device class names.

Device Class Name	Description
comm	Communications port.
comm/datamodem	Modem through a communications port.
comm/datamodem/ portname	Name of the device to which a modem is connected.
wave/in	Wave audio device (input only).
wave/out	Wave audio device (output only).
midi/in	Midi sequencer (input only).
midi/out	Midi sequencer (output only).
tapi/line	Line device.
tapi/phone	Phone device.
ndis	Network device.
tapi/terminal	Terminal device.

Note These names are not case sensitive; you can use any combination of uppercase and lowercase letters.

Additional device classes and device class names may be available on a given system. In general, if a device does not belong to one of the default device classes, the manufacturer typically defines a new device class and assigns a unique device class name. Check the documentation for the device to determine what additional device classes are available for it. Note, however, that although the device class and media mode are related, they are not the same. A media mode describes a format of

information on a call, and a device class defines the programming interface used to manage that information. So, even if a manufacturer defines a new media mode, it is not necessarily true that the manufacturer also needs to define a new device class to support the mode.

The format of the configuration data used with the **lineSetDevConfig** and **lineGetDevConfig** functions also depends on the device class. In general, you use **lineGetDevConfig** to save a copy of the current device configuration data and then later use **lineSetDevConfig** with the saved configuration data to restore the device configuration to the previous state. This is a convenient way to temporarily change the configuration without requiring the user to manually restore it to the previous state. Because the exact format of the device configuration data may be different with each service provider, you should not use **lineSetDevConfig** and **lineGetDevConfig** to manipulate the device configuration data directly. Some formats are provided only for information.

comm

The comm device class consists of communications ports. You access these devices by using the Win32 file and communications functions.

The **lineGetID** and **phoneGetID** functions fill a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_ASCII value and appending a null-terminated string that specifies the name of the communication port (such as COM1). You use this port name in a call to the **CreateFile** function to open the communication device for the line or phone.

comm/datamodem

The comm/datamodem device class consists of modem devices. You access these devices by using the Win32 file and communications functions. Devices in this class are associated with line devices that support the LINEMEDIAMODE_DATAMODEM media mode, which is specified in the **dwMediaModes** member of the **LINEDEVCAPS** structure for the line device.

The **lineGetID** and **phoneGetID** functions fill a **VARSTRING** structure, setting **dwStringFormat** to the STRINGFORMAT_BINARY value and appending these additional members:

```
HANDLE hComm;             // handle of open comm. device
CHAR   szDeviceName[1];   // name of comm. device
```

The **hComm** member is the handle of the open communications port. This member is NULL if the port is not yet open or if the *dwSelect* parameter of **lineGetID** is not the LINECALLSELECT_CALL value. If a call is active, the service provider typically opens the port itself to get direct control of the communications hardware, but is only required to return a valid handle if the line is *connected*. The service provider opens the port using the FILE_FLAG_OVERLAPPED value and then configures the port using the settings specified by the **lineSetDevConfig** function. You can set additional configuration options for the device by using Win32 functions with the returned handle.

The **szDeviceName** member is a null-terminated ASCII string that specifies the name of the communications port associated with the line, address, or call.

If **hComm** is a valid handle, you can use it in subsequent calls to Win32 file functions, such as **ReadFile** and **WriteFile**, to send and receive data on the call. When you are finished using the communications port and preferably before you use the **lineDeallocateCall** function to deallocate the call, you must close the port by using the **CloseHandle** function.

When using the **lineGetDevConfig** and **lineSetDevConfig** functions, some service providers require that the configuration data for this device class have the following format:

```
typedef struct  tagDEVCFG  {
  DEVCFGHDR   dfgHdr;
  COMMCONFIG  commconfig;
} DEVCFG, *PDEVCFG, FAR* LPDEVCFG;

// Device setting information
typedef struct  tagDEVCFGDR  {
  DWORD       dwSize;
  DWORD       dwVersion;
  WORD        fwOptions;
  WORD        wWaitBong;
} DEVCFGHDR;
```

The following is device configuration information for use with the **lineGetDevConfig** and **lineSetDevConfig** functions.

dwSize
Sum of the size of the **DEVCFGHDR** structure and the actual size of **COMMCONFIG** structure.

dwVersion
Version number of the Unimodem **DevConfig** structure. This member can be MDMCFG_VERSION (0x00010003).

dwOptions
Option flags that appear on the Unimodem Option page. This member can be a combination of these

values:

TERMINAL_PRE (1)
Displays the pre-terminal screen.

TERMINAL_POST (2)
Displays the post-terminal screen.

MANUAL_DIAL (4)
Dials the phone manually, if capable of doing so.

LAUNCH_LIGHTS (8)
Displays the modem tray icon.

Only the LAUNCH_LIGHTS value is set by default

WWaitBong
Number of seconds (in two seconds granularity) to replace the wait for credit tone ($).

Commconfig
COMMCONFIG structure that can be used with the Win32 communications and MCX functions.

comm/datamodem/portname

The comm/datamodem/portname device class consists of the device names to which modems are attached. When this device name is specified in a call to the **lineGetID** function, the function fills the **VARSTRING** structure with a null-terminated ANSI (not UNICODE) string specifying the name of the port to which the specified modem is attached, such as "COM1\0". This is intended primarily for identification purposes in the user interface, but could be used under some circumstances to open the device directly, bypassing the service provider (if the service provider does not already have the device open itself). If there is no port associated with the device, a null string ("\0") is returned in the **VARSTRING** structure (with a string length of 1).

wave/in

The wave/in device class consists of audio devices for low-level wave audio input. You access these devices by using the wave functions, which are described in the Microsoft Win32 Software Development Kit (SDK). Devices in this class are associated with line devices that support the LINEMEDIAMODE_AUTOMATEDVOICE media modem, which is specified in the **dwMediaModes** member of the **LINEDEVCAPS** structure for the line device.

The **lineGetID** and **phoneGetID** functions fill a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending this additional member:

```
DWORD DeviceId;  // identifier of audio device
```

The **DeviceId** member is the identifier of a closed audio device. You use this identifier in a call to the **waveInOpen** function to open the device for input. You can use the resulting device handle to record digitized audio data from the line or phone device.

Although a "wave" device class also exists for low-level wave audio devices, you should always use the wave/in device class for low-level wave input.

wave/out

The wave/out device class consists of audio devices for low-level wave audio output. You access these devices by using the wave functions, which are described in the Win32 SDK. Devices in this class are associated with line devices that support the LINEMEDIAMODE_AUTOMATEDVOICE media mode, which is specified in the **dwMediaModes** member of the **LINEDEVCAPS** structure for the line device.

The **lineGetID** and **phoneGetID** functions fill a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending this additional member:

```
DWORD DeviceId;  // identifier of audio device
```

The **DeviceId** member is the identifier of a closed audio device. You use this identifier in a call to the **waveOutOpen** function to open the device for output. You can use the resulting device handle to play digitized audio data at the line or phone device.

Although a "wave" device class also exists for low-level wave audio devices, you should always use the wave/out device class for low-level wave output.

midi/in

The midi/in device class consists of MIDI sequencers that are used for input. You access these devices by using the MIDI functions, which are described in the Win32 SDK.

The **lineGetID** and **phoneGetID** functions fill a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending this additional member:

```
DWORD DeviceId;  // identifier of MIDI device
```

The **DeviceId** member is the identifier of a closed MIDI device. You use this identifier in a call to the **midiInOpen** function to open the device for input. You can use the resulting device handle to record MIDI data from the line or phone device.

midi/out

The midi/out device class consists of MIDI sequencers that are used for output. You access these devices by using the MIDI functions, which are described in the Win32 SDK.

The **lineGetID** and **phoneGetID** functions fill a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending this additional member:

```
DWORD DeviceId;  // identifier of MIDI device
```

The **DeviceId** member is the identifier of a closed MIDI device. You use this identifier in a call to the **midiOutOpen** function to open the device for output. You can use the resulting device handle to play MIDI data at the line or phone device.

tapi/line

The tapi/line device class consists of all line devices. You access these devices using the TAPI line functions.

The **lineGetID** function fills a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending this additional member.

```
DWORD dwDeviceI;  // line device identifier
```

The **dwDeviceId** member is the identifier of the line device associated with the line handle given by **lineGetID**.

The **phoneGetID** function also fills a **VARSTRING** structure, setting **dwStringFormat** to STRINGFORMAT_BINARY and appending this additional member:

```
DWORD adwDeviceIds[];  // array of line device identifiers
```

The **adwDeviceIds** member is an array containing the device identifiers of all line devices that are associated with the phone device. If there are no associated line devices, **phoneGetID** returns the PHONEERR_INVALDEVICECLASS value.

tapi/phone

The tapi/phone device class consists of all phone devices. You access these devices by using the TAPI phone functions.

The **phoneGetID** function fills a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending this additional member:

```
DWORD dwDeviceI;  // phone device identifier
```

The **dwDeviceId** member is the identifier of the phone device associated with the phone handle given by **phoneGetID**.

The **lineGetID** function also fills a **VARSTRING** structure, setting **dwStringFormat** to STRINGFORMAT_BINARY and appending this additional member:

```
DWORD adwDeviceIds[];  // array of phone device identifiers
```

The **adwDeviceIds** member is an array containing the device identifiers of all phone devices that are associated with the given line device. If there are no associated phone devices, **lineGetID** returns the LINEERR_INVALDEVICECLASS value.

ndis

The ndis device class consists of devices that can be associated with network driver interface specification (NDIS) media access control (MAC) drivers to support network communications. You access these devices by using functions.

The **lineGetID** and **phoneGetID** functions fill a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending these additional members:

```
HANDLE  hDevice;          // NDIS connection identifier
CHAR    szDeviceType[1];  // name of device
```

The **hDevice** member is the identifier to pass to a MAC, such as the asynchronous MAC for dial-up networking, to associate a network connection with the call/modem connection. The **szDeviceType** member is a null-terminated ASCII string specifying the name of the device associated with the identifier. For more information, see documentation about writing NDIS MAC drivers for use with dial-up networking.

tapi/terminal

The tapi/terminal device class consists of the phone devices associated with each terminal on a line or the terminal on each line associated with a phone device. You access these devices by using the TAPI line or phone functions.

The **lineGetID** function fills a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending this additional member:

```
DWORD adwDeviceId[];  // array of phone device identifiers
```

The **adwDeviceId** member is an array of phone device identifiers. There is one array element for each terminal specified by the **dwNumTerminals** member in the **LINEDEVCAPS** structure for the given line device. Each element specifies the identifier of the phone device associated with the corresponding terminal on the line. If there is no phone device associated with a terminal, the element is set to -1 (0xFFFFFFFF).

The **phoneGetID** function fills a **VARSTRING** structure, setting the **dwStringFormat** member to the STRINGFORMAT_BINARY value and appending this additional member:

```
DWORD adwTerminalID[];  // array of terminal identifiers
```

The **adwTerminalID** member is an array of terminal identifiers. There is one array element for each line device identifier specified by the **lineInitialize** or **lineInitializeEx** function. Each array element contains the terminal identifier associated with the phone device for the given line device. If there is no phone device, the element is set to -1 (0xFFFFFFFF). The terminal identifiers range in value from zero to one less than the number specified by the **dwNumTerminals** member in the **LINEDEVCAPS** structure.

Quick Function Reference

The following is the quick function reference for basic telephony services, supplementary telephony services, assisted telephony, and extended telephony services.

Basic Telephony Services Functions

The Basic Telephony functions are listed by category in the following tables. A function is identified as *asynchronous* if it will indicate completion in a REPLY message to the application. If the function always returns its result to the application immediately, the function is considered *synchronous*.

TAPI Initialization and Shutdown

lineInitializeEx	Initializes the Telephony API line abstraction for use by the invoking application. Synchronous.
lineShutdown	Shuts down the application's use of the line Telephony API. Synchronous.

Line Version Negotiation

lineNegotiateAPIVersion	Allows an application to negotiate an API version to use. Synchronous.

Line Status and Capabilities

lineGetDevCaps	Returns the capabilities of a given line device. Synchronous.
lineGetDevConfig	Returns configuration of a media stream device. Synchronous.
lineGetLineDevStatus	Returns current status of the specified open line device. Synchronous.
lineSetDevConfig	Sets the configuration of the specified media stream device. Synchronous.
lineSetStatusMessages	Specifies the status changes for which the application wants to be notified. Synchronous.
lineGetStatusMessages	Returns the application's current line and address status message settings. Synchronous.
lineGetID	Retrieves a device ID associated with the specified open line, address, or call. Synchronous.
lineGetIcon	Allows an application to retrieve an icon for display to the user. Synchronous.
lineConfigDialog	Causes the provider of the specified line device to display a dialog box that allows the user to configure parameters related to the line device. Synchronous.
lineConfigDialogEdit	Displays a dialog box allowing the user to change configuration information for a line device. Synchronous. Version 0x00010004.

Addresses

lineGetAddressCaps	Returns the telephony capabilities of an address. Synchronous.
lineGetAddressStatus	Returns current status of a specified address. Synchronous.
lineGetAddressID	Retrieves the address ID of an address specified using an alternate format. Synchronous.

Opening and Closing Line Devices

lineOpen	Opens a specified line device for providing subsequent monitoring and/or control of the line. Synchronous.
lineClose	Closes a specified opened line device. Synchronous.

Address Formats

lineTranslateAddress	Translates between an address in canonical format and an address in dialable format. Synchronous.
lineSetCurrentLocation	Sets the location used as the context for address translation. Synchronous.
lineSetTollList	Manipulates the toll list. Synchronous.
lineGetTranslateCaps	Returns address translation capabilities. Synchronous.

Call States and Events

lineGetCallInfo	Returns mostly constant information about a call. Synchronous.
lineGetCallStatus	Returns complete call status information for the specified call. Synchronous.
lineSetAppSpecific	Sets the application-specific field of a call's information structure. Synchronous.

Request Recipient Services

These functions are used only in support of assisted telephony.

LineRegisterRequestRecipient	Registers or deregisters the application as a request recipient for the specified request mode. Synchronous.
lineGetRequest	Gets the next request from the Telephony DLL. Synchronous.

Making Calls

lineMakeCall	Makes an outbound call and returns a call handle for it. Asynchronous.
lineDial	Dials (parts of one or more) dialable

	addresses. Asynchronous.
Answering Inbound Calls	
lineAnswer	Answers an inbound call. Asynchronous.
Toll Saver Support	
lineSetNumRings	Indicates the number of rings after which inbound calls are to be answered. Synchronous.
lineGetNumRings	Returns the minimum number of rings requested with lineSetNumRings. Synchronous.
Call Privilege Control	
lineSetCallPrivilege	Sets the application's privilege to the privilege specified. Synchronous.
Call Drop	
lineDrop	Disconnects a call, or abandons a call attempt in progress. Asynchronous.
lineDeallocateCall	Deallocates the specified call handle. Synchronous.
Call Handle Manipulation	
lineHandoff	Hands off call ownership and/or changes an application's privileges to a call. Synchronous.
lineGetNewCalls	Returns call handles to calls on a specified line or address for which the application does not yet have handles. Synchronous.
lineGetConfRelatedCalls	Returns a list of call handles that are part of the same conference call as the call specified as a parameter. Synchronous.
Location and Country Information	
lineTranslateDialog	Displays a dialog box allowing the user to change location and calling card information. Synchronous. Version 0x00010004.
lineGetCountry	Retrieves dialing rules and other information about a given country. Synchronous. Version 0x00010004.

Supplementary Telephony Services Functions

The supplementary telephony functions are listed by category in the following tables. Line services are listed first, phone services next. A function is identified as *asynchronous* if it will indicate completion in a REPLY message to the application. If the function always returns its result to the application immediately, the function is considered *synchronous*.

Line Services

Bearer Mode and Rate

lineSetCallParams	Requests a change in the call parameters of an existing call. Synchronous.

Media Monitoring

lineMonitorMedia	Enables or disables media mode notification on a specified call. Synchronous.

Digit Monitoring and Gathering

lineMonitorDigits	Enables or disables digit detection notification on a specified call. Synchronous.
lineGatherDigits	Performs the buffered gathering of digits on a call. Synchronous.

Tone Monitoring

lineMonitorTones	Specifies which tones to detect on a specified call. Synchronous.

Media Control

lineSetMediaControl	Sets up a call's media stream for media control. Synchronous.
lineSetMediaMode	Sets the media mode(s) of the specified call in its LINECALLINFO structure. Synchronous.

Generating Inband Digits and Tones

lineGenerateDigits	Generates inband digits on a call. Synchronous.
lineGenerateTone	Generates a given set of tones inband on a call. Synchronous.

Call Accept and Redirect

lineAccept	Accepts an offered call and starts alerting both caller (ringback) and called party (ring). Asynchronous.
lineRedirect	Redirects an offering call to another address. Asynchronous.

Call Reject

lineDrop	See Call Drop table under Basic Telephony Services. Asynchronous.

Call Hold

	lineHold	Places the specified call on hard hold. Asynchronous.
	lineUnhold	Retrieves a held call. Asynchronous.

Making Calls

	lineSecureCall	Secures an existing call from interference by other events such as call-waiting beeps on data connections. Asynchronous.

Call Transfer

	lineSetupTransfer	Prepares a specified call for transfer to another address. Asynchronous.
	lineCompleteTransfer	Transfers a call that was set up for transfer to another call, or enters a three-way conference. Asynchronous.
	lineBlindTransfer	Transfers a call to another party. Asynchronous.
	lineSwapHold	Swaps the active call with the call currently on consultation hold. Asynchronous.

Call Conference

	lineSetupConference	Prepares a given call for the addition of another party. Asynchronous.
	LinePrepareAddToConference	Prepares to add a party to an existing conference call by allocating a consultation call that can later be added to the conference call that is placed on conference hold. Asynchronous.
	LineAddToConference	Adds a consultation call to an existing conference call. Asynchronous.
	LineRemoveFromConference	Removes a party from a conference call. Asynchronous.

Call Park

	linePark	Parks a given call at another address. Asynchronous.
	lineUnpark	Retrieves a parked call. Asynchronous.

Call Forwarding

	lineForward	Sets or cancels call forwarding requests. Asynchronous.

Call Pickup

linePickup	Picks up a call that is alerting at another number. Picks up a call alerting at another destination address and returns a call handle for the picked-up call (**linePickup** can also be used for call waiting). Asynchronous.

Sending Information to Remote Party

lineReleaseUserUserInfo	Releases user-to-user information, permitting the system to overwrite this storage with new information. Asynchronous. Version 0x00010004.
lineSendUserUserInfo	Sends user-to-user information to the remote party on the specified call. Asynchronous.

Call Completion

lineCompleteCall	Places a call completion request. Asynchronous.
lineUncompleteCall	Cancels a call completion request. Asynchronous.

Setting a Terminal for Phone Conversations

lineSetTerminal	Specifies the terminal device to which the specified line, address events, or call media stream events are routed. Asynchronous.

Application Priority

lineGetAppPriority	Retrieves handoff and/or Assisted Telephony priority information for an application. Synchronous. Version 0x00010004.
lineSetAppPriority	Sets the handoff and/or Assisted Telephony priority for an application. Synchronous. Version 0x00010004.

Service Provider Management

lineAddProvider	Installs a Telephony service provider. Synchronous. Version 0x00010004.
lineConfigProvider	Displays configuration dialog box of a service provider. Synchronous. Version 0x00010004.
lineRemoveProvider	Removes an existing Telephony service provider. Synchronous. Version 0x00010004.
lineGetProviderList	Retrieves a list of installed service providers. Synchronous. Version 0x00010004.

Agents

lineAgentSpecific	Allows the application to access proprietary handler-specific functions of the agent handler associated with the address. Asynchronous. Version 0x00020000.
LineGetAgentActivityList	Obtains the list of activities from which an application selects the functions an agent is performing. Asynchronous. Version 0x00020000.
lineGetAgentCaps	Obtains the agent-related capabilities supported on the specified line device. Asynchronous. Version 0x00020000.
LineGetAgentGroupList	Obtains the list of agent groups into which an agent can log into on the automatic call distributor. Asynchronous. Version 0x00020000.
lineGetAgentStatus	Obtains the agent-related status on the specified address. Asynchronous. Version 0x00020000.
lineSetAgentActivity	Sets the agent activity code associated with a particular address. Asynchronous. Version 0x00020000.
lineSetAgentGroup	Sets the agent groups into which the agent is logged into on a particular address. Asynchronous. Version 0x00020000.
lineSetAgentState	Sets the agent state associated with a particular address. Asynchronous. Version 0x00020000.

Proxies

lineProxyMessage	Used by a registered proxy request handler to generate TAPI messages. Synchronous. Version 0x00020000.
lineProxyResponse	Indicates completion of a proxy request by a registered proxy handler. Synchronous. Version 0x00020000.

Quality of Service

lineSetCallQualityOfService	Requests a change of the quality of service parameters for an existing call. Asynchronous. Version 0x00020000.

Miscellaneous

lineSetCallData	Sets the **CallData** member of the LINECALLINFO structure. Asynchronous. Version 0x00020000.

lineSetCallTreatment	Sets the sounds the user hears when a call is unanswered or on hold. Asynchronous. Version 0x00020000. **lineSetLineDevStatus** Sets the line device status. Asynchronous. Version 0x00020000.

Phone Services

TAPI Initialization and Shutdown

phoneInitializeEx	Initializes the Telephony API phone abstraction for use by the invoking application. Synchronous.
phoneShutdown	Shuts down the application's use of the phone Telephony API. Synchronous.

Phone Version Negotiation

phoneNegotiateAPIVersion	Allows an application to negotiate an API version to use. Synchronous.

Opening and Closing Phone Devices

phoneOpen	Opens the specified phone device, giving the application either owner or monitor privileges. Synchronous.
phoneClose	Closes a specified open phone device. Synchronous.

Phone Status and Capabilities

phoneGetDevCaps	Returns the capabilities of a given phone device. Synchronous.
phoneGetID	Returns a device ID for the given device class associated with the specified phone device. Synchronous.
phoneGetIcon	Allows an application to retrieve an icon for display to the user. Synchronous.
phoneConfigDialog	Causes the provider of the specified phone device to display a dialog box that allows the user to configure parameters related to the phone device. Synchronous.

Hookswitch Devices

phoneSetHookSwitch	Sets the hookswitch mode of one or more of the hookswitch devices of an open phone device. Asynchronous.
phoneGetHookSwitch	Queries the hookswitch mode of a hookswitch device of an open phone device. Synchronous.
phoneSetVolume	Sets the volume of a hookswitch device's speaker of an open phone device. Asynchronous.
phoneGetVolume	Returns the volume setting of a hookswitch device's speaker of an

		open phone device. Synchronous.
	phoneSetGain	Sets the gain of a hookswitch device's mic of an open phone device. Asynchronous.
	phoneGetGain	Returns the gain setting of a hookswitch device's mic of an opened phone. Synchronous.
Display		
	phoneSetDisplay	Writes information to the display of an open phone device. Asynchronous.
	phoneGetDisplay	Returns the current contents of a phone's display. Synchronous.
Ring		
	phoneSetRing	Rings an open phone device according to a given ring mode. Asynchronous.
	phoneGetRing	Returns the current ring mode of an opened phone device. Synchronous.
Buttons		
	phoneSetButtonInfo	Sets the information associated with a button on a phone device. Asynchronous.
	phoneGetButtonInfo	Returns information associated with a button on a phone device. Synchronous.
Lamps		
	phoneSetLamp	Lights a lamp on a specified open phone device in a given lamp lighting mode. Asynchronous.
	phoneGetLamp	Returns the current lamp mode of the specified lamp. Synchronous.
Data Areas		
	phoneSetData	Downloads a buffer of data to a given data area in the phone device. Asynchronous.
	phoneGetData	Uploads the contents of a given data area in the phone device to a buffer. Synchronous.
Status		
	phoneSetStatusMessages	Specifies the status changes for which the application wants to be notified. Synchronous.
	phoneGetStatusMessages	Returns the status changes for which

	the application wants to be notified. Synchronous.
phoneGetStatus	Returns the complete status of an open phone device. Synchronous.

Assisted Telephony Services Functions

The Assisted Telephony Services functions are:

tapiRequestMakeCall	Submits a request to place a voice call.
tapiRequestMediaCall	Obsolete. Do not use.
tapiRequestDrop	Obsolete. Do not use.
tapiGetLocationInfo	Returns country code and city/area code information.

Extended Telephony Services Functions

The following tables list by category the extended telephony functions for both line and phone devices.

Extended Line Services

lineNegotiateExtVersion	Allows an application to negotiate an extension version to use with the specified line device. Asynchronous.
lineDevSpecific	Device-specific escape function. Synchronous.
lineDevSpecificFeature	Device-specific escape function to allow sending switch features to the switch. Asynchronous.

Extended Phone Services

phoneDevSpecific	Device-specific escape function to allow vendor-dependent extensions. Asynchronous.
PhoneNegotiateExtVersion	Allows an application to negotiate an extension version to use with the specified phone device. Synchronous.

Unicode Support

The following section contains information about support for Unicode.

Functions with Unicode (W) Versions

The following TAPI functions are implemented in Unicode (W) and ANSI (A) versions. In general, the implementation of the ANSI version calls the Unicode version and performs necessary conversions of ANSI parameters and structure fields to and from Unicode; the following table indicates the parameters that are converted.

Applications that explicitly call the generic (neither "W" or "A" suffix) version of a function will execute the ANSI version, for backward compatibility with previous versions of TAPI.

Note The entire Telephony Service Provider Interface (TSPI) is Unicode for version 2.0.

In the following table, references to string fields in TAPI structures consist of a portion of the field names. For example, the "Caller Address" in the **LINEFORWARD** structure is pointed to by a field named **dwCallerAddressOffset** and delimited by a field named **dwCallerAddressSize**; in the table, this string is identified simply as **CallerAddress**.

TAPI Function	Parameters and Structure Fields Converted in ANSI Version of Function
lineAddProvider	*lpszProviderName*
lineBlindTransfer	*lpszDestAddress*
lineConfigDialog	*lpszDeviceClass*
lineConfigDialogEdit	*lpszDeviceClass* **Note** Application must handle conversion of strings in *lpDeviceConfigIn* and *lpDeviceConfigOut*, if these are directly manipulated.
lineDial	*lpszDestAddress*

lineForward	*lpForwardList* (**LINEFORWARDLIST**)
	• *ForwardList* (**LINEFORWARD**)
	  • *CallerAddress*
	  • *DestAddress*
	lpCallParams (**LINECALLPARAMS**)
	• *OrigAddress*
	• *DisplayableAddress*
	• *CalledParty*
	• *Comment*
	• *TargetAddress*
	• *DeviceClass*
	• *CallingPartyID*

lineGatherDigits	*lpsDigits* *lpszTerminationDigits*
lineGenerateDigits	*lpszDigits*
lineGetAddressCaps	*lpAddressCaps* (**LINEADDRESSCAPS**)

	• *Address* • *CompletionMsgText* • *DeviceClasses* • *CallTreatmentList* (**LINECALLTREATMENTENTRY**) • *CallTreatmentName*
lineGetAddressID	*lpsAddress*
lineGetAddressStatus	*lpAddressStatus* (**LINEADDRESSSTATUS**) • *Forward* (**LINEFORWARD**) • *CallerAddress* • *DestAddress*
lineGetAgentActivityList	*lpAgentActivityList* (**LINEAGENTACTIVITYLIST**) • *List* (**LINEAGENTACTIVITYENTRY**) • *Name*
lineGetAgentCaps	*lpAgentCaps* (**LINEAGENTCAPS**) • *AgentHandlerInfo*
lineGetAgentGroupList	*lpAgentGroupListI*(**LINEAGENTGROUPLIST**) • *List* (**LINEAGENTGROUPENTRY**) • *Name*
lineGetAgentStatus	*lpAgentStatus* (**LINEAGENTSTATUS**) • *Activity* • *GroupList* (**LINEAGENTGROUPENTRY**) • *Name*
lineGetAppPriority	*lpszAppFilename* *lpExtensionName*
lineGetCallInfo	*lpCallInfo* (**LINECALLINFO**) • *CallerID* • *CallerIDName* • *CalledID* • *CalledIDName* • *ConnectID* • *ConnectedIDName* • *RedirectionID* • *RedirectionIDName* • *RedirectingID* • *RedirectingIDName* • *AppName* • *DisplayableAddress* • *CalledParty*

	• *Comment*
lineGetCountry	*lpLineCountryList* (**LINECOUNTRYLIST**) • *CountryList* (**LINECOUNTRYENTRY**) • *CountryName* • *SameAreaRule* • *LongDistanceRule* • *InternationalRule*
lineGetDevCaps	*lpLineDevCaps* (**LINEDEVCAPS**) • *ProviderInfo* • *SwitchInfo* • *LineName* • *TerminalText* • *DeviceClasses* **Note** *dwStringFormat* is obsolete.
LineGetDevConfig	*lpszDeviceClass* **Note** Application must handle conversion of strings in *lpDeviceConfig*, if these are directly manipulated.
LineGetIcon	*lpszDeviceClass*
lineGetID	*lpszDeviceClass* **Note** Application must handle conversion of strings in *lpDeviceID*, if these are directly manipulated.
LineGetLineDevStatus	*lpLineDevStatus* (**LINEDEVSTATUS**) • *AppInfo* (LINEAPPINFO) • *MachineName* • *UserName* • *ModuleFilename* • *FriendlyName*
lineGetProviderList	*lpProviderList* (**LINEPROVIDERLIST**) • *ProviderList* (**LINEPROVIDERENTRY**) • *ProviderFilename*
lineGetRequest	*lpRequestBuffer* (LINEREQMAKECALL • *szDestAddress* • *szAppName* • *szCalledParty* • *szComment*

lineGetTranslateCaps	*lpTranslateCaps* (**LINETRANSLATECAPS**) • *CardList* (**LINECARDENTRY**) • *CardName* • *SameAreaRule* • *LongDistanceRule* • *InternationalRule* • *LocationList* (**LINELOCATIONENTRY** • *LocationName* • *CityCode* • *LocalAccessCode* • *LongDistanceAccessCode* • *TollPrefixList* • *celCallWaiting*
lineHandoff	*lpszFileName*
lineInitializeEx	*lpszFriendlyAppName*
lineMakeCall	*lpszDestAddress* *lpCallParams* (**LINECALLPARAMS**) • *OrigAddress* • *DisplayableAddress* • *CalledParty* • *Comment* • *TargetAddress* • *DeviceClass* • *CallingPartyID*
lineOpen	*lpCallParams* (**LINECALLPARAMS**) • *OrigAddress* • *DisplayableAddress* • *CalledParty* • *Comment* • *TargetAddress* • *DeviceClass* • *CallingPartyID*
linePark	*lpszDirAddress* *lpNonDirAddress* (**VARSTRING**) • *String*
linePickup	*lpszDestAddress* *lpszGroupID*
linePrepareAddToConference	*lpCallParams* (**LINECALLPARAMS**) • *OrigAddress* • *DisplayableAddress* • *CalledParty* • *Comment* • *TargetAddress*

	• *DeviceClass* • *CallingPartyID*
lineRedirect	*lpszDestAddress*
lineSetAppPriority	*lpszAppFilename* *lpszExtensionName*
lineSetDevConfig	*lpszDeviceClass* **Note** Application must handle conversion of strings in *lpDeviceConfig*, if these are directly manipulated.
lineSetTollList	*lpszAddressIn*
lineSetupConference	*lpCallParams* (**LINECALLPARAMS**) • *OrigAddress* • *DisplayableAddress* • *CalledParty* • *Comment* • *TargetAddress* • *DeviceClass* • *CallingPartyID*
lineSetupTransfer	*lpCallParams* (**LINECALLPARAMS**) • *OrigAddress* • *DisplayableAddress* • *CalledParty* • *Comment* • *TargetAddress* • *DeviceClass* • *CallingPartyID*
lineTranslateAddress	*lpszAddressIn* *lpTranslateOutput* (**LINETRANSLATEOUTPUT**) • *DialableString* • *DisplayableString*
lineTranslateDialog	*lpszAddressIn*
lineUnpark	*lpszDestAddress*
phoneConfigDialog	*lpszDeviceClass*
phoneGetButtonInfo	*lpButtonInfo* (**PHONEBUTTONINFO**) • *ButtonText*
phoneGetDevCaps	*lpPhoneCaps* (**PHONECAPS**) • *ProviderInfo* • *PhoneInfo* • *PhoneName* • *DeviceClasses* **Note** *dwStringFormat* is obsolete.

phoneGetIcon	*lpszDeviceClass*
phoneGetID	*lpszDeviceClass* **Note** Application must handle conversion of strings in *lpDeviceID*, if these are directly manipulated.
phoneGetStatus	*lpPhoneStatus* (**PHONESTATUS**) • *OwnerName*
phoneInitializeEx	*lpszFriendlyAppName*
phoneSetButtonInfo	*lpButtonInfo* (**PHONEBUTTONINFO**) • *ButtonTest*
tapiGetLocationInfo	*lpszCountryCode* *lpszCityCode*
tapiRequestMakeCall	*lpszDestAddress* *lpszAppName* *lpszCalledParty* *lpszComment*

Functions without Unicode Versions

The following functions are provided only in a generic version without an "A" or "W" suffix.

TAPI Function	Comments
lineAccept	The memory pointed to by *lpsUserUserInfo* is presumed to contain binary data for end-to-end transfer. The application must provide data in a form ready for transmission.
lineAddToConference	–
lineAgentSpecific	The memory pointed to by *lpParams* is private between the application and agent handler. The application must provide data in the form specified in the agent handler extension definition.
lineAnswer	The memory pointed to by *lpsUserUserInfo* is presumed to contain binary data for end-to-end transfer. The application must provide data in a form ready for transmission.
lineClose	–
lineCompleteCall	–
lineCompleteTransfer	–
lineConfigProvider	–
lineDeallocateCall	–
lineDevSpecific	The memory pointed to by *lpParams* is private between the application and service provider. The application must provide data in the form specified in the service provider extension definition.
lineDevSpecificFeature	The memory pointed to by *lpParams* is private between the application and service provider. The application must provide data in the form specified in the service provider extension definition.
lineDrop	The memory pointed to by *lpsUserUserInfo* is presumed to contain binary data for end-to-end transfer. The application must provide data in a form ready for transmission.
lineGenerateTone	–
lineGetCallStatus	–
lineGetConfRelatedCalls	–
lineGetNewCalls	–

lineGetMessage	–
lineGetNumRings	–
lineGetStatusMessages	–
lineHold	–
lineMonitorDigits	–
lineMonitorMedia	–
lineMonitorTones	–
lineNegotiateAPIVersion	–
lineNegotiateExtVersion	–
lineProxyMessage	–
lineProxyResponse	The fields in the **LINEPROXYREQUEST** structure are *always* Unicode.
lineRegisterRequestRecipient	–
lineReleaseUserUserInfo	–
lineRemoveFromConference	–
lineRemoveProvider	–
lineSecureCall	–
lineSendUserUserInfo	The memory pointed to by *lpsUserUserInfo* is presumed to contain binary data for end-to-end transfer. The application must provide data in a form ready for transmission.
lineSetAgentActivity	–
lineSetAgentGroup	**Note** Group names are ignored.
lineSetAgentState	–
lineSetAppSpecific	–
lineSetCallData	The memory pointed to by *lpCallData* is in a format specified by the application or a group of cooperating applications. The format of the data is beyond the scope of TAPI and is not converted by TAPI.
lineSetCallParams	–
lineSetCallPrivilege	–
lineSetCallQualityOfService	The format of data in the provider-specific portion of the QOS structure is beyond the scope of TAPI and is not converted by TAPI.
lineSetCallTreatment	–
lineSetCurrentLocation	–
lineSetLineDevStatus	–
lineSetMediaControl	–
lineSetMediaMode	–
lineSetNumRings	–

lineSetStatusMessages	–
lineSetTerminal	–
lineShutdown	–
lineSwapHold	–
lineUncompleteCall	–
lineUnhold	–
phoneClose	–
phoneDevSpecific	The memory pointed to by *lpParams* is private between the application and service provider. The application must provide data in the form specified in the service provider extension definition.
phoneGetData	The memory pointed to by *lpData* is private between the application and service provider. The application must process data in the form specified in the service provider definition.
phoneGetDisplay	The memory pointed to by *lpDisplay* is private between the application and service provider. The application must process data in the form specified in the service provider definition.
phoneGetGain	–
phoneGetHookSwitch	–
phoneGetLamp	–
phoneGetMessage	–
phoneGetRing	–
phoneGetStatusMessages	–
phoneGetVolume	–
phoneNegotiateAPIVersion	–
phoneNegotiateExtVersion	–
phoneOpen	–
phoneSetData	The memory pointed to by *lpParams* is private between the application and service provider. The application must provide data in the form specified in the service provider definition.
phoneSetDisplay	The memory pointed to by *lpDisplay* is private between the application and service provider. The application must provide data in the form specified in the service provider definition.
phoneSetGain	–
phoneSetHookSwitch	–
phoneSetLamp	–

phoneSetRing	–
phoneSetStatusMessages	–
phoneSetVolume	–
phoneShutdown	–
tapiRequestDrop	This function is obsolete and unavailable to Microsoft® Win32® API applications.
tapiRequestMediaCall	This function is obsolete and unavailable to Microsoft Win32 applications.

Reference

This section includes the telephony API function, message, structure, and constant references for line devices, phone devices, and assisted telephony.

Functions

This section contains an alphabetical list of the line device, phone device, and assisted telephony functions in the Telephony applications programming interface (API).

The information for each function includes a list of the valid call states on entry of the function and typical call state transitions when the request completes. Note that the actual states in which a function may be performed may be further limited by the capabilities of the service provider. Applications must check the **dwCallFeatures** field in the **LINECALLSTATUS** structure, the **dwAddressFeatures** field in the **LINEADDRESSSTATUS** structure, and the **dwLineFeatures** field in the **LINEDEVSTATUS** structure to determine whether or not a function is permitted at that point in time.

Line Device Functions

This section contains the functions for line devices.

lineAccept Quick Info

Overview

Group

The **lineAccept** function accepts the specified offered call. It may optionally send the specified user-to-user information to the calling party.

```
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 containing user-to-user information to be sent to the remote party as part of the call accept. This pointer can be left NULL if no user-to-user information is to be sent. User-to-user information is only sent if supported by the underlying network (see **LINEDEVCAPS**). The protocol discriminator field for the user-to-user information, if required, should appear as the first byte of the buffer pointed to by *lpsUserUserInfo*, and must be accounted for in *dwSize*.

dwSize

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

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful, or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONUNAVAIL, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED, LINEERR_INVALPOINTER, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_USERUSERINFOTOOBIG.

Remarks

The **lineAccept** function is used in telephony environments like Integrated Services Digital Network (ISDN) that allow alerting associated with incoming calls to be separate from the initial offering of the call. When a call comes in, it is first offered. For some small amount of time, the application may have the option to reject the call using **lineDrop**, redirect the call to another station using **lineRedirect**, answer the call using **lineAnswer**, or accept the call using **lineAccept**. After a call has been successfully accepted, alerting at both the called and calling device begins. After a call has been accepted by an application, the call state typically transitions to *accepted*.

Alerting is reported to the application by the LINE_LINEDEVSTATE message with the *ringing* indication.

The **lineAccept** function may also be supported by non-ISDN service providers. The call state transition to *accepted* can be used by other applications as an indication that another application has claimed responsibility for the call and has presented the call to the user.

The application has the option to send user-to-user information at the time of the accept. Even if user-to-user information is sent, there is no guarantee that the network will deliver this information to the calling party. An application should consult a line's device capabilities to determine whether call accept is available.

For information about the listing of service dependencies, see Service Dependencies..

See Also

LINE_REPLY, **lineAnswer**, **LINEDEVCAPS**, **lineDrop**, **lineRedirect**

lineAddProvider Quick Info

Overview

Group

The **lineAddProvider** function installs a new Telephony Service Provider into the Telephony system.

```
LONG lineAddPr ovider(

  LPCSTR lpszProviderFilename,
  HWND hwndOwner,
  LPDWORD lpdwPermanentProviderID
);
```

Parameters

lpszProviderFilename

A pointer to a NULL-terminated string containing the path of the service provider to be added.

hwndOwner

A handle to a window to which any dialogs which 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 ID of the newly installed service provider.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INIFILECORRUPT, LINEERR_NOMEM, LINEERR_INVALPARAM, LINEERR_NOMULTIPLEINSTANCE, LINEERR_INVALPOINTER, LINEERR_OPERATIONFAILED.

Remarks

During this function call, TAPI checks to ensure that it can access the service provider by calling its **TSPI_providerInstall** function; if this is unsuccessful (if the DLL or function cannot be found, or if **TSPI_providerInstall** returns an error), the function fails and the provider is not added to the telephony system. If this is successful, and the Win32 Telephony system is active (one or more applications have called **lineInitialize** or **lineInitializeEx**), TAPI does not attempt to launch the newly-added service provider. Instead, in order to activate the new service provider, TAPI issues a message to restart Windows. When the activation succeeds, applications will be informed of any new devices created by way of LINE_CREATE or PHONE_CREATE messages, or by a LINE_LINEDEVSTATE message requesting reinitialization (if the application does not support the CREATE messages).

This function copies no files—not the service provider DLL itself nor any supporting files; it is the responsibility of the application managing the addition of the provider to ensure that the provider is installed in a directory where it can be found by TAPI (for example, \WINDOWS, \WINDOWS\SYSTEM, or elsewhere on the path), and that all other files necessary for operation.

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so; the function will work the same way for all

applications.

See Also

LINE_CREATE, LINE_LINEDEVSTATE, **lineInitialize**, **lineInitializeEx**, PHONE_CREATE

lineAddToConference Quick Info

Overview

Group

The **lineAddToConference** function adds the call specified by *hConsultCall* to the conference call specified by *hConfCall*.

```
LONG lineAddToConference(

  HCALL hConfCall,
  HCALL hConsultCall
);
```

Parameters

hConfCall

A handle to the conference call. The application must be an owner of this call. Any monitoring (media, tones, digits) on a conference call applies only to the *hConfCall*, not to the individual participating calls. Call state of *hConfCall* must be *onHoldPendingConference* or *onHold*.

hConsultCall

A handle to the call to be added to the conference call. The application must be an owner of this call. This call cannot be a parent of another conference or a participant in any conference. Depending on the device capabilities indicated in **LINEADDRESSCAPS**, the *hConsultCall* may not necessarily have been established using **lineSetupConference** or **linePrepareAddToConference**. The call state of *hConsultCall* must be *connected*, *onHold*, *proceeding*, or *ringback*. Many PBXs allow calls to be added to conferences before they are actually answered.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful, or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_CONFERENCEFULL, LINEERR_NOTOWNER, LINEERR_INVALCONFCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSTATE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

If LINEERR_INVALCALLHANDLE is returned, the specified call handle for the added call is invalid, *hConsultCall* is a parent of another conference or already a participant in a conference, *hConsultCall* cannot be added for other reasons (such as, it must have been established using **lineSetupConference** or **linePrepareAddToConference**), or *hConsultCall* and *hConfCall* are calls on different open lines.

The call handle of the added party remains valid after adding the call to a conference. Its state typically changes to *conferenced* while the state of the conference call typically becomes *connected.* Using **lineGetConfRelatedCalls**, you can obtain a list of call handles that are part of the same conference call as the specified call. The specified call is either a conference call or a participant call in a conference call. New handles are generated for those calls for which the application does not already have handles, and the application is granted monitor privilege to those calls. The handle to an individual participating call can be used later to remove that party from the conference call using **lineRemoveFromConference**.

Note that if **lineGetConfRelatedCalls** is called immediately after **lineAddToConference**, it may not return a complete list of related calls because TAPI waits to receive a LINE_CALLSTATE message indicating that the call has entered LINECALLSTATE_CONFERENCED before it considers the call to actually be part of the conference (that is, the conferenced state is confirmed by the service provider). Once the application has received the LINE_CALLSTATE message, **lineGetConfRelatedCalls** returns complete information. Note that all calls that are part of a conference must exist on the same open line.

The call states of the calls participating in a conference are not independent. For example, when dropping a conference call, all participating calls may automatically become idle. An application should consult the line's device capabilities to determine what form of conference removal is available. The application should track the LINE_CALLSTATE messages to determine what happened to the calls involved.

The conference call is established either by **lineSetupConference** or **lineCompleteTransfer**. The call added to a conference is typically established using **lineSetupConference** or **linePrepareAddToConference**. Some switches may allow adding arbitrary calls to the conference, and such a call may have been set up using **lineMakeCall** and be on (hard) hold. The application may examine the **dwAddrCapFlags** field of the **LINEADDRESSCAPS** structure to determine the permitted operations.

See Also

LINE_CALLSTATE, **LINEADDRESSCAPS**, **lineCompleteTransfer**, **lineGetConfRelatedCalls**, **lineMakeCall**, **linePrepareAddToConference**, **lineSetupConference**, **lineRemoveFromConference**

lineAgentSpecific Quick Info

Overview

Group

The **lineAgentSpecific** function allows the application to access proprietary handler-specific functions of the agent handler associated with the address. The meaning of the extensions are specific to the agent handler. Each set of agent-related extensions is identified by a universally unique 128-bit extension ID which must be obtained, along with the specification for the extension, from the promulgator of that extension (usually the author of the agent handler software on the telephony server). The list of extensions supported by the agent handler is obtained from the **LINEAGENTCAPS** structure returned by **lineGetAgentCaps**.

```
LONG lineAgentSpecific(

  HLINE hLine,
  DWORD dwAddressID,
  DWORD dwAgentExtensionIDIndex,
  LPVOID lpParams,
  DWORD dwSize
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

An address on the open line device.

dwAgentExtensionIDIndex

The position in the *ExtensionIDList* structure in **LINEAGENTCAPS** of the agent handler extension being invoked.

lpParams

A pointer to a memory area used to hold a parameter block. The format of this parameter block is device specific and its contents are passed by TAPI to and from the agent handler application on the telephony server. This parameter block must specify the function to be invoked and include sufficient room for any data to be returned.

dwSize

The size in bytes of the parameter block area.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALADDRESSID, LINEERR_INVALAGENTID, LINEERR_INVALLINEHANDLE, LINEERR_INVALPARAM, LINEERR_INVALPOINTER, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_STRUCTURETOOSMALL, LINEERR_UNINITIALIZED.

Additional return values are specific to the agent handler.

Remarks

This operation is part of the Extended Telephony services. It provides access to an agent handler-specific feature without defining its meaning.

This function provides a generic parameter profile. The interpretation of the parameter structure is handler specific. Indications and replies sent back to the application that are handler specific should use the LINE_AGENTSPECIFIC message.

An agent handler can provide access to handler-specific functions by defining parameters for use with this function. Applications that want to make use of these extensions should consult the vendor-specific documentation that describes what extensions are defined. An application that relies on these extensions will typically not be able to work with other agent handler environments.

See Also

LINEAGENTCAPS, **lineGetAgentCaps**

lineAnswer Quick Info

Overview

Group

The **lineAnswer** function answers the specified offering call.

```
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 containing user-to-user information to be sent to the remote party at the time of answering the call. This pointer can be left NULL if no user-to-user information is to be sent. User-to-user information is only sent if supported by the underlying network (see **LINEDEVCAPS**). The protocol discriminator field for the user-to-user information, if required, should appear as the first byte of the buffer pointed to by *lpsUserUserInfo*, and must be accounted for in *dwSize*.

dwSize

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

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INUSE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSTATE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_USERUSERINFOTOOBIG, LINEERR_NOTOWNER.

Remarks

When a new call arrives, applications with an interest in the call are sent a LINE_CALLSTATE message to provide the new call handle and to inform the application about the call's state and the privileges to the new call (such as monitor or owner). The application with owner privilege for the call can answer this call using **lineAnswer**. After the call has been successfully answered, the call typically transitions to the *connected* state. Initially, only one application is given owner privilege to the inbound call.

In some telephony environments (like ISDN), where user alerting is separate from call offering, the application may have the option to accept a call prior to answering or to reject or redirect the *offering* call.

If a call comes in (is offered) at the time another call is already active, the new call is connected to by

invoking **lineAnswer**. The effect this has on the existing active call depends on the line's device capabilities. The first call may be unaffected, it may automatically be dropped, or it may automatically be placed on hold. The appropriate LINE_CALLSTATE messages report state transitions to the application about both calls.

In a bridged situation, if a call is connected but in the LINECONNECTEDMODE_INACTIVE state, it may be joined using the **lineAnswer** function.

The application has the option to send user-to-user information at the time of the answer. Even if user-to-user information can be sent, there is no guarantee that the network will deliver this information to the calling party. An application should consult a line's device capabilities to determine whether sending user-to-user information upon answering the call is available.

See Also

LINE_CALLSTATE, LINE_REPLY, **LINEDEVCAPS**

lineBlindTransfer Quick Info

Overview

Group

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

```
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 identifying where the call is to be transferred to. The destination address uses the standard dialable number format.

dwCountryCode

The country code of the destination. This is used by the implementation to select the call progress protocols for the destination address. If a value of zero is specified, a default call-progress protocol defined by the service provider is used.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_INVALCOUNTRYCODE, LINEERR_INVALCALLSTATE, LINEERR_INVALPOINTER, LINEERR_NOMEM, LINEERR_OPERATIONUNAVAIL, LINEERR_NOTOWNER, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALADDRESS, LINEERR_UNINITIALIZED, LINEERR_ADDRESSBLOCKED, LINEERR_OPERATIONFAILED.

Remarks

If LINEERR_INVALADDRESS is returned, no dialing has occurred.

Blind transfer differs from a consultation transfer in that no consultation call is made visible to the application. After the blind transfer successfully completes, the specified call is typically cleared from the application's line, and it transitions to the *idle* state. Note that the application's call handle remains valid after the transfer has completed. The application must deallocate its handle when it is no longer interested in the transferred call. It uses **lineDeallocateCall** for this purpose.

See Also

LINE_REPLY, **lineDeallocateCall**

lineClose Quick Info

Overview

Group

The **lineClose** function closes the specified open line device.

```
LONG lineClose(

  HLINE hLine
);
```

Parameters

hLine

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL.

Remarks

If an application calls **lineClose** while it still has active calls on the opened line, the application's ownership of these calls is revoked. If the application was the sole owner of these calls, the calls are dropped as well. It is good programming practice for an application to dispose of the calls it owns on an opened line by explicitly relinquishing ownership and/or by dropping these calls prior to closing the line.

If the close was successful, a LINE_LINEDEVSTATE message is sent to all applications that are monitoring the line status of open/close changes. Outstanding asynchronous replies are suppressed.

Service providers may find it useful or necessary to forcibly reclaim line devices from an application that has the line open. This may be useful to prevent a misbehaved application from monopolizing the line device for too long. If this happens, a LINE_CLOSE message is sent to the application, specifying the line handle of the line device that was closed.

The **lineOpen** function allocates resources to the invoking application, and applications may be prevented from opening a line if resources are unavailable. Therefore, an application that only occasionally uses a line device (such as for making outbound calls) should close the line to free resources and allow other applications to open the line.

See Also

LINE_CLOSE, LINE_LINEDEVSTATE, **lineOpen**

lineCompleteCall Quick Info

Overview

Group

The **lineCompleteCall** function specifies how a call that could not be connected normally should be completed instead. The network or switch may not be able to complete a call because network resources are busy or the remote station is busy or doesn't answer. The application can request that the call be completed in one of a number of ways.

```
LONG lineCompleteCall(

  HCALL hCall,
  LPDWORD lpdwCompletionID,
  DWORD dwCompletionMode,
  DWORD dwMessageID
);
```

Parameters

hCall

A handle to the call whose completion is requested. The application must be an owner of the call. The call state of *hCall* must be *busy, ringback*.

lpdwCompletionID

A pointer to a DWORD-sized memory location. The completion ID is used to identify individual completion requests in progress. A completion ID becomes invalid and may be reused after the request completes or after an outstanding request is canceled.

dwCompletionMode

The way in which the call is to be completed. Note that *dwCompletionMode* is allowed to have only a single flag set. This parameter uses the following LINECALLCOMPLMODE_ constants:

LINECALLCOMPLMODE_CAMPON

Queues the call until the call can be completed. The call remains in the *busy* state while queued.

LINECALLCOMPLMODE_CALLBACK

Requests the called station to return the call when it returns to *idle*.

LINECALLCOMPLMODE_INTRUDE

Adds the application to the existing physical call at the called station (barge in).

LINECALLCOMPLMODE_MESSAGE

Leave a short predefined message for the called station ("Leave Word Calling"). The message to be sent is specified by *dwMessageID*.

dwMessageID

The message that is to be sent when completing the call using LINECALLCOMPLMODE_MESSAGE. This ID selects the message from a small number of predefined messages.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_COMPLETIONOVERRUN, LINEERR_NOMEM, LINEERR_INVALCALLCOMPLMODE, LINEERR_NOTOWNER, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALMESSAGEID, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED.

Remarks

This function is considered complete when the request has been accepted by the network or switch; not when the request is fully completed in the way specified. After this function completes, the call typically transitions to *idle*. When the called station or network enters a state where the call can be completed as requested, the application will be notified by a LINE_CALLSTATE message with the call state equal to *offering*. The call's **LINECALLINFO** record lists the reason for the call as CALLCOMPLETION and provide the completion ID as well. It is possible to have multiple call completion requests outstanding at any given time; the maximum number is device dependent. The completion ID is also used to refer to each individual request so requests can be canceled by calling **lineUncompleteCall**.

See Also

LINE_CALLSTATE, LINE_REPLY, **LINECALLINFO**, **lineUncompleteCall**

lineCompleteTransfer Quick Info

Overview

Group

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

```
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 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, this parameter is ignored by TAPI.

dwTransferMode

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

LINETRANSFERMODE_TRANSFER

Resolve the initiated transfer by transferring the initial call to the consultation call.

LINETRANSFERMODE_CONFERENCE

Resolve the initiated transfer by conferencing all three parties into a three-way conference call. A conference call is created and returned to the application.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_NOTOWNER, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCONSULTCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALTRANSFERMODE, LINEERR_RESOURCEUNAVAIL,

LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Remarks

The LINE_REPLY message sent in response to a call to the **lineCompleteTransfer** function is based on the status of the call specified by the *hCall* parameter.

This operation completes the transfer of the original call, *hCall*, to the party currently connected by *hConsultCall*. The consultation call will typically have been dialed on the consultation call allocated as part of **lineSetupTransfer**, but it may be any call to which the switch is capable of transferring *hCall*.

The transfer request can be resolved either as a transfer or as a three-way conference call. When resolved as a transfer, the parties connected by *hCall* and *hConsultCall* are connected to each other, and both *hCall* and *hConsultCall* are typically cleared from the application's line and transition to the *idle* state. Note that the application's call handle remains valid after the transfer has completed. The application must deallocate its handle with **lineDeallocateCall** when it is no longer interested in the transferred call.

When resolved as a conference, all three parties enter into a conference call. Both existing call handles remain valid but will transition to the *conferenced* state. A conference call handle will be created and returned, and it will transition to the *connected* state.

If **lineGetConfRelatedCalls** is called immediately after **lineCompleteTransfer** with the result that the calls are conferenced, **lineGetConfRelatedCalls** may not return a complete list of related calls. This is because TAPI waits to receive a LINE_CALLSTATE message indicating that the call has entered LINECALLSTATE_CONFERENCED before it considers the call to actually be part of the conference. That is, it waits for the service provider to confirm the conferenced state. Once the application has received the LINE_CALLSTATE message, **lineGetConfRelatedCalls** returns complete information.

It may also be possible to perform a blind transfer of a call using **lineBlindTransfer**.

See Also

LINE_CALLSTATE, LINE_REPLY, **lineBlindTransfer**, **lineDeallocateCall**, **lineGetConfRelatedCalls**, **lineSetupTransfer**

lineConfigDialog Quick Info

Overview

Group

The **lineConfigDialog** function causes the provider of the specified line device to display a dialog (attached to *hwndOwner* of the application) to allow the user to configure parameters related to the line device.

```
LONG lineConfigDialog(

  DWORD dwDeviceID,
  HWND hwndOwner,
  LPCSTR lpszDeviceClass
);
```

Parameters

dwDeviceID

The line device to be configured.

hwndOwner

A handle to a window to which the dialog is to be attached. Can be NULL to indicate that any window created during the function should have no owner window.

lpszDeviceClass

A pointer to a NULL-terminated string that identifies a device class name. This device class allows the application to select a specific subscreen of configuration information applicable to that device class. This parameter is optional and can be left NULL or empty, in which case the highest level configuration is selected.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INUSE, LINEERR_OPERATIONFAILED, LINEERR_INVALDEVICECLASS, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPARAM, LINEERR_UNINITIALIZED, LINEERR_INVALPOINTER, LINEERR_OPERATIONUNAVAIL, LINEERR_NODEVICE.

Remarks

The **lineConfigDialog** function causes the service provider to display a modal dialog (attached to *hwndOwner* of the application) to allow the user to configure parameters related to the line specified by *dwDeviceID*. The *lpszDeviceClass* parameter allows the application to select a specific subscreen of configuration information applicable to the device class in which the user is interested; the permitted strings are the same as for **lineGetID**. For example, if the line supports the Comm API, passing "COMM" as *lpszDeviceClass* causes the provider to display the parameters related specifically to Comm (or, at least, start at the corresponding point in a multilevel configuration dialog chain, so the user doesn't have to "dig" to find the parameters of interest).

The *lpszDeviceClass* parameter would be "tapi/line" , "", or NULL to cause the provider to display the highest level configuration for the line.

See Also

lineGetID

lineConfigDialogEdit Quick Info

Overview

Group

The **lineConfigDialogEdit** function causes the provider of the specified line device to display a dialog (attached to *hwndOwner* of the application) to allow the user to configure parameters related to the line device.

```
LONG lineConfigDialogEdit(

  DWORD dwDeviceID,
  HWND hwndOwner,
  LPCSTR lpszDeviceClass,
  LPVOID const lpDeviceConfigIn,
  DWORD dwSize,
  LPVARSTRING lpDeviceConfigOut
);
```

Parameters

dwDeviceID

The line device to be configured.

hwndOwner

A handle to a window to which the dialog is to be attached. Can be NULL to indicate that any window created during the function should have no owner window.

lpszDeviceClass

A pointer to a NULL-terminated string that identifies a device class name. This device class allows the application to select a specific subscreen of configuration information applicable to that device class. This parameter is optional and can be left NULL or empty, in which case the highest level configuration is selected.

lpDeviceConfigIn

A pointer to the opaque configuration data structure that was returned by **lineGetDevConfig** (or a previous invocation of **lineConfigDialogEdit**) in the variable portion of the **VARSTRING** structure.

dwSize

The number of bytes in the structure pointed to by *lpDeviceConfigIn*. This value will have been returned in the **dwStringSize** field in the **VARSTRING** structure returned by **lineGetDevConfig** or a previous invocation of **lineConfigDialogEdit**.

lpDeviceConfigOut

A pointer to the memory location of type **VARSTRING** where the device configuration structure is returned. Upon successful completion of the request, this location is filled with the device configuration. The **dwStringFormat** field in the **VARSTRING** structure will be set to STRINGFORMAT_BINARY. Prior to calling **lineGetDevConfig** (or a future invocation of **lineConfigDialogEdit**), the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible

return values are:

LINEERR_BADDEVICEID, LINEERR_OPERATIONFAILED, LINEERR_INVALDEVICECLASS, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPARAM, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NODRIVER, LINEERR_OPERATIONUNAVAIL, LINEERR_NOMEM, LINEERR_NODEVICE.

Remarks

If LINEERR_STRUCTURETOOSMALL is returned, the **dwTotalSize** field of the **VARSTRING** structure pointed to by *lpDeviceConfigOut* does not specify enough memory to contain the entire configuration structure. The **dwNeededSize** field has been set to the amount required. To the extent that user entries were reflected in information that could not be returned due to insufficient space, those edits are lost; applications should therefore allocate the maximum amount of space that may be needed by the device class to return its configuration structure (for more information, see documentation for the device class).

The **lineConfigDialogEdit** function causes the service provider to display a modal dialog (attached to *hwndOwner* of the application) to allow the user to configure parameters related to the line specified by *dwDeviceID*.

The *lpszDeviceClass* parameter allows the application to select a specific subscreen of configuration information applicable to the device class in which the user is interested; the permitted strings are the same as for **lineGetID**. For example, if the line supports the Comm API, passing "COMM" as *lpszDeviceClass* causes the provider to display the parameters related specifically to Comm (or, at least, start at the corresponding point in a multilevel configuration dialog chain, so the user doesn't have to "dig" to find the parameters of interest).

The *lpszDeviceClass* parameter would be "tapi/line" , "", or NULL to cause the provider to display the highest level configuration for the line.

The difference between this function and **lineConfigDialog** is the source of the parameters to edit and the result of the editing. In **lineConfigDialog**, the parameters edited are those currently in use on the device (or set for use on the next call), and any changes made have (to the maximum extent possible) an immediate impact on any active connection; also, the application must use **lineGetDevConfig** to fetch the result of parameter changes from **lineConfigDialog**. With **lineConfigDialogEdit**, the parameters to edit are passed in from the application, and the results are returned to the application, with *no* impact on active connections; the results of the editing are returned with this function, and the application does not need to call **lineGetDevConfig**. Thus, **lineConfigDialogEdit** permits an application to provide the ability for the user to set up parameters for future calls without having an impact on any active call. Note, however, the output of this function can be passed to **lineSetDevConfig** to affect the current call or next call.

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so; the function will work the same way for all applications.

See Also

lineConfigDialog, **lineGetDevConfig**, **lineGetID**, **lineSetDevConfig**, **VARSTRING**

lineConfigProvider Quick Info

Overview

Group

The **lineConfigProvider** function causes a service provider to display its configuration dialog.

```
LONG lineConfigProvider(

  HWND hwndOwner,
  DWORD dwPermanentProviderID
);
```

Parameters

hwndOwner

A handle to a window to which the configuration dialog (displayed by **TSPI_providerConfig**) will be attached. Can be NULL to indicate that any window created during the function should have no owner window.

dwPermanentProviderID

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INIFILECORRUPT, LINEERR_NOMEM, LINEERR_INVALPARAM, LINEERR_OPERATIONFAILED.

Remarks

This is basically a straight pass-through to **TSPI_providerConfig**.

Although this is a new function that older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so; the function will work the same way for all applications.

lineDeallocateCall Quick Info

Overview

Group

The **lineDeallocateCall** function deallocates the specified call handle.

```
LONG lineDeallocateCall(

  HCALL hCall
);
```

Parameters

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 except when the application is the sole owner of the call and the call is not in the *idle* state. The call handle is no longer valid after it has been deallocated.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSTATE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

The deallocation does not affect the call state of the physical call. It does, however, release internal resources related to the call.

In API versions less than 0x00020000, if the application is the sole owner of a call and the call is not in the *idle* state, LINEERR_INVALCALLSTATE is returned. In this case, the application can first drop the call using **lineDrop** and deallocate its call handle afterwards. An application that has monitor privilege for a call can always deallocate its handle for the call.

In API versions 0x0002000 and greater, the sole owner of the call *can* deallocate its handle even though the call is not in the *idle* state. (This allows for distributed control of the call in a client/server environment.) Be aware that leaving the call without an owner may result in the user being unable to terminate the call if there are monitoring applications open preventing TAPI from calling **TSPI_lineCloseCall**. Use this feature only if the application can determine that the call can be controlled externally by the user (see LINEADDRCAPFLAGS_CLOSEDROP).

In API versions less than 0x00020000, when the **lineDeallocateCall** function deallocates a call handle, it also suspends further processing of any outstanding LINE_REPLY messages for the call. An application must be designed *not* to wait indefinitely for LINE_REPLY messages for each corresponding call to an asynchronous function if it also uses the **lineDeallocateCall** function to deallocate handles.

In API versions 0x0002000 and greater, **lineDeallocateCall** does not suspend outstanding LINE_REPLY messages; every asynchronous function that returns a *dwRequestID* to the application always results in the delivery of the associated LINE_REPLY message unless the application calls **lineShutdown**.

See Also

LINE_REPLY, **lineDrop**, **lineShutdown**

lineDevSpecific Quick Info

Overview

Group

The **lineDevSpecific** function enables service providers to provide access to features not offered by other TAPI functions. The meaning of the extensions are device specific, and taking advantage of these extensions requires the application to be fully aware of them.

```
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 ID on the given line device.

hCall

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

lpParams

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

dwSize

The size in bytes of the parameter block area.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful, or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESSID, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Additional return values are device specific.

Remarks

This operation is part of the Extended Telephony services. It provides access to a device-specific feature without defining its meaning. This operation is only available if the application has successfully negotiated a device-specific extension version.

This function provides a generic parameter profile. The interpretation of the parameter structure is device specific. Whether *dwAddressID* and/or *hCall* are expected to be valid is device-specific. If specified, they must belong to *hLine*. Indications and replies sent back the application that are device specific should use the LINE_DEVSPECIFIC message.

A service provider can provide access to device-specific functions by defining parameters for use with this function. Applications that want to make use of these device-specific extensions should consult the device-specific (in this case, vendor-specific) documentation that describes what extensions are defined. An application that relies on these device-specific extensions will typically not be able to work with other service provider environments.

See Also

LINE_DEVSPECIFIC, LINE_REPLY

lineDevSpecificFeature Quick Info

Overview

Group

The **lineDevSpecificFeature** function enables service providers to provide access to features not offered by other TAPI functions. The meaning of these extensions are device specific, and taking advantage of these extensions requires the application to be fully aware of them.

```
LONG lineDevSpecificFeature(

  HLINE hLine,
  DWORD dwFeature,
  LPVOID lpParams,
  DWORD dwSize
);
```

Parameters

hLine

A handle to the line device.

dwFeature

The feature to invoke on the line device. This parameter uses the PHONEBUTTONFUNCTION_ constants.

lpParams

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

dwSize

The size of the buffer in bytes.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALFEATURE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALLINEHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Additional return values are device specific.

Remarks

This operation is part of the Extended Telephony services. It provides access to a device-specific feature without defining its meaning. This operation is only available if the application has successfully negotiated a device-specific extension version.

This function provides the application with phone feature-button emulation capabilities. When an application invokes this operation, it specifies the equivalent of a button-press event. This method of

invoking features is device dependent, as TAPI does not define their meaning. Note that an application that relies on these device-specific extensions will typically not work with other service provider environments.

Note also that the structure pointed to by *lpParams* should not contain any pointers because they would not be properly translated (thunked) when running a 16-bit application in a 32-bit version of TAPI and vice versa.

See Also

LINE_REPLY

lineDial Quick Info

Overview

Group

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

```
LONG lineDial(

  HCALL hCall,
  LPCSTR lpszDestAddress,
  DWORD dwCountryCode
);
```

Parameters

hCall

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

lpszDestAddress

The destination to be dialed using the standard dialable number format.

dwCountryCode

The country code of the destination. This is used by the implementation to select the call progress protocols for the destination address. If a value of zero is specified, a service provider-defined default call progress protocol is used.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_ADDRESSBLOCKED, LINEERR_INVALPOINTER, LINEERR_DIALBILLING, LINEERR_NOMEM, LINEERR_DIALDIALTONE, LINEERR_NOTOWNER, LINEERR_DIALPROMPT, LINEERR_OPERATIONFAILED, LINEERR_DIALQUIET, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_UNINITIALIZED, LINEERR_INVALCOUNTRYCODE.

Remarks

If LINEERR_INVALADDRESS is returned, no dialing has been done. If LINEERR_DIALBILLING, LINEERR_DIALQUIET, LINEERR_DIALDIALTONE, or LINEERR_DIALPROMPT is returned, none of the actions otherwise performed by **lineDial** have occurred. For example, none of the dialable addresses prior to the offending character has been dialed, no hookswitch state has changed, and so on.

The **lineDial** function is used for dialing on an existing call appearance. For example, after a call has been set up for transfer or conference, a consultation call is automatically allocated, and the **lineDial** function would be used to perform the dialing of this consultation call. Note that **lineDial** may be invoked multiple times in the course of multistage dialing, if the line's device capabilities allow it. Also, multiple addresses may be provided in a single dial string separated by CRLF. Service providers that provide inverse multiplexing can establish individual physical calls with each of the addresses and can return a single call handle to the aggregate of all calls to the application. All addresses would use the same

country code.

Dialing is considered complete after the address has been passed to the service provider; not after the call is finally connected. Service providers that provide inverse multiplexing may allow multiple addresses to be provided at once. The service provider sends LINE_CALLSTATE messages to the application to inform it about the progress of the call. To abort a call attempt while a call is being established, the invoking application should use **lineDrop**.

An application can set the *lpszDestAddress* parameter of the **lineDial** function to the address of an empty string to indicate that dialing is complete, but only if the previous calls to the **lineMakeCall** and **lineDial** functions have had the strings specified by *lpszDestAddress* terminated with semicolons.

See Also

LINE_CALLSTATE, LINE_REPLY, **lineDrop**, **lineMakeCall**

lineDrop Quick Info

Overview

Group

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

```
LONG lineDrop(

  HCALL hCall,
  LPCSTR lpsUserUserInfo,
  DWORD dwSize
);
```

Parameters

hCall

A handle to the call to be dropped. The application must be an owner of the call. The call state of *hCall* can be any state except *idle*.

lpsUserUserInfo

A pointer to a string containing user-to-user information to be sent to the remote party as part of the call disconnect. This pointer can be left NULL if no user-to-user information is to be sent. User-to-user information is only sent if supported by the underlying network (see **LINEDEVCAPS**). The protocol discriminator field for the user-to-user information, if required, should appear as the first byte of the buffer pointed to by *lpsUserUserInfo*, and must be accounted for in *dwSize*.

dwSize

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

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_NOTOWNER, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_USERUSERINFOTOOBIG, LINEERR_INVALCALLSTATE, LINEERR_UNINITIALIZED.

Remarks

When invoking **lineDrop**, related calls may sometimes be affected as well. For example, dropping a conference call may drop all individual participating calls. LINE_CALLSTATE messages are sent to the application for all calls whose call state is affected. A dropped call typically transitions to the *idle* state. Invoking **lineDrop** on a call in the *offering* state rejects the call. Not all telephone networks provide this capability.

A call in the *onholdpending* state will typically revert to the *connected* state. When dropping the consultation call to the third party for a conference call or when removing the third party in a previously established conference call, the provider (and switch) may release the conference bridge and revert the

call back to a normal two-party call. If this is the case, *hConfCall* transitions to the *idle* state, and the only remaining participating call will transition to the *connected* state. Some switches automatically "unhold" the other call.

The application has the option to send user-to-user information at the time of the drop. Even if user-to-user information can be sent, there is no guarantee that the network will deliver this information to the remote party.

Note that in various bridged or party-line configurations when multiple parties are on the call, **lineDrop** may not actually clear the call. For example, in a bridged situation, a **lineDrop** operation may possibly not actually drop the call because the status of other stations on the call may govern; instead, the call may simply be changed to the LINECONNECTEDMODE_INACTIVE mode if it remains *connected* at other stations.

See Also

LINE_CALLSTATE, LINE_REPLY, **LINEDEVCAPS**

lineForward Quick Info

Overview

Group

The **lineForward** function forwards calls destined for the specified address on the specified line, according to the specified forwarding instructions. When an originating address (*dwAddressID*) is forwarded, the specified incoming calls for that address are deflected to the other number by the switch. This function provides a combination of forward and do-not-disturb features. This function can also cancel forwarding currently in effect.

```
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 specified is to be forwarded. If TRUE, all addresses on the line are forwarded and *dwAddressID* is ignored; if FALSE, only the address specified as *dwAddressID* is forwarded.

dwAddressID

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

lpForwardList

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

dwNumRingsNoAnswer

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

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 that is being forwarded to, 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 will be set to NULL if no consultation call is created.

lpCallParams

A pointer to a structure of type **LINECALLPARAMS**. This pointer is 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 are used. Otherwise, the specified call parameters are used for establishing *hConsultCall*.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

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

Remarks

A successful forwarding indicates only that the request has been accepted by the service provider, not that forwarding is set up at the switch. A LINE_ADDRESSSTATE (forwarding) message provides confirmation for forwarding having been set up at the switch.

Forwarding of the address(es) remains in effect until this function is called again. The most recent forwarding list replaces the old one. Forwarding can be canceled by specifying a NULL pointer as *lpForwardList*. If a NULL destination address is specified for an entry in the forwarding list, the operation acts as a do-not-disturb.

Forwarding status of an address may also be affected externally; for example, by administrative actions at the switch or by a user from another station. It may not be possible for the service provider to be aware of this state change, and it may not be able to keep in synchronization with the forwarding state known to the switch.

Because a service provider may not know the forwarding state of the address "for sure" (that is, it may have been forwarded or unforwarded in an unknown way), **lineForward** will succeed unless it fails to set the new forwarding instructions. In other words, a request that all forwarding be canceled at a time that there is no forwarding in effect will be successful. This is because there is no "unforwarding"—you can only change the previous set of forwarding instructions.

The success or failure of this operation does not depend on the previous set of forwarding instructions, and the same is true when setting different forwarding instructions. The provider should "unforward everything" prior to setting the new forwarding instructions. Because this may take time in analog telephony environments, a provider may also want to compare the current forwarding with the new one, and only issue instructions to the switch to get to the final state (leaving unchanged forwarding unaffected).

Invoking **lineForward** when **LINEFORWARDLIST** has *dwNumEntries* set to zero has the same effect as providing a NULL *lpForwardList* parameter. It cancels all forwarding currently in effect.

See Also

LINE_ADDRESSSTATE, LINE_REPLY, **LINECALLPARAMS**, **LINEFORWARDLIST**

lineGatherDigits Quick Info

Overview

Group

The **lineGatherDigits** function initiates the buffered gathering of digits on the specified call. The application specifies a buffer in which to place the digits and the maximum number of digits to be collected.

```
LONG lineGatherDigits(

  HCALL hCall,
  DWORD dwDigitModes,
  LPSTR lpsDigits,
  DWORD dwNumDigits,
  LPCSTR lpszTerminationDigits,
  DWORD dwFirstDigitTimeout,
  DWORD dwInterDigitTimeout
);
```

Parameters

hCall

A handle to the call on which digits are to be gathered. The application must be an owner of the call. The call state of *hCall* can be any state.

dwDigitModes

The digit mode(s) to be monitored. Note that *dwDigitModes* is allowed to have one or more flags set. This parameter uses the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE

Detect digits as audible clicks that are the result of the use of rotary pulse sequences. Valid digits for pulse mode are '0' through '9'.

LINEDIGITMODE_DTMF

Detect digits as DTMF tones. Valid digits for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#'.

lpsDigits

A pointer to the buffer where detected digits are to be stored as ASCII characters. Digits may not show up in the buffer one at a time as they are collected. Only after a LINE_GATHERDIGITS message is received should the content of the buffer be assumed to be valid. If *lpsDigits* is NULL, the digit gathering currently in progress on the call is terminated and *dwNumDigits* is ignored. Otherwise, *lpsDigits* is assumed to have room for *dwNumDigits* digits.

dwNumDigits

The number of digits to be collected before a LINE_GATHERDIGITS message is sent to the application. The *dwNumDigits* parameter is ignored when *lpsDigits* is NULL. This function fails if *dwNumDigits* is zero.

lpszTerminationDigits

Specifies a NULL-terminated string of termination digits as ASCII characters. If one of the digits in the string is detected, that termination digit is appended to the buffer, digit collection is terminated, and the LINE_GATHERDIGITS message is sent to the application.

Valid characters for pulse mode are '0' through '9'. Valid characters for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#'. If this pointer is NULL, or if it points to an empty string, the function behaves as though no termination digits were supplied.

dwFirstDigitTimeout

The time duration in milliseconds in which the first digit is expected. If the first digit is not received in this timeframe, digit collection is aborted and a LINE_GATHERDIGITS message is sent to the application. The buffer only contains the NULL character, indicating that no digits were received and the first digit timeout terminated digit gathering. The call's line-device capabilities specify the valid range for this parameter or indicate that timeouts are not supported.

dwInterDigitTimeout

The maximum time duration in milliseconds between consecutive digits. If no digit is received in this timeframe, digit collection is aborted and a LINE_GATHERDIGITS message is sent to the application. The buffer only contains the digits collected up to this point followed by a NULL character, indicating that an interdigit timeout terminated digit gathering. The call's line-device capabilities specify the valid range for this parameter or indicate that timeouts are not supported.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_NOMEM, LINEERR_INVALCALLSTATE, LINEERR_NOTOWNER, LINEERR_INVALDIGITMODE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALDIGITS, LINEERR_OPERATIONFAILED, LINEERR_INVALPARAM, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_INVALTIMEOUT.

Remarks

Digit collection is terminated when the requested number of digits has been collected. It is also terminated when one of the digits detected matches a digit in *szTerminationDigits* before the specified number of digits has been collected. The detected termination digit is also placed in the buffer and the partial buffer is returned.

Another way of cancelling digit collection is when one of the timeouts expires. The *dwFirstDigitTimeout* expires if the first digit is not received in this time period. The *dwInterDigitTimout* expires if the second, third, (and so forth) digit is not received within that time period from the previously detected digit, and a partial buffer is returned.

A fourth method for terminating digit detection is by calling this function again while collection is in progress. The old collection session is terminated, any digits collected up to that point are copied to the buffer supplied from the previous call to this function, and the buffer is delivered when the LINE_GATHERDIGITS message is sent to the application. The mechanism for terminating digit gathering without initiating another gathering of the digits is by invoking this function with *lpsDigits* equal to NULL.

This function is considered successful if digit collection has been correctly initiated, not when digit collection has terminated. In all cases where a partial buffer is returned, valid digits (if any) are followed by an ASCII NULL character.

Although this function can be invoked in any call state, digits can typically only be gathered while the call is in the *connected* state.

The message LINE_GATHERDIGITS is sent only to the application that initiated the request. It is also sent when partial buffers are returned because of timeouts or matching termination digits, or when the request is canceled by another **lineGatherDigits** request on the call. Only one gather−digits request can

be active on a call at any given time across all applications that are owners of the call. Given the asynchronous behavior of the operation, an application that issues multiple **lineGatherDigits** in quick succession may be able to do so and receive several LINE_GATHERDIGITS messages later. While this would be unusual application behavior, the application will be able to count the number of these messages to allow cancel messages to be matched with the earlier requests. In any case, only the most recent request should be assumed to be valid.

An application can use **lineMonitorDigits** to enable or disable unbuffered digit detection. Each time a digit is detected in this fashion, a LINE_MONITORDIGITS message is sent to the application. Both buffered and unbuffered digit detection can be enabled for the same call simultaneously.

Gathering of digits on a conference call applies only to the *hConfCall*, not to the individual participating calls.

If the **lineGatherDigits** function is used to cancel a previous request to gather digits, the function copies any digits collected up to that point to the buffer specified in the original function call and sends a LINE_GATHERDIGITS message to the application, regardless of whether the *lpszDigits* parameter in the second call specifies a NULL or different address.

See Also

LINE_GATHERDIGITS, LINE_MONITORDIGITS, **lineMonitorDigits**

lineGenerateDigits Quick Info

Overview

Group

The **lineGenerateDigits** function initiates the generation of the specified digits on the specified call as inband tones using the specified signaling mode. Invoking this function with a NULL value for *lpszDigits* aborts any digit generation currently in progress. Invoking **lineGenerateDigits** or **lineGenerateTone** while digit generation is in progress aborts the current digit generation or tone generation and initiates the generation of the most recently specified digits or tone.

```
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. Note that *dwDigitMode* is allowed to have only a single flag set. This parameter uses the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE

Uses pulse/rotary for digit signaling. Valid digits for pulse mode are '0' through '9'.

LINEDIGITMODE_DTMF

Uses DTMF tones for digit signaling. Valid digits for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#'.

lpszDigits

A pointer to a NULL-terminated character buffer that contains the digits to be generated. Valid characters for pulse mode are '0' through '9' and ',' (comma). Valid characters for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#', and ',' (comma). A comma injects an extra delay between the signaling of the previous and next digits it separates. The duration of this pause is configuration defined, and the line's device capabilities indicate what this duration is. Multiple commas may be used to inject longer pauses. Invalid digits are ignored during the generation, rather than being reported as an error. The exclamation character (!) is a valid character in the string specifed by the *lpszDigits* parameter for both DTMF and pulse mode. This character causes a "hookflash" operation, as described for dialable addresses.

dwDuration

Both the duration in milliseconds of DTMF digits and pulse and DTMF inter-digit spacing. A value of zero will use a default value. The *dwDuration* parameter must be within the range specified by **MinDialParams** and **MaxDialParams** in **LINEDEVCAPS**. If out of range, the actual value is set to the nearest value in the range.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_NOTOWNER, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALDIGITMODE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

The **lineGenerateDigits** function is considered to have completed successfully when the digit generation has been successfully initiated, not when all digits have been generated. In contrast to **lineDial**, which dials digits in a network-dependent fashion, **lineGenerateDigits** guarantees to produce the digits as inband tones over the voice channel using DTMF or hookswitch dial pulses when using pulse. The **lineGenerateDigits** function is generally not suitable for making calls or dialing. It is intended for end-to-end signaling over an established call.

After all digits in *lpszDigits* have been generated, or after digit generation has been aborted or canceled, a LINE_GENERATE message is sent to the application.

Only one inband generation request (tone generation or digit generation) is allowed to be in progress per call across all applications that are owners of the call. Digit generation on a call is canceled by initiating either another digit generation request or a tone generation request. To cancel the current digit generation, the application can invoke **lineGenerateDigits** and specify NULL for the *lpszDigits* parameter.

Depending on the service provider and hardware, the application may monitor the digits it generates itself. If that is not desired, the application can disable digit monitoring while generating digits.

See Also

LINE_GENERATE, **LINEDEVCAPS**, **lineDial**, **lineGenerateTone**

lineGenerateTone Overview

Overview

Overview

The **lineGenerateTone** function generates the specified inband tone over the specified call. Invoking this function with a zero for *dwToneMode* aborts the tone generation currently in progress on the specified call. Invoking **lineGenerateTone** or **lineGenerateDigits** while tone generation is in progress aborts the current tone generation or digit generation and initiates the generation of the newly specified tone or digits.

```
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. A custom tone is composed of a set of arbitrary frequencies. A small number of standard tones are predefined. The duration of the tone is specified with *dwDuration* for both standard and custom tones. Note that *dwToneMode* can only have one bit set. If no bits are set (the value 0 is passed), tone generation is canceled. This parameter uses the following LINETONEMODE_ constants:

LINETONEMODE_CUSTOM

The tone is a custom tone, defined by the specified frequencies.

LINETONEMODE_RINGBACK

The tone to be generated is ring tone. The exact ringback tone is service provider defined.

LINETONEMODE_BUSY

The tone is a standard (station) busy tone. The exact busy tone is service provider defined.

LINETONEMODE_BEEP

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

LINETONEMODE_BILLING

The tone is billing information tone such as a credit card prompt tone. The exact billing tone is service provider defined.

A value of zero for *dwToneMode* cancels tone generation.

dwDuration

Duration in milliseconds during which the tone should be sustained. A value of zero for *dwDuration* uses a default duration for the specified tone. Default values are:

CUSTOM: infinite

RINGBACK: infinite

BUSY: infinite

BEEP: infinite

BILLING: fixed (single cycle)

dwNumTones

The number of entries in the *lpTones* array. This field is ignored if *dwToneMode* is not equal to CUSTOM.

lpTones

A pointer to a **LINEGENERATETONE** array that specifies the tone's components. This parameter is ignored for non-custom tones. If *lpTones* is a multi-frequency tone, the various tones are played simultaneously.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_NOTOWNER, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALPOINTER, LINEERR_OPERATIONFAILED, LINEERR_INVALTONEMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALTONE, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Remarks

The **lineGenerateTone** function is considered to have completed successfully when the tone generation has been successfully initiated, not when the generation of the tone is done. The function allows the inband generation of several predefined tones, such as ring back, busy tones, and beep. It also allows for the fabrication of custom tones by specifying their component frequencies, cadence, and volume. Because these tones are generated as inband tones, the call would typically have to be in the *connected* state for tone generation to be effective. When the generation of the tone is complete, or when tone generation is canceled, a LINE_GENERATE message is sent to the application.

Only one inband generation request (tone generation or digit generation) is allowed to be in progress per call across all applications that are owners of the call. This implies that if tone generation is currently in progress on a call, invoking **lineGenerateDigits** cancels the tone generation.

If the LINEERR_INVALPOINTER error value is returned, the specified *lpTones* parameter is invalid or the value specified by the *dwNumTones* parameter is too large.

See Also

LINE_GENERATE, **lineGenerateDigits**, **LINEGENERATETONE**

lineGetAddressCaps Overview

Overview

Overview

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

```
LONG lineGetAddressCaps(

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

Parameters

hLineApp

The handle to the application's registration with TAPI.

dwDeviceID

The line device containing the address to be queried.

dwAddressID

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

dwAPIVersion

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

dwExtVersion

The version number of the service provider-specific 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 is filled with address capabilities information. Prior to calling **lineGetAddressCaps**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_OPERATIONFAILED, LINEERR_INCOMPATIBLEEXTVERSION, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALADDRESSID, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALAPPHANDLE, LINEERR_UNINITIALIZED, LINEERR_INVALPOINTER,

LINEERR_OPERATIONUNAVAIL, LINEERR_NODRIVER, LINEERR_NODEVICE.

Remarks

Valid address IDs range from zero to one less than the number of addresses returned by **lineGetDevCaps**. The version number to be supplied is the version number that was returned as part of the line's device capabilities by **lineGetDevCaps**.

See Also

LINEADDRESSCAPS, **lineGetDevCaps**, **lineNegotiateAPIVersion**

lineGetAddressID Overview

Overview

Overview

The **lineGetAddressID** function returns the address ID associated with an address in a different format on the specified line.

```
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 where the address ID is returned.

dwAddressMode

The address mode of the address contained in *lpsAddress*. The *dwAddressMode* parameter is allowed to have only a single flag set. This parameter uses the following LINEADDRESSMODE_ constants:

LINEADDRESSMODE_DIALABLEADDR

The address is specified by its dialable address. The *lpsAddress* parameter is the dialable address or canonical address format.

lpsAddress

A pointer to a data structure holding the address assigned to the specified line device. The format of the address is determined by *dwAddressMode*. Because the only valid value is LINEADDRESSMODE_DIALABLEADDR, *lpsAddress* uses the common dialable number format and is NULL-terminated.

dwSize

The size of the address contained in *lpsAddress*.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALLINEHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALADDRESSMODE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALADDRESS, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Remarks

The **lineGetAddressID** function is used to map a phone number (address) assigned to a line device back to its *dwAddressID* in the range 0 to the number of addresses minus one returned in the line's device capabilities. The **lineMakeCall** function allows the application to make a call by specifying a line handle and an address on the line. The address can be specified as a *dwAddressID*, as a phone number, or as a device-specific name or identifier. Using a phone number may be practical in environments where a single line is assigned multiple addresses. Note that LINEADDRESSMODE_ADDRESSID may not be used with **lineGetAddressID**.

See Also

lineMakeCall

lineGetAddressStatus Overview

Overview

Overview

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

```
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 is 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** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESSID, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINEHANDLE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_OPERATIONUNAVAIL, LINEERR_OPERATIONFAILED.

See Also

LINEADDRESSSTATUS

lineGetAgentActivityList Overview

Overview

Overview

The **lineGetAgentActivityList** function obtains the identities of activities which the application can select using **lineSetAgentActivity** to indicate what function the agent is actually performing at the moment.

```
LONG lineGetAgentActivityList(

  HLINE hLine,
  DWORD dwAddressID,
  LPLINEAGENTACTIVITYLIST lpAgentActivityList
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

The address on the open line device whose agent status is to be queried.

lpAgentAcvitityList

A pointer to a variably sized structure of type **LINEAGENTACTIVITYLIST**. Upon successful completion of the request, this structure is filled with a list of the agent activity codes which can be selected using **lineSetAgentActivity**. Prior to calling **lineGetAgentActivityList**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALADDRESSID, LINEERR_OPERATIONFAILED, LINEERR_INVALAGENTID, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

See Also

LINEAGENTACTIVITYLIST, **lineSetAgentActivity**

lineGetAgentCaps Overview

Overview

Overview

The **lineGetAgentCaps** function obtains the agent-related capabilities supported on the specified line device. If a specific agent is named, the capabilities will include a listing of ACD groups into which the agent is permitted to log in.

```
LONG lineGetAgentCaps(

  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAddressID,
  DWORD dwAppAPIVersion,
  LPLINEAGENTCAPS lpAgentCaps
);
```

Parameters

hLineApp

The handle to the application's registration with TAPI.

dwDeviceID

The line device containing the address to be queried.

dwAddressID

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

dwAppAPIVersion

The highest API version supported by the application. This should *not* be the value negotiated using **lineNegotiateAPIVersion** on the device being queried.

lpAgentCaps

A pointer to a variably sized structure of type **LINEAGENTCAPS**. Upon successful completion of the request, this structure is filled with agent capabilities information. Prior to calling **lineGetAgentCaps**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_OPERATIONFAILED, LINEERR_INVALADDRESSID, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NODRIVER, LINEERR_UNINITIALIZED, LINEERR_NODEVICE.

See Also

LINEAGENTCAPS, **lineNegotiateAPIVersion**

lineGetAgentGroupList Overview

Overview

Overview

The **lineGetAgentGroupList** function obtains the identities of agent groups (combination of queue, supervisor, skill level, and so on) into which the agent currently logged in on the workstation is permitted to log into on the automatic call distributor.

```
LONG lineGetAgentGroupList(

  HLINE hLine,
  DWORD dwAddressID,
  LPLINEAGENTGROUPLIST lpAgentGroupList
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

The address on the open line device whose agent status is to be queried.

lpAgentGroupList

A pointer to a variably sized structure of type **LINEAGENTGROUPLIST**. Upon successful completion of the request, this structure is filled with a list of the agent groups into which the agent may log in at this time (which should include any groups into which the agent is already logged in, if any). Prior to calling **lineGetAgentGroupList**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALADDRESSID, LINEERR_INVALAGENTID, LINEERR_INVALLINEHANDLE, LINEERR_INVALPOINTER, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_STRUCTURETOOSMALL, LINEERR_UNINITIALIZED.

See Also

LINEAGENTGROUPLIST

lineGetAgentStatus Overview

Overview

Overview

The **lineGetAgentStatus** function obtains the agent-related status on the specified address.

```
LONG lineGetAgentStatus(

  HLINE hLine,
  DWORD dwAddressID,
  LPLINEAGENTSTATUS lpAgentStatus
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

The address on the open line device whose agent status is to be queried.

lpAgentStatus

A pointer to a variably sized structure of type **LINEAGENTSTATUS**. Upon successful completion of the request, this structure is filled with agent status information. Prior to calling **lineGetAgentStatus**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALADDRESSID, LINEERR_INVALLINEHANDLE, LINEERR_INVALPOINTER, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_STRUCTURETOOSMALL, LINEERR_UNINITIALIZED.

See Also

LINEAGENTSTATUS

lineGetAppPriority Overview

Overview

Overview

The **lineGetAppPriority** function allows an application to determine whether or not it is in the handoff priority list for a particular media mode or Assisted Telephony request mode, and, if so, its position in the priority list.

```
LONG lineGetAppPriority(

  LPCSTR lpszAppFilename,
  DWORD dwMediaMode,
  LPLINEEXTENSIONID const lpExtensionID,
  DWORD dwRequestMode,
  LPVARSTRING lpExtensionName,
  LPDWORD lpdwPriority
);
```

Parameters

lpszAppFilename

A pointer to a string containing the application executable module filename (without directory information). In API versions 0x00020000 and greater, the parameter can be in either long or 8.3 filename format. In API versions less than 0x00020000, the parameter must specify a filename in the 8.3 format; long filenames cannot be used.

dwMediaMode

The media mode for which the priority information is to be obtained. The value may be one of the LINEMEDIAMODE_ constants; only a single bit may be on. The value 0 should be used if checking application priority for Assisted Telephony requests.

lpExtensionID

A pointer to structure of type **LINEEXTENSIONID**. This parameter is ignored.

dwRequestMode

If the *dwMediaMode* parameter is 0, this parameter specifies the Assisted Telephony request mode for which priority is to be checked. It must be either LINEREQUESTMODE_MAKECALL or LINEREQUESTMODE_MEDIACALL. This parameter is ignored if *dwMediaMode* is non-zero.

lpExtensionName

This parameter is ignored.

lpdwPriority

A pointer to a DWORD-size memory location into which TAPI will write the priority of the application for the specified media or request mode. The value 0 will be returned if the application is not in the stored priority list and does not currently have any line device open with ownership requested of the specified media mode or having registered for the specified request mode.

In API versions less than 0x00020000, the value -1 (0xFFFFFFFF) is returned if the application has the line open for the specified media mode or has registered for the specified requests, but the application is not in the stored priority list (that is, it is in the temporary priority list only). In API versions 0x00020000 and greater, the value 0 is returned to indicate this condition.

Otherwise, the value will indicate the application's position in the list (with 1 being highest priority, and increasing values indicating decreasing priority).

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INIFILECORRUPT, LINEERR_INVALREQUESTMODE, LINEERR_INVALAPPNAME, LINEERR_NOMEM, LINEERR_INVALMEDIAMODE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL.

Remarks

If LINEERR_INVALMEDIAMODE is returned, the value specified in *dwMediaMode* is not 0, not a valid extended media mode, and not a LINEMEDIAMODE_ constant, or more than one bit is on in the parameter value.

Also, long filenames are now permitted for *lpszAppFilename*; 8.3 names are acceptable, but no longer required.

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so; the function will work the same way for all applications.

See Also

LINEEXTENSIONID, **VARSTRING**

lineGetCallInfo Overview

Overview

Overview

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

```
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, this structure is filled with call-related information. Prior to calling **lineGetCallInfo**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL.

Remarks

A separate **LINECALLINFO** structure exists for every inbound or outbound call. The structure contains primarily fixed information about the call. An application would typically be interested in checking this information when it receives its handle for a call by the LINE_CALLSTATE message, or each time it receives notification by a LINE_CALLINFO message that parts of the call information structure have changed. These messages supply the handle for the call as a parameter.

See Also

LINE_CALLINFO, LINE_CALLSTATE, **LINECALLINFO**

lineGetCallStatus Overview

Overview

Overview

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

```
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, this structure is filled with call status information. Prior to calling **lineGetCallStatus**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL.

Remarks

The **lineGetCallStatus** function returns the dynamic status of a call, whereas **lineGetCallInfo** returns primarily static information about a call. Call status information includes the current call state, detailed mode information related to the call while in this state (if any), as well as a list of the available API functions the application can invoke on the call while the call is in this state. An application would typically be interested in requesting this information when it receives notification about a call state change by the LINE_CALLSTATE message.

See Also

LINE_CALLSTATE, **LINECALLSTATUS**, **lineGetCallInfo**

lineGetConfRelatedCalls Overview

Overview

Overview

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

```
LONG lineGetConfRelatedCalls(

  HCALL hCall,
  LPLINECALLLIST lpCallList
);
```

Parameters

hCall

A handle to a call. This is 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 are returned in this structure. The first call in the list is the conference call, the other calls are the participant calls. The application is granted 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 is unchanged. Prior to calling **lineGetConfRelatedCalls**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_NOCONFERENCE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

The specified call can either be a conference call handle or a handle to a participant call. For example, a consultation call that has not yet been added to a conference call is not part of a conference. The first entry in the list that is returned is the conference call handle, the other handles are all the participant calls. The specified call is always one of the calls returned in the list. Calls in the list to which the application does not already have a call handle are assigned monitor privilege; privileges to calls for which the application already has handles are unchanged. The application can use **lineSetCallPrivilege** to change the privilege of the call.

Note that if **lineGetConfRelatedCalls** is called immediately after a call is added to a conference using **lineCompleteTransfer**, **lineGetConfRelatedCalls** may not return a complete list of related calls because TAPI waits to receive a LINE_CALLSTATE message indicating that the call has entered LINECALLSTATE_CONFERENCED before it considers the call to actually be part of the conference (that

is, the *conferenced* state is confirmed by the service provider). Once the application has received the LINE_CALLSTATE message, **lineGetConfRelatedCalls** returns complete information.

The application can invoke **lineGetCallInfo** and **lineGetCallStatus** for each call in the list to determine the call's information and status, respectively.

See Also

LINE_CALLSTATE, **lineCompleteTransfer**, **lineGetCallInfo**, **lineGetCallStatus**, **lineSetCallPrivilege**

lineGetCountry Overview

Overview

Overview

The **lineGetCountry** function fetches the stored dialing rules and other information related to a specified country, the first country in the country list, or all countries.

```
LONG lineGetCountry(

  DWORD dwCountryID,
  DWORD dwAPIVersion,
  LPLINECOUNTRYLIST lpLineCountryList
);
```

Parameters

dwCountryID

The country ID (*not* the country code) of the country for which information is to be obtained. If the value 1 is specified, information on the first country in the country list is obtained. If the value 0 is specified, information on all countries is obtained (which may require a great deal of memory−20Kbytes or more).

dwAPIVersion

The highest version of TAPI supported by the application (*not* necessarily the value negotiated by **lineNegotiateAPIVersion** on some particular line device).

lpLineCountryList

A pointer to a location to which a **LINECOUNTRYLIST** structure will be loaded. Prior to calling **lineGetCountry**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_NOMEM, LINEERR_INIFILECORRUPT, LINEERR_OPERATIONFAILED, LINEERR_INVALCOUNTRYCODE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER.

Remarks

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so. The function will work the same way for all applications.

See Also

LINECOUNTRYLIST, **lineNegotiateAPIVersion**

lineGetDevCaps Overview

Overview

Overview

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

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

Parameters

hLineApp

The handle to the application's registration with TAPI.

dwDeviceID

The line device to be queried.

dwAPIVersion

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

dwExtVersion

The version number of the service provider-specific extensions to be used. This number is obtained by **lineNegotiateExtVersion**. It can be left 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 is filled with line device capabilities information. Prior to calling **lineGetDevCaps**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_OPERATIONFAILED, LINEERR_INCOMPATIBLEEXTVERSION, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALAPPHANDLE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NODRIVER, LINEERR_OPERATIONUNAVAIL, LINEERR_NODEVICE.

Remarks

Before using **lineGetDevCaps**, the application must negotiate the API version number to use, and, if desired, the extension version to use.

The API and extension version numbers are those under which TAPI and the service provider must operate. If version ranges do not overlap, the application, API, or service-provider versions are incompatible and an error is returned.

One of the fields in the **LINEDEVCAPS** structure returned by this function contains the number of addresses assigned to the specified line device. The actual address IDs used to reference individual addresses vary from zero to one less than the returned number. The capabilities of each address may be different. Use **lineGetAddressCaps** for each available <*dwDeviceID*, *dwAddressID*> combination to determine the exact capabilities of each address.

See Also

LINEDEVCAPS, **lineGetAddressCaps**, **lineNegotiateAPIVersion**, **lineNegotiateExtVersion**

lineGetDevConfig Overview

Overview

Overview

The **lineGetDevConfig** function returns an "opaque" data structure object, the contents of which are specific to the line (service provider) and device class. The data structure object stores the current configuration of a media-stream device associated with the line device.

```
LONG lineGetDevConfig(

  DWORD dwDeviceID,
  LPVARSTRING lpDeviceConfig,
  LPCSTR lpszDeviceClass
);
```

Parameters

dwDeviceID

The line device to be configured.

lpDeviceConfig

A pointer to the memory location of type **VARSTRING** where the device configuration structure is returned. Upon successful completion of the request, this location is filled with the device configuration. The **dwStringFormat** field in the **VARSTRING** structure will be set to STRINGFORMAT_BINARY. Prior to calling **lineGetDevConfig**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

lpszDeviceClass

A pointer to a NULL-terminated ASCII string that specifies the device class of the device whose configuration is requested. Valid device class **lineGetID** strings are the same as those specified for the function.

Return Values

Returns zero if the function is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INVALDEVICECLASS, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_STRUCTURETOOSMALL, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_NODEVICE.

Remarks

Call states are device specific.

The **lineGetDevConfig** function can be used to retrieve a data structure from TAPI that specifies the configuration of a media stream device associated with a particular line device. For example, the contents of this structure could specify data rate, character format, modulation schemes, and error control protocol settings for a "datamodem" media device associated with the line.

Typically, an application will call **lineGetID** to identify the media stream device associated with a line, and then call **lineConfigDialog** to allow the user to set up the device configuration. It could then call

lineGetDevConfig, and save the configuration information in a phone book (or other database) associated with a particular call destination. When the user later wishes to call the same destination again, **lineSetDevConfig** can be used to restore the configuration settings selected by the user. The functions **lineSetDevConfig**, **lineConfigDialog**, and **lineGetDevConfig** can be used, in that order, to allow the user to view and update the settings.

The exact format of the data contained within the structure is specific to the line and media stream API (device class), is undocumented, and is undefined. The structure returned by this function cannot be directly accessed or manipulated by the application, but can only be stored intact and later used in **lineSetDevConfig** to restore the settings. The structure also cannot necessarily be passed to other devices, even of the same device class (although this may work in some instances, it is not guaranteed).

See Also

lineConfigDialog, **lineGetID**, **lineSetDevConfig**, **VARSTRING**

lineGetIcon Overview

Overview

Overview

The **lineGetIcon** function allows an application to retrieve a service line device-specific (or provider-specific) icon for display to the user.

```
LONG lineGetIcon(

  DWORD dwDeviceID,
  LPCSTR lpszDeviceClass,
  LPHICON lphIcon
);
```

Parameters

dwDeviceID

The line device whose icon is requested.

lpszDeviceClass

A pointer to a NULL-terminated string that identifies a device class name. This device class allows the application to select a specific sub-icon applicable to that device class. This parameter is optional and can be left NULL or empty, in which case the highest-level icon associated with the line device rather than a specified media stream device would be selected.

lphIcon

A pointer to a memory location in which the handle to the icon is returned.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALDEVICECLASS, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_NODEVICE.

Remarks

The **lineGetIcon** function causes the provider to return a handle (in *lphIcon*) to an icon resource (obtained from **LoadIcon**) that is associated with the specified line. The icon handle is for a resource associated with the provider. The application must use **CopyIcon** if it wishes to reference the icon after the provider is unloaded, which is unlikely to happen as long as the application has the line open.

The *lpszDeviceClass* parameter allows the provider to return different icons based on the type of service being referenced by the caller. The permitted strings are the same as for **lineGetID**. For example, if the line supports the Comm API, passing "COMM" as *lpszDeviceClass* causes the provider to return an icon related specifically to the Comm device functions of the service provider. The parameters "tapi/line", "", or NULL may be used to request the icon for the line service.

For applications using an API version less than 0x00020000, if the provider does not return an icon (whether because the given device class is invalid or the provider does not support icons), TAPI substitutes a generic Win32 Telephony line device icon. For applications using API version 0x00020000 or greater, TAPI substitutes the default line icon *only* if the *lpszDeviceClass* parameter is "tapi/line", "" or

NULL. For any other device class, if the given device class is not valid or the provider does not support icons for the class, **lineGetIcon** returns LINEERR_INVALDEVICECLASS.

See Also

lineGetID

lineGetID Overview

Overview

Overview

The **lineGetID** function returns a device ID for the specified device class associated with the selected line, address, or call.

```
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 ID 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 ID is returned. Upon successful completion of the request, this location is filled with the device ID. The format of the returned information depends on the method used by the device class API for naming devices. Prior to calling **lineGetID**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

lpszDeviceClass

A pointer to a NULL-terminated ASCII string that specifies the device class of the device whose ID is requested. Valid device class strings are those used in the SYSTEM.INI section to identify device classes.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALLINEHANDLE, LINEERR_NOMEM, LINEERR_INVALADDRESSID, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSELECT, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NODEVICE, LINEERR_UNINITIALIZED.

Remarks

The **lineGetID** function can be used to retrieve a line-device ID when given a line handle. This is useful after a line device has been opened using LINEMAPPER as a device ID in order to determine the real line-device ID of the opened line. This function can also be used to obtain the device ID of a phone device or media device (for device classes such as COM, wave, MIDI, phone, line, or NDIS) associated with a call, address or line. This ID can then be used with the appropriate API (such as phone, midi, wave) to select the corresponding media device associated with the specified call.

See Device Classes in TAPI for device class names.

A vendor that defines a device-specific media mode also needs to define the corresponding device-specific (proprietary) API to manage devices of the media mode. To avoid collisions on device class names assigned independently by different vendors, a vendor should select a name that uniquely identifies both the vendor and, following it, the media type. For example: "intel/video".

See Also

VARSTRING

lineGetLineDevStatus Overview

Overview

Overview

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

```
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, this structure is filled with the line's device status. Prior to calling **lineGetLineDevStatus**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL.

Remarks

An application uses **lineGetLineDevStatus** to query the line device for its current line status. This status information applies globally to all addresses on the line device. Use **lineGetAddressStatus** to determine status information about a specific address on a line.

See Also

LINEDEVSTATUS, **lineGetAddressStatus**

lineGetMessage Overview

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** for further details).

```
LONG lineGetMessage(

  HLINEAPP hLineApp,
  LPLINEMESSAGE lpMessage,
  DWORD dwTimeout
);
```

Parameters

hLineApp

The handle returned by **lineInitializeEx**. The application must have 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 will contain the next message which 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's time-out interval never elapses.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALAPPHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_NOMEM.

Remarks

If the **lineGetMessage** function has been called with a non-zero timeout and the application calls **lineShutdown** on another thread, this function will return immediately with LINEERR_INVALAPPHANDLE.

If the timeout expires (or was zero) and no message could be fetched from the queue, the function returns with the error LINEERR_OPERATIONFAILED.

See Also

lineInitializeEx, **LINEINITIALIZEEXPARAMS**, **LINEMESSAGE**, **lineShutdown**

lineGetNewCalls Overview

Overview

Overview

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 is granted monitor privilege to these calls.

```
LONG 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.

dwSelect

The selection of calls that are requested. Note that *dwSelect* can only have one bit set. This parameter uses the following LINECALLSELECT_ constants:

LINECALLSELECT_LINE

Selects calls on the specified line device. The *hLine* parameter must be a valid line handle; *dwAddressID* is ignored.

LINECALLSELECT_ADDRESS

Selects calls on the specified address on the specified line device. Both *hLine* and *dwAddressID* must be valid.

lpCallList

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESSID, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSELECT, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINEHANDLE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Remarks

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. The application is always given monitor privilege to the new call handles. Also, when opening a line, an application uses this function to become aware of existing calls.

The application can invoke **lineGetCallInfo** and **lineGetCallStatus** for each call in the list to determine the call's information and status, respectively. It can use **lineSetCallPrivilege** to change its privilege to owner.

See Also

LINECALLLIST, **lineGetCallInfo**, **lineGetCallStatus**, **lineSetCallPrivilege**

lineGetNumRings Overview

Overview

Overview

The **lineGetNumRings** function determines the number of rings an inbound call on the given address should ring prior to answering the call.

```
LONG lineGetNumRings(

  HLINE hLine,
  DWORD dwAddressID,
  LPDWORD lpdwNumRings
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

An address on the line device.

lpdwNumRings

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESSID, LINEERR_OPERATIONFAILED, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Remarks

The **lineGetNumRings** and **lineSetNumRings** functions, when used in combination, provide a mechanism to support the implementation of toll-saver features across multiple independent applications.

An application that receives a handle for a call in the *offering* state and a LINE_LINEDEVSTATE *ringing* message should wait a number of rings equal to the number returned by **lineGetNumRings** before answering the call in order to honor the toll-saver settings across all applications. The **lineGetNumRings** function returns the minimum of all application's number of rings specified by **lineSetNumRings**. Because this number may vary dynamically, an application should invoke **lineGetNumRings** each time it has the option to answer a call. If no application has called **lineSetNumRings**, the number or rings returned is 0xFFFFFFFF. A separate LINE_LINEDEVSTATE *ringing* message is sent to the application for each ring cycle.

If call classification is performed by TAPI of answering all calls of unknown media mode and filtering the media stream, TAPI honors this number as well.

Note that this operation is purely informational and does not in itself affect the state of any calls on the line device.

See Also

LINE_LINEDEVSTATE, **lineSetNumRings**

lineGetProviderList Overview

Overview

Overview

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

```
LONG lineGetProviderList(

  DWORD dwAPIVersion,
  LPLINEPROVIDERLIST lpProviderList
);
```

Parameters

dwAPIVersion

The highest version of TAPI supported by the application (*not* necessarily the value negotiated by **lineNegotiateAPIVersion** on some particular line device).

lpProviderList

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_NOMEM, LINEERR_INIFILECORRUPT, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL.

Remarks

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so. The function will work the same way for all applications.

See Also

lineNegotiateAPIVersion, **LINEPROVIDERLIST**

lineGetRequest Overview

Overview

Overview

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

```
LONG 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. Note that *dwRequestMode* can only have one bit set. This parameter uses the following LINEREQUESTMODE_ constants:

LINEREQUESTMODE_MAKECALL

A **tapiRequestMakeCall** request.

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 placed in the buffer depends on the request mode. The application-allocated buffer is assumed to be of sufficient size to hold the request.

If *dwRequestMode* is LINEREQUESTMODE_MAKECALL, interpret the content of the request buffer using the **LINEREQMAKECALL** structure.

If *dwRequestMode* is LINEREQUESTMODE_MEDIACALL, interpret the content of the request buffer using the **LINEREQMEDIACALL** structure.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALAPPHANDLE, LINEERR_NOTREGISTERED, LINEERR_INVALPOINTER, LINEERR_OPERATIONFAILED, LINEERR_INVALREQUESTMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_NOREQUEST.

Remarks

A telephony-enabled application can request that a call be placed on its behalf by invoking **tapiRequestMakeCall**. These requests are queued by TAPI and the highest priority application that has registered to handle the request is sent a LINE_REQUEST message with indication of the mode of the request that is pending. Typically, this application is the user's call-control application. The LINE_REQUEST message indicates that zero or more requests may be pending for the registered application to process; after receiving LINE_REQUEST, it is the responsibility of the recipient application to call **lineGetRequest** until LINEERR_NOREQUEST is returned, indicating that no more requests are

pending.

Next, the call-control application that receives this message invokes **lineGetRequest**, specifying the request mode and a buffer that is large enough to hold the request. The call-control application then interprets and executes the request.

After execution of **lineGetRequest**, TAPI purges the request from its internal queue, making room available for a subsequent request. It is therefore possible for a new LINE_REQUEST message to be received immediately upon execution of **lineGetRequest**, should the same or another application issue another request. It is the responsibility of the request recipient application to handle this scenario by some mechanism (for example, by making note of the additional LINE_REQUEST and deferring a subsequent **lineGetRequest** until processing of the preceding request completes, by getting the subsequent request and buffer as necessary, or by another appropriate means).

Note that the subsequent LINE_REQUEST should not be ignored because it will not be repeated by TAPI.

See Also

LINE_REQUEST, **LINEREQMAKECALL**, **tapiRequestMakeCall**

lineGetStatusMessages Overview

Overview

Overview

The **lineGetStatusMessages** function enables an application to query which notification messages the application is set up to receive for events related to status changes for the specified line or any of its addresses.

```
LONG lineGetStatusMessages(

  HLINE hLine,
  LPDWORD lpdwLineStates,
  LPDWORD lpdwAddressStates
);
```

Parameters

hLine

A handle to the line device.

lpdwLineStates

A bit array that identifies for which line device status changes a message is to be sent to the application. If a flag is TRUE, that message is enabled; if FALSE, it is disabled. Note that multiple flags can be set. This parameter uses the following LINEDEVSTATE_ constants:

LINEDEVSTATE_OTHER

Device-status items other than those listed below have changed. The application should check the current device status to determine which items have 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 containing this constant. For example, in the United States, service providers send a message with this constant every six seconds.

LINEDEVSTATE_CONNECTED

The line was previously disconnected and is now connected to TAPI.

LINEDEVSTATE_NUMCOMPLETIONS

The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_DISCONNECTED

This line was previously connected and is now disconnected from TAPI.

LINEDEVSTATE_MSGWAITON

The "message waiting" indicator is turned on.

LINEDEVSTATE_MSGWAITOFF

The "message waiting" indicator is turned off.

LINEDEVSTATE_INSERVICE

The line is connected to TAPI. This happens when TAPI is first activated or when the line wire is physically plugged in and in service at the switch while TAPI is active.

LINEDEVSTATE_OUTOFSERVICE

The line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE

Maintenance is being performed on the line at the switch. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_OPEN

The line has been opened by some application.

LINEDEVSTATE_CLOSE

The line has been closed by some application.

LINEDEVSTATE_NUMCALLS

The number of calls on the line device has changed.

LINEDEVSTATE_TERMINALS

The terminal settings have changed.

LINEDEVSTATE_ROAMMODE

The roam mode of the line device has changed.

LINEDEVSTATE_BATTERY

The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL

The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC

The line's device-specific information has changed.

LINEDEVSTATE_REINIT

Items have changed in the configuration of line devices. To become aware of these changes (for example, the appearance of new line devices) the application should reinitialize its use of TAPI. The *hDevice* parameter of the LINE_LINEDEVSTATE message is left NULL for this state change as it applies to any of the lines in the system.

LINEDEVSTATE_LOCK

The locked status of the line device has changed.

LINEDEVSTATE_REMOVED

The device is being removed from the system by the service provider (most likely through user action, through a control panel or similar utility). A LINE_LINEDEVSTATE message with this value will normally be immediately followed by a LINE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in LINEERR_NODEVICE being returned to the application. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

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. Multiple flags can be set. This

parameter uses the following LINEADDRESSSTATE_ constants:

LINEADDRESSSTATE_OTHER

Address-status items other than those listed below have changed. The application should check the current address status to determine which items have changed.

LINEADDRESSSTATE_DEVSPECIFIC

The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO

The address has changed to idle (it is now in use by zero stations).

LINEADDRESSSTATE_INUSEONE

The address has changed from being idle or from being in use by many bridged stations to being in use by just one station.

LINEADDRESSSTATE_INUSEMANY

The monitored or bridged address has changed from being in use by one station to being used by more than one station.

LINEADDRESSSTATE_NUMCALLS

The number of calls on the address has changed. This is the result of 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 has changed, including the number of rings for determining a "no answer" condition. The application should check the address status to determine details about the address's current forwarding status.

LINEADDRESSSTATE_TERMINALS

The terminal settings for the address have changed.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALLINEHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

TAPI defines a number of messages that notify applications about events occurring on lines and addresses. An application may not be interested in receiving all address and line status change messages. The **lineSetStatusMessages** function can be used to select which messages the application wants to receive. By default, address status and line status reporting is disabled.

See Also

LINE_CLOSE, LINE_LINEDEVSTATE, **lineSetStatusMessages**

lineGetTranslateCaps Overview

Overview

Overview

The **lineGetTranslateCaps** function returns address translation capabilities.

```
LONG lineGetTranslateCaps(

  HLINEAPP hLineApp,
  DWORD dwAPIVersion,
  LPLINETRANSLATECAPS lpTranslateCaps
);
```

Parameters

hLineApp

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

dwAPIVersion

The highest version of TAPI supported by the application (*not* necessarily the value negotiated by **lineNegotiateAPIVersion** on some particular line device).

lpTranslateCaps

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

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

See Also

lineInitializeEx, **lineNegotiateAPIVersion**, **LINETRANSLATECAPS**

lineHandoff Overview

Overview

Overview

The **lineHandoff** function gives ownership of the specified call to another application. The application can be either specified directly by its filename or indirectly as the highest priority application that handles calls of the specified media mode.

```
LONG 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 ASCII string. If this pointer parameter is non-NULL, it contains the filename of the application that is the target of the handoff. If NULL, the handoff target is the highest priority application that has opened the line for owner privilege for the specified media mode. A valid filename does not include the path of the file.

dwMediaMode

The media mode 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 is ignored if *lpszFileName* is not NULL. Only a single flag may be set in the *dwMediaMode* parameter at any one time. This parameter uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN

The target application is the one that handles calls of unknown media mode (unclassified calls).

LINEMEDIAMODE_INTERACTIVEVOICE

The target application is the one that handles calls with the interactive voice media mode (live conversations).

LINEMEDIAMODE_AUTOMATEDVOICE

Voice energy is present on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

The target application is the one that handles calls with the data modem media mode.

LINEMEDIAMODE_G3FAX

The target application is the one that handles calls with the group 3 fax media mode.

LINEMEDIAMODE_TDD

The target application is the one that handles calls with the TDD (Telephony Devices for the Deaf) media mode.

LINEMEDIAMODE_G4FAX

The target application is the one that handles calls with the group 4 fax media mode.

LINEMEDIAMODE_DIGITALDATA

The target application is the one that handles calls that are digital data calls.

LINEMEDIAMODE_TELETEX

The target application is the one that handles calls with the teletex media mode.

LINEMEDIAMODE_VIDEOTEX

The target application is the one that handles calls with the videotex media mode.

LINEMEDIAMODE_TELEX

The target application is the one that handles calls with the telex media mode.

LINEMEDIAMODE_MIXED

The target application is the one that handles calls with the ISDN mixed media mode.

LINEMEDIAMODE_ADSI

The target application is the one that handles calls with the ADSI (Analog Display Services Interface) media mode.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALMEDIAMODE, LINEERR_TARGETNOTFOUND, LINEERR_INVALPOINTER, LINEERR_TARGETSELF, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_NOTOWNER.

Remarks

The **lineHandoff** function returns LINEERR_TARGETSELF if the calling application attempted an indirect handoff (that is, set the *lpszFileName* parameter to NULL) and TAPI determined that the application is itself the highest priority application for the given media mode. If LINEERR_TARGETNOTFOUND is returned, a target for the call handoff was not found. This may occur if the named application did not open the same line with the LINECALLPRIVILEGE_OWNER bit in the *dwPrivileges* parameter of **lineOpen**. Or, in the case of media-mode handoff, no application has opened the same line with the LINECALLPRIVILEGE_OWNER bit in the *dwPrivileges* parameter of **lineOpen** and with the media mode specified in the *dwMediaMode* parameter having been specified in the *dwMediaModes* parameter of **lineOpen**.

Call handoff allows ownership of a call to be passed among applications. There are two types of handoff. In the first type, if the application knows the filename of the target application, it can simply specify the filename of that application. If an instance of the target application has opened the line device, ownership of the call will be passed to the other application; otherwise, the handoff will fail and an error is returned. This form of handoff will succeed if the call handle is handed off to the same file name as the application requesting the handoff.

The second type of handoff is based on media mode. In this case, the application indirectly specifies the target application by means of a media mode. The highest priority application that has currently opened the line device for that media mode is the target for the handoff. If there is no such application, the handoff fails and an error is returned.

The **lineHandoff** function does not change the media mode of a call. To change the media mode of a call, the application should use **lineSetMediaMode** on the call, specifying the new media mode. This changes the call's media as stored in the call's **LINECALLINFO** structure.

If handoff is successful, the receiving application will receive a LINE_CALLSTATE message for the call. This message indicates that the receiving application has owner privilege to the call (*dwParam3*). In addition, the number of owners and/or monitors for the call may have changed. This is reported by the LINE_CALLINFO message, and the receiving application can then invoke **lineGetCallStatus** and **lineGetCallInfo** to retrieve more information about the received call.

The receiving application should first check the media mode in **LINECALLINFO**. If only a single media mode flag is set, the call is officially of that media mode, and the application can act accordingly. If UNKNOWN and other media mode flags are set, then the media mode of the call is officially UNKNOWN but is assumed to be of one of the media modes for which a flag is set in **LINECALLINFO**. The application should assume that it ought to probe for the highest priority media mode.

If the probe succeeds (either for that media mode or for another one), the application should set the media mode field in **LINECALLINFO** to just the single media mode that was recognized. If the media mode is for that media mode, the application can act accordingly; otherwise, if it makes a determination for another media mode, it must first hand off the call to that media mode.

If the probe fails, the application should clear the corresponding media mode flag in **LINECALLINFO** and hand off the call, specifying *dwMediaMode* as LINEMEDIAMODE_UNKNOWN. It should also deallocate its call handle (or revert back to monitoring).

If none of the media modes succeeded in making a determination, only the UNKNOWN flag will remain set in the media mode field of **LINECALLINFO** at the time the media application attempts to hand off the call back to UNKNOWN. The final **lineHandoff** will fail if the application is the only remaining owner of the call. This informs the application that it should drop the call and deallocate its handle, in which case the call is abandoned. The privileges of the invoking application to the call are unchanged by this operation, but the application can change its privileges to a call with **lineSetCallPrivilege**.

See Also

LINECALLINFO, **lineGetCallStatus**, **lineOpen**, **lineSetCallPrivilege**, **lineSetMediaMode**

lineHold Overview

Overview

Overview

The **lineHold** function places the specified call on hold.

```
LONG lineHold(

  HCALL hCall
);
```

Parameters

hCall

A handle to the call to be placed on hold. The application must be an owner of the call. The call state of *hCall* must be *connected*.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_RESOURCEUNAVAIL, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED.

Remarks

The call on hold is temporarily disconnected allowing the application to use the line device for making or answering other calls. The **lineHold** function performs a so-called "hard hold" of the specified call (as opposed to a "consultation call"). A call on hard hold typically cannot be transferred or included in a conference call, but a consultation call can. Consultation calls are initiated using **lineSetupTransfer**, **lineSetupConference**, or **linePrepareAddToConference**.

After a call has been successfully placed on hold, the call state typically transitions to *onHold*. A held call is retrieved by **lineUnhold**. While a call is on hold, the application may receive LINE_CALLSTATE messages about state changes of the held call. For example, if the held party hangs up, the call state may transition to *disconnected*.

In a bridged situation, a **lineHold** operation may possibly not actually place the call on hold, because the status of other stations on the call may govern (for example, attempting to "hold" a call when other stations are participating will not be possible); instead, the call may simply be changed to the LINECONNECTEDMODE_INACTIVE mode if it remains *connected* at other stations.

See Also

LINE_CALLSTATE, **linePrepareAddToConference**, **lineSetupConference**, **lineSetupTransfer**, **lineUnhold**

lineInitialize Overview

Overview

Overview

The **lineInitialize** function is obsolete. It continues to be exported by TAPI.DLL and TAPI32.DLL for backward compatibility with applications using API versions 0x00010003 and 0x00010004.

Applications using API version 0x00020000 or greater must use **lineInitializeEx** instead.

For Windows 95 applications only

The **lineInitialize** function initializes the application's use of TAPI.DLL for subsequent use of the line abstraction. It registers the application's specified notification mechanism and returns the number of line devices available to the application. A line device is any device that provides an implementation for the line-prefixed functions in the Telephony API.

```
LONG 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 ASCII string that contains only displayable ASCII characters. If this parameter is not NULL, it contains an application-supplied name for the application. This name is provided in the **LINECALLINFO** structure to indicate, in a user-friendly way, which application originated, or originally accepted or answered the call. This information can be useful for call logging purposes. If *lpszAppName* is NULL, the application's filename is used instead.

lpdwNumDevs

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

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

Remarks

If LINEERR_REINIT is returned and TAPI reinitialization has been requested (for example as a result of adding or removing a Telephony service provider), then **lineInitialize** requests are rejected with this error until the last application shuts down its usage of the API (using **lineShutdown**). At that time, the new configuration becomes effective and applications are once again permitted to call **lineInitialize**. If the LINEERR_INVALPARAM error value is returned, the specified *hInstance* parameter is invalid.

The application can refer to individual line devices by using line device IDs that range from zero to *dwNumDevs* minus one. An application should not assume that these line devices are capable of anything beyond what is specified by the Basic Telephony subset without first querying their device capabilities using **lineGetDevCaps** and **lineGetAddressCaps**.

Applications should not invoke **lineInitialize** without subsequently opening a line (at least for monitoring). If the application is not monitoring and not using any devices, it should call **lineShutdown** so that memory resources allocated by TAPI.DLL can be released if unneeded, and TAPI.DLL itself can be unloaded from memory while not needed.

Another reason for performing a **lineShutdown** is that if a user changes the device configuration (adds or removes a line or phone), there is no way for TAPI to notify an application that has a line or phone handle open at the time. Once a reconfiguration has taken place, causing a LINEDEVSTATE_REINIT message to be sent, no applications can open a device until all applications have performed a **lineShutdown**. If any service provider fails to initialize properly, this function fails and returns the error indicated by the service provider.

On all TAPI platforms, **lineInitialize** is equivalent to **lineInitializeEx()** using the LINEINITIALIZEEXOPTION_USEHIDDENWINDOW option.

lineInitializeEx Overview

The **lineInitializeEx** function initializes the application's use of TAPI for subsequent use of the line abstraction. It registers the application's specified notification mechanism and returns the number of line devices available to the application. A line device is any device that provides an implementation for the line-prefixed functions in the Telephony API.

```
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 application's usage handle for TAPI.

hInstance

The instance handle of the client application or DLL. The application or DLL may pass NULL for this parameter, in which case TAPI will use 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 (for more information see **lineCallbackFunc**). This parameter is 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 displayable ASCII characters. If this parameter is not NULL, it contains an application-supplied name of the application. This name is provided in the **LINECALLINFO** structure to indicate, in a user-friendly way, which application originated, or originally accepted or answered the call. This information can be useful for call logging purposes. If *lpszFriendlyAppName* is NULL, the application's module filename is 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 is filled with the number of line devices 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 it is designed to support (for example, the same value it would pass into *dwAPIHighVersion* parameter of **lineNegotiateAPIVersion**). Artificially high values must not be used; the value must be accurately set (for this release, to 0x00020000). TAPI will translate any newer messages or structures into values or formats supported by the application's version. Upon successful completion of this request, this location is filled with the highest API version supported by

TAPI (for this release, 0x00020000), thereby allowing the application to detect and adapt to having been installed on a system with an older version of TAPI.

lpLineInitializeExParams

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALAPPNAME, LINEERR_OPERATIONFAILED, LINEERR_INIFILECORRUPT, LINEERR_INVALPOINTER, LINEERR_REINIT, LINEERR_NOMEM, LINEERR_INVALPARAM.

Remarks

Applications must select one of three mechanisms by which TAPI notifies the application of telephony events: Hidden Window, Event Handle, or Completion Port.

The Hidden Window mechanism is selected by specifying LINEINITIALIZEEXOPTION_USEHIDDENWINDOW in the **dwOptions** field in the **LINEINITIALIZEEXPARAMS** structure. In this mechanism (which is the only mechanism available to TAPI 1.x applications), TAPI creates a window in the context of the application during the **lineInitializeEx** function, and subclasses the window so that all messages posted to it are handled by a WNDPROC in TAPI itself. When TAPI has a message to deliver to the application, TAPI posts a message to the hidden window. When the message is received (which can happen only when the application calls the Windows **GetMessage** API), Windows switches the process context to that of the application and invokes the WNDPROC in TAPI. TAPI then delivers the message to the application by calling the **LineCallbackProc**, a pointer to which the application provided as a parameter in its call to **lineInitializeEx** (or **lineInitialize**, for TAPI 1.3 and 1.4 applications). This mechanism requires the application to have a message queue (which is not desirable for service processes) and to service that queue regularly to avoid delaying processing of telephony events. The hidden window is destroyed by TAPI during the **lineShutdown** function.

The Event Handle mechanism is selected by specifying LINEINITIALIZEEXOPTION_USEEVENT in the **dwOptions** field in the **LINEINITIALIZEEXPARAMS** structure. In this mechanism, TAPI creates an event object on behalf of the application, and returns a handle to the object in the **hEvent** field in **LINEINITIALIZEEXPARAMS**. The application must not manipulate this event in any manner (for example, must not call **SetEvent**, **ResetEvent**, **CloseHandle**, and so on) or undefined behavior will result; the application may only wait on this event using functions such as **WaitForSingleObject** or **MsgWaitForMultipleObjects**. TAPI will signal this event whenever a telephony event notification is pending for the application; the application must call **lineGetMessage** to fetch the contents of the message. The event is reset by TAPI when no events are pending. The event handle is closed and the event object destroyed by TAPI during the **lineShutdown** function. The application is not required to wait on the event handle that is created; the application could choose instead to call **lineGetMessage** and have it block waiting for a message to be queued.

The Completion Port mechanism is selected by specifying LINEINITIALIZEEXOPTION_USECOMPLETION PORT in the **dwOptions** field in the **LINEINITIALIZEEXPARAMS** structure. In this mechanism, whenever a telephony event needs to be sent to the application, TAPI will send it to the application using **PostQueuedCompletionStatus** to the completion port that the application specified in the **hCompletionPort** field in **LINEINITIALIZEEXPARAMS**, tagged with the completion key that the application specified in the **dwCompletionKey** field in **LINEINITIALIZEEXPARAMS**. The application must have previously created the completion port using **CreateIoCompletionPort**. The application retrieves events using

GetQueuedCompletionStatus. Upon return from **GetQueuedCompletionStatus**, the application will have the specified **dwCompletionKey** written to the DWORD pointed to by the *lpCompletionKey* parameter, and a pointer to a **LINEMESSAGE** structure returned to the location pointed to by *lpOverlapped*. After the application has processed the event, it is the application's responsibility to call **LocalFree** to release the memory used to contain the **LINEMESSAGE** structure. Because the application created the completion port (thereby allowing it to be shared for other purposes), the application must close it; the application must not close the completion port until after calling **lineShutdown**.

When a multi-threaded application is using the Event Handle mechanism and more than one thread is waiting on the handle, or the Completion Port notification mechanism and more than one thread is waiting on the port, it is possible for telephony events to be processed out of sequence. This is not due to the sequence of delivery of events from TAPI, but would be caused by the time slicing of threads or the execution of threads on separate processors.

If LINEERR_REINIT is returned and TAPI reinitialization has been requested, for example as a result of adding or removing a Telephony service provider, then **lineInitializeEx** requests are rejected with this error until the last application shuts down its usage of the API (using **lineShutdown**), at which time the new configuration becomes effective and applications are once again permitted to call **lineInitializeEx**.

If the LINEERR_INVALPARAM error value is returned, the specified *hInstance* parameter is invalid.

The application can refer to individual line devices by using line device IDs that range from zero to *dwNumDevs* minus one. An application should not assume that these line devices are capable of any particular TAPI function without first querying their device capabilities by **lineGetDevCaps** and **lineGetAddressCaps**.

See Also

lineCallbackFunc, **LINECALLINFO**, **lineGetAddressCaps**, **lineGetDevCaps**, **lineGetMessage**, **lineInitialize**, **LINEINITIALIZEEXPARAMS**, **LINEMESSAGE**, **lineNegotiateAPIVersion**, **lineShutdown**

lineCallbackFunc

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

```
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 associated with the callback. The nature of this handle (line handle or call handle) can be determined by the context provided by *dwMsg*. 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 passed back to the application in the callback. This DWORD is not interpreted by TAPI.

dwParam1

A parameter for the message.

dwParam2

A parameter for the message.

dwParam3

A parameter for the message.

Remarks

For information about parameter values passed to this function, see Line Device Messages.

All callbacks occur in the application's context. The callback function must reside in a DLL or application module.

lineMakeCall Overview

Overview

Overview

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

```
LONG lineMakeCall(

  HLINE hLine,
  LPHCALL lphCall,
  LPCSTR lpszDestAddress,
  DWORD dwCountryCode,
  LPLINECALLPARAMS const lpCallParams
);
```

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 LINE_REPLY message is received by the application indicating that the **lineMakeCall** function successfully completed. Use this handle to identify the call when invoking other telephony operations on the call. The application will initially be the sole owner of this call. This handle is void if the function returns an error (synchronously or asynchronously by the reply message).

lpszDestAddress

A pointer to the destination address. This follows the standard dialable number format. This pointer can be NULL for non-dialed addresses (as with a hot phone) or when all dialing will be performed using **lineDial**. In the latter case, **lineMakeCall** allocates an available call appearance that would typically remain in the *dialtone* state until dialing begins. Service providers that have inverse multiplexing capabilities may allow an application to specify multiple addresses at once.

dwCountryCode

The country code of the called party. If a value of zero is specified, a default is used by the implementation.

lpCallParams

A pointer to a **LINECALLPARAMS** structure. This structure allows the application to specify how it wants the call to be set up. If NULL is specified, a default 3.1 kHz voice call is established and an arbitrary origination address on the line is selected. This structure allows the application to select elements such as the call's bearer mode, data rate, expected media mode, origination address, blocking of caller ID information, and dialing parameters.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_ADDRESSBLOCKED, LINEERR_INVALLINEHANDLE, LINEERR_BEARERMODEUNAVAIL, LINEERR_INVALLINESTATE, LINEERR_CALLUNAVAIL, LINEERR_INVALMEDIAMODE, LINEERR_DIALBILLING, LINEERR_INVALPARAM, LINEERR_DIALDIALTONE, LINEERR_INVALPOINTER, LINEERR_DIALPROMPT, LINEERR_INVALRATE, LINEERR_DIALQUIET, LINEERR_NOMEM, LINEERR_INUSE, LINEERR_OPERATIONFAILED, LINEERR_INVALADDRESS, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALADDRESSID, LINEERR_RATEUNAVAIL, LINEERR_INVALADDRESSMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALBEARERMODE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALCALLPARAMS, LINEERR_UNINITIALIZED, LINEERR_INVALCOUNTRYCODE, LINEERR_USERUSERINFOTOOBIG.

Remarks

If LINEERR_INVALLINESTATE is returned, the line is currently not in a state in which this operation can be performed. A list of currently valid operations can be found in the **dwLineFeatures** field (of the type LINEFEATURE_) in the **LINEDEVSTATUS** structure. Calling **lineGetLineDevStatus** updates the information in **LINEDEVSTATUS**. If LINEERR_DIALBILLING, LINEERR_DIALQUIET, LINEERR_DIALDIALTONE, or LINEERR_DIALPROMPT is returned, none of the actions otherwise performed by **lineMakeCall** have occurred; for example, none of the dialable address prior to the offending character has been dialed, no hookswitch state has changed, and so on.

After dialing has completed, several LINE_CALLSTATE messages are usually sent to the application to notify it about the progress of the call. No generally valid sequence of call-state transitions is specified, as no single fixed sequence of transitions can be guaranteed in practice. A typical sequence may cause a call to transition from *dialtone*, *dialing*, *proceeding*, *ringback*, to *connected*. With non-dialed lines, the call may typically transition directly to *connected* state.

An application has the option to specify an originating address on the specified line device. A service provider that models all stations on a switch as addresses on a single line device allows the application to originate calls from any of these stations using **lineMakeCall**.

The call parameters allow the application to make non-voice calls or request special call setup options that are not available by default.

An application can partially dial using **lineMakeCall** and continue dialing using **lineDial**. To abandon a call attempt, use **lineDrop**.

After **lineMakeCall** returns a success reply message to the application, a LINE_CALLSTATE message is sent to the application to indicate the current state of the call. This state will not necessarily be LINECALLSTATE_DIALTONE.

See Also

LINE_CALLSTATE, LINE_REPLY, **LINECALLPARAMS**, **LINEDEVSTATUS**, **lineDial**, **lineDrop**, **lineGetLineDevStatus**

lineMonitorDigits Overview

Overview

Overview

The **lineMonitorDigits** function enables and disables the unbuffered detection of digits received on the call. Each time a digit of the specified digit mode is detected, a message is sent to the application indicating which digit has been detected.

```
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, digit monitoring is canceled. This parameter can have multiple flags set, and uses the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE

Detect digits as audible clicks that are the result of rotary pulse sequences. Valid digits for pulse are '0' through '9'.

LINEDIGITMODE_DTMF

Detect digits as DTMF tones. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

LINEDIGITMODE_DTMFEND

Detect and provide application notification of DTMF down edges. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_INVALDIGITMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

This function is considered successful if digit monitoring has been correctly initiated; not when digit monitoring has terminated. Digit monitoring remains in effect until it is explicitly disabled by calling **lineMonitorDigits** with *dwDigitModes* set to zero, until the call transitions to idle, or when the application deallocates its call handle for the call. Although this function can be invoked in any call state, digits are usually detected only while the call is in the *connected* state.

Each time a digit is detected, a LINE_MONITORDIGITS message is sent to the application passing the

detected digit as a parameter.

An application can use **lineMonitorDigits** to enable or disable unbuffered digit detection. It can use **lineGatherDigits** for buffered digit detection. After buffered digit gathering is complete, a LINE_GATHERDIGITS message is sent to the application. Both buffered and unbuffered digit detection can be enabled on the same call simultaneously.

Monitoring of digits on a conference call applies only to the *hConfCall*, not to the individual participating calls.

See Also

LINE_GATHERDIGITS, LINE_MONITORDIGITS, **lineGatherDigits**

lineMonitorMedia Overview

Overview

Overview

The **lineMonitorMedia** function enables and disables the detection of media modes on the specified call. When a media mode is detected, a message is sent to the application.

```
LONG lineMonitorMedia(

  HCALL hCall,
  DWORD dwMediaModes
);
```

Parameters

hCall

A handle to the call. The call state of *hCall* can be any state except *idle*.

dwMediaModes

The media modes to be monitored. A value of zero for the *dwMediaModes* parameter cancels all media mode detection. This parameter can have multiple flags set. This parameter uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_INTERACTIVEVOICE

The application wants to handle calls of the interactive voice media type (it manages live voice calls).

LINEMEDIAMODE_AUTOMATEDVOICE

Voice energy is present on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

The application wants to handle calls with the data modem media mode.

LINEMEDIAMODE_G3FAX

The application wants to handle calls of the group 3 fax media type.

LINEMEDIAMODE_TDD

The application wants to handle calls with the TDD (Telephony Devices for the Deaf) media mode.

LINEMEDIAMODE_G4FAX

The application wants to handle calls of the group 4 fax media type.

LINEMEDIAMODE_DIGITALDATA

The application wants to handle calls of the digital data media type

LINEMEDIAMODE_TELETEX

The application wants to handle calls with the teletex media mode.

LINEMEDIAMODE_VIDEOTEX

The application wants to handle calls with the videotex media mode.

LINEMEDIAMODE_TELEX

The application wants to handle calls with the telex media mode.

LINEMEDIAMODE_MIXED

The application wants to handle calls with the ISDN mixed media mode.

LINEMEDIAMODE_ADSI

The application wants to handle calls with the ADSI (Analog Display Services Interface) media mode.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_INVALMEDIAMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

The media modes specified with **lineOpen** relate only to enabling the detection of these media modes by the service provider for the purpose of handing off new incoming calls to the proper application. They do not impact any of the media-mode notification messages that are expected because of a previous invocation of **lineMonitorMedia**.

This function is considered successful if media-mode monitoring has been correctly initiated, not when media mode monitoring has terminated. Media monitoring for a given media mode remains in effect until it is explicitly disabled by calling **lineMonitorMedia** with a *dwMediaModes* parameter set to zero, until the call transitions to *idle*, or when the application deallocates its call handle for the call. The **lineMonitorMedia** function is primarily an event reporting mechanism. The media mode of call, as indicated in **LINECALLINFO**, is not affected by the service provider's detection of the media mode. Only the controlling application can change a call's media mode.

Default media monitoring performed by the service provider corresponds to the union of all media modes specified on **lineOpen**.

Although this function can be invoked in any call state, a call's media mode can typically only be detected while the call is in certain call states. These states may be device specific. For example, in ISDN, a message may indicate the media mode of the media stream before the media stream exists. Similarly, distinctive ringing or the called ID information about the call can be used to identify the media mode of a call. Otherwise, the call may have to be answered (call in the *connected* state) to allow a service provider to determine the call's media mode by filtering the media stream. Because filtering a call's media stream implies a computational overhead, applications should disable media monitoring when not required. By default, media monitoring is enabled for newly inbound calls, because a call's media mode selects the application that should handle the call.

An outbound application that deals with voice media modes may want to monitor the call for silence (a tone) to distinguish who or what is at the called end of a call. For example, a person at home may answer calls with just a short "hello." A person in the office may provide a longer greeting, indicating name and company name. An answering machine may typically have an even longer greeting.

Because media-mode detection enabled by **lineMonitorMedia** is implemented as a read-only operation of the call's media stream, it is not disruptive.

Monitoring of media on a conference call applies only to the *hConfCall*, not to the individual participating

calls

See Also

LINECALLINFO, lineOpen

lineMonitorTones Overview

Overview

Overview

The **lineMonitorTones** function enables and disables the detection of inband tones on the call. Each time a specified tone is detected, a message is sent to the application.

```
LONG lineMonitorTones(

  HCALL hCall,
  LPLINEMONITORTONE const lpToneList,
  DWORD dwNumEntries
);
```

Parameters

hCall

A handle to the call on whose voice channel tones are to be monitored. 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 report a tone detection. Tone monitoring in progress is canceled or changed by calling this operation with either NULL for *lpToneList* or with another tone list.

dwNumEntries

The number of entries in *lpToneList*. This parameter is ignored if *lpToneList* is NULL.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_INVALCALLSTATE, LINEERR_INVALPOINTER, LINEERR_INVALTONE, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

Remarks

This function is successful if tone monitoring has been correctly initiated, not when tone monitoring has terminated. Tone monitoring will remain in effect until it is explicitly disabled by calling **lineMonitorTones** with another tone list (or NULL), until the call transitions to *idle*, or when the application deallocates its call handle for the call.

Although this function can be invoked in any call state, tones can typically only be detected while the call is in the *connected* state. Tone detection typically requires computational resources. Depending on the service provider and other activities that compete for such resources, the number of tones that can be detected may vary over time. Also, an equivalent amount of resources may be consumed for monitoring a single triple frequency tone versus three single frequency tones. If resources are overcommitted, the LINEERR_RESOURCEUNAVAIL error is returned.

Note that **lineMonitorTones** is also used to detect silence. Silence is specified as a tone with all zero frequencies.

Monitoring of tones on a conference call applies only to the *hConfCall*, not to the individual participating calls

If the LINEERR_INVALPOINTER error value is returned, the specified *lpToneList* parameter is invalid or the value specified by the *dwNumEntries* parameter is too large.

See Also

LINEMONITORTONE

lineNegotiateAPIVersion Overview

Overview

Overview

The **lineNegotiateAPIVersion** function allows an application to negotiate an API version to use.

```
LONG lineNegotiateAPIVersion(

  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPILowVersion,
  DWORD dwAPIHighVersion,
  LPDWORD lpdwAPIVersion,
  LPLINEEXTENSIONID lpExtensionID
);
```

Parameters

hLineApp

The handle to the application's registration with TAPI.

dwDeviceID

The line device to be queried.

dwAPILowVersion

The least recent API version the application is compliant with. The high-order word is the major version number; the low-order word is the minor version number.

dwAPIHighVersion

The most recent API version the application is compliant with. The high-order word is the major version number; the low-order word is the minor version number.

lpdwAPIVersion

A pointer to a DWORD-sized location that contains the API version number that was negotiated. If negotiation is successful, this number will be 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, then, upon a successful negotiation, this structure is filled with the extension ID 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.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_OPERATIONFAILED, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_OPERATIONUNAVAIL, LINEERR_NODEVICE.

Remarks

Use **lineInitializeEx** to determine the number of line devices present in the system. The device ID specified by *dwDeviceID* varies from zero to one less than the number of line devices present.

The **lineNegotiateAPIVersion** function is used to negotiate the API version number to use. It also retrieves the extension ID supported by the line device, and returns zeros if no extensions are supported. If the application wants to use the extensions defined by the returned extension ID, it must call **lineNegotiateExtVersion** to negotiate the extension version to use.

The API version number negotiated is that under which TAPI can operate. If version ranges do not overlap, the application and API or service provider versions are incompatible and an error is returned.

See Also

LINEEXTENSIONID, **lineInitializeEx**, **lineNegotiateExtVersion**

lineNegotiateExtVersion Overview

Overview

Overview

The **lineNegotiateExtVersion** function allows an application to negotiate an extension version to use with the specified line device. This operation need not be called if the application does not support extensions.

```
LONG lineNegotiateExtVersion(

  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  DWORD dwExtLowVersion,
  DWORD dwExtHighVersion,
  LPDWORD lpdwExtVersion
);
```

Parameters

hLineApp

The handle to the application's registration with TAPI.

dwDeviceID

The line device to be queried.

dwAPIVersion

The API version number that was negotiated for the specified line device using **lineNegotiateAPIVersion**.

dwExtLowVersion

The least recent extension version of the extension ID returned by **lineNegotiateAPIVersion** that the application is compliant with. The high-order word is the major version number; the low-order word is the minor version number.

dwExtHighVersion

The most recent extension version of the extension ID returned by **lineNegotiateAPIVersion** that the application is compliant with. The high-order word is the major version number; the low-order word is the minor version number.

lpdwExtVersion

A pointer to a DWORD-sized location that contains the extension version number that was negotiated. If negotiation is successful, this number will be in the range between *dwExtLowVersion* and *dwExtHighVersion*.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_NODRIVER, LINEERR_INCOMPATIBLEEXTVERSION, LINEERR_OPERATIONFAILED, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NODEVICE, LINEERR_OPERATIONUNAVAIL.

Remarks

Use **lineInitializeEx** to determine the number of line devices present in the system. The device ID specified by *dwDeviceID* varies from zero to one less than the number of line devices present.

The **lineNegotiateAPIVersion** function negotiates the API version number to use. It also retrieves the extension ID supported by the line device, which is zeros if no extensions are provided. Version numbers should be incremented by one for each release. Leaving gaps in release version numbering may cause unexpected results.

If the application wants to use the extensions defined by the returned extension ID, it must call **lineNegotiateExtVersion** to negotiate the extension version to use.

The extension version number negotiated is that under which the application and service provider must both operate. If version ranges do not overlap, the application and service provider versions are incompatible and an error is returned.

See Also

lineInitializeEx, **lineNegotiateAPIVersion**

lineOpen Overview

Overview

Overview

The **lineOpen** function opens the line device specified by its device ID and returns a line handle for the corresponding opened line device. This line handle is used in subsequent operations on the line device.

```
LONG lineOpen(
  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  LPHLINE lphLine,
  DWORD dwAPIVersion,
  DWORD dwExtVersion,
  DWORD dwCallbackInstance,
  DWORD dwPrivileges,
  DWORD dwMediaModes,
  LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hLineApp

A handle to the application's registration with TAPI.

dwDeviceID

Identifies the line device to be opened. It can either be a valid device ID or the value:

LINEMAPPER

This value is used to open a line device in the system that supports the properties specified in *lpCallParams*. The application can use **lineGetID** to determine the ID of the line device that was opened.

lphLine

A pointer to an HLINE handle, which is then loaded with the handle representing the opened line device. Use this handle to identify the device when invoking other functions on the open line device.

dwAPIVersion

The API version number under which the application and Telephony API have agreed to operate. This number is obtained with **lineNegotiateAPIVersion**.

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. This number is obtained with **lineNegotiateExtVersion**.

dwCallbackInstance

User-instance data passed back to the application with each message associated with this line or addresses or calls on this line. This parameter is not interpreted by the Telephony API.

dwPrivileges

The privilege the application wants for the calls it is notified for. This parameter can be a combination

of the LINECALLPRIVILEGE_ constants. For applications using API version 0x00020000 or greater, values for this parameter can also be combined with the LINEOPENOPTION_ constants:

LINECALLPRIVILEGE_NONE

The application wants to make only outbound calls.

LINECALLPRIVILEGE_MONITOR

The application only wants to monitor inbound and outbound calls.

LINECALLPRIVILEGE_OWNER

The application wants to own inbound calls of the types specified in *dwMediaModes*.

LINECALLPRIVILEGE_MONITOR + LINECALLPRIVILEGE_OWNER

The application wants to own inbound calls of the types specified in *dwMediaModes*, but if it cannot be an owner of a call, it wants to be a monitor.

LINEOPENOPTION_SINGLEADDRESS

The application is interested only in new calls that appear on the address specified by the **dwAddressID** field in the **LINECALLPARAMS** structure pointed to by the *lpCallParams* parameter (which must be specified). If LINEOPENOPTION_SINGLEADDRESS is specified but either *lpCallParams* is invalid or the included **dwAddressID** does not exist on the line, the open fails with LINERR_INVALADDRESSID.

In addition to setting the **dwAddressID** member of the **LINECALLPARAMS** structure to the desired address, the application must also set **dwAddressMode** in **LINECALLPARAMS** to LINEADDRESSMODE_ADDRESSID.

The LINEOPENOPTION_SINGLEADDRESS option affects only TAPI's assignment of initial call *ownership* of calls created by the service provider using a LINE_NEWCALL message. An application that opens the line with LINECALLPRIVILEGE_MONITOR will continue to receive monitoring handles to all calls created on the line. Furthermore, the application is not restricted in any way from making calls or performing other operations that affect other addresses on the line opened.

LINEOPENOPTION_PROXY

The application is willing to handle certain types of requests from other applications that have the line open, as an adjunct to the service provider. Requests will be delivered to the application using LINE_PROXYREQUEST messages; the application responds to them by calling **lineProxyResponse**, and can also generate TAPI messages to other applications having the line open by calling **lineProxyMessage**.

When this option is specified, the application must also specify which specific proxy requests it is prepared to handle. It does so by passing, in the *lpCallParams* parameter, a pointer to a **LINECALLPARAMS** structure in which the **dwDevSpecificSize** and **dwDevSpecificOffset** members have been set to delimit an array of DWORDs. Each element of this array shall contain one of the LINEPROXYREQUEST_ constants. For example, a proxy handler application that supports all five of the Agent-related functions would pass in an array of five DWORDs (**dwDevSpecificSize** would be 20 decimal) containing the five defined LINEPROXYREQUEST_ values.

The proxy request handler application can run on any machine that has authorization to control the line device. However, requests will always be routed through the server on which the service provider is executing that actually controls the line device. Thus, it is most efficient if the application handling proxy requests (such as ACD agent control) executes directly on the server along with the service provider.

Subsequent attempts, by the same application or other applications, to open the line device and register to handle the same proxy requests as an application that is already registered fail with LINEERR_NOTREGISTERED.

To stop handling requests on the line, the application simply calls **lineClose**.

Other flag combinations return the LINEERR_INVALPRIVSELECT error.

dwMediaModes

The media mode or modes of interest to the application. This parameter is used to register the application as a potential target for inbound call and call handoff for the specified media mode. This parameter is meaningful only if the bit LINECALLPRIVILEGE_OWNER in *dwPrivileges* is set (and ignored if it is not). This parameter uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN

The application wants to handle calls of unknown media type (unclassified calls).

LINEMEDIAMODE_INTERACTIVEVOICE

The application wants to handle calls of the interactive voice media type. That is, it manages voice calls with the human user on this end of the call.

LINEMEDIAMODE_AUTOMATEDVOICE

Voice energy is present on the call. The voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

The application wants to handle calls with the data-modem media mode.

LINEMEDIAMODE_G3FAX

The application wants to handle calls of the group 3 fax media type.

LINEMEDIAMODE_TDD

The application wants to handle calls with the TDD (Telephony Devices for the Deaf) media mode.

LINEMEDIAMODE_G4FAX

The application wants to handle calls of the group 4 fax media type.

LINEMEDIAMODE_DIGITALDATA

The application wants to handle calls of the digital-data media type.

LINEMEDIAMODE_TELETEX

The application wants to handle calls with the teletex media mode.

LINEMEDIAMODE_VIDEOTEX

The application wants to handle calls with the videotex media mode.

LINEMEDIAMODE_TELEX

The application wants to handle calls with the telex media mode.

LINEMEDIAMODE_MIXED

The application wants to handle calls with the ISDN mixed media mode.

LINEMEDIAMODE_ADSI

The application wants to handle calls with the ADSI (Analog Display Services Interface) media mode.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

lpCallParams

A pointer to a structure of type **LINECALLPARAMS**. This pointer is only used if LINEMAPPER is used; otherwise *lpCallParams* is ignored. It describes the call parameter that the line device should be able to provide.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_ALLOCATED, LINEERR_LINEMAPPERFAILED, LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_NOMEM, LINEERR_INCOMPATIBLEEXTVERSION, LINEERR_OPERATIONFAILED, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALMEDIAMODE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_INVALPRIVSELECT, LINEERR_REINIT, LINEERR_NODEVICE, LINEERR_OPERATIONUNAVAIL.

Remarks

If LINEERR_ALLOCATED is returned, the line cannot be opened due to a "persistent" condition, such as that of a serial port being exclusively opened by another process. If LINEERR_RESOURCEUNAVAIL is returned, the line cannot be opened due to a dynamic resource overcommitment such as in DSP processor cycles or memory. This overcommitment may be transitory, caused by monitoring of media mode or tones, and changes in these activities by other applications may make it possible to reopen the line within a short time period. If LINEERR_REINIT is returned and TAPI reinitialization has been requested, for example as a result of adding or removing a Telephony service provider, then **lineOpen** requests are rejected with this error until the last application shuts down its usage of the API (using **lineShutdown**), at which time the new configuration becomes effective and applications are once again permitted to call **lineInitializeEx**.

Opening a line always entitles the application to make calls on any address available on the line. The ability of the application to deal with inbound calls or to be the target of call handoffs on the line is determined by the *dwMediaModes* parameter. The **lineOpen** function registers the application as having an interest in monitoring calls or receiving ownership of calls that are of the specified media modes. If the application just wants to monitor calls, then it can specify LINECALLPRIVILEGE_MONITOR. If the application just wants to make outbound calls, it can specify LINECALLPRIVILEGE_NONE. If the application is willing to control unclassified calls (calls of unknown media mode), it can specify LINECALLPRIVILEGE_OWNER and LINEMEDIAMODE_UNKNOWN. Otherwise, the application should specify the media mode it is interested in handling.

The media modes specified with **lineOpen** add to the default value for the provider's media mode monitoring for initial inbound call type determination. The **lineMonitorMedia** function modifies the mask that controls LINE_MONITORMEDIA messages. If a line device is opened with owner privilege and an extension media mode is not registered, then the error LINEERR_INVALMEDIAMODE is returned.

An application that has successfully opened a line device can always initiate calls using **lineMakeCall**, **lineUnpark**, **linePickup**, **lineSetupConference** (with a NULL *hCall*), as well as use **lineForward** (assuming that doing so is allowed by the device capabilities, line state, and so on).

A single application may specify multiple flags simultaneously to handle multiple media modes. Conflicts may arise if multiple applications open the same line device for the same media mode. These conflicts are resolved by a priority scheme in which the user assigns relative priorities to the applications. Only the highest priority application for a given media mode will ever receive ownership (unsolicited) of a call of that media mode. Ownership can be received when an inbound call first arrives or when a call is handed off.

Any application (including any lower priority application) can always acquire ownership with **lineGetNewCalls** or **lineGetConfRelatedCalls**. If an application opens a line for monitoring at a time that calls exist on the line, LINE_CALLSTATE messages for those existing calls are not automatically generated to the new monitoring application. The application can query the number of current calls on the line to determine how many calls exist, and, if it wants, it can call **lineGetNewCalls** to obtain handles to these calls.

An application that handles automated voice should also select the interactive voice open mode and be assigned the lowest priority for interactive voice. The reason for this is that service providers will report all voice media modes as interactive voice. If media mode determination is not performed by the application for the UNKNOWN media type, and no interactive voice application has opened the line device, voice calls would be unable to reach the automated voice application, but be dropped instead.

The same application, or different instantiations of the same application, may open the same line multiple times with the same or different parameters.

When an application opens a line device it must specify the negotiated API version and, if it wants to use the line's extensions, it should specify the line's device-specific extension version. These version numbers should have been obtained with **lineNegotiateAPIVersion** and **lineNegotiateExtVersion**. Version numbering allows the mixing and matching of different application versions with different API versions and service provider versions.

LINEMAPPER allows an application to select a line indirectly—by means of the services it wants from it. When opening a line device using LINEMAPPER, the following is true: All fields from beginning of the **LINECALLPARAMS** data structure through **dwAddressMode** are relevant. If **dwAddressMode** is LINEADDRESSMODE_ADDRESSID it means that any address on the line is acceptable, otherwise if **dwAddressMode** is LINEADDRESSMODE_DIALABLEADDR, indicating that a specific originating address (phone number) is searched for, or if it is a provider-specific extension, then **dwOrigAddressSize/Offset** and the portion of the variable part they refer to are also relevant. If **dwAddressMode** is a provider-specific extension additional information may be contained in the **dwDeviceSpecific** variably sized field.

See Also

LINE_CALLSTATE, LINE_MONITORMEDIA, LINE_PROXYREQUEST, **LINECALLPARAMS**, **lineClose**, **lineForward**, **lineGetConfRelatedCalls**, **lineGetNewCalls**, **lineInitializeEx**, **lineMakeCall**, **lineMonitorMedia**, **lineNegotiateAPIVersion**, **lineNegotiateExtVersion**, **linePickup**, **lineProxyMessage**, **lineProxyResponse**, **lineSetupConference**, **lineShutdown**, **lineUnpark**

linePark Overview

Overview

Overview

The **linePark** function parks the specified call according to the specified park mode.

```
LONG linePark(
  HCALL hCall,
  DWORD dwParkMode,
  LPCSTR lpszDirAddress,
  LPVARSTRING lpNonDirAddress
);
```

Parameters

hCall

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

dwParkMode

The park mode with which the call is to be parked. This parameter can have only a single flag set, and it uses the following LINEPARKMODE_ constants:

LINEPARKMODE_DIRECTED

The application specifies at which address the call is to be parked in *lpszDirAddress*.

LINEPARKMODE_NONDIRECTED

This operation reports to the application where the call has been parked in *lpNonDirAddress*.

lpszDirAddress

A pointer to a NULL-terminated string that indicates the address where the call is to be parked when using directed park. The address is in dialable number format. This parameter is ignored for nondirected park.

lpNonDirAddress

A pointer to a structure of type **VARSTRING**. For nondirected park, the address where the call is parked is returned in this structure. This parameter is ignored for directed park. Within the **VARSTRING** structure, **dwStringFormat** must be set to STRINGFORMAT_ASCII (an ASCII string buffer containing a NULL-terminated string), and the terminating NULL must be accounted for in the **dwStringSize**. Prior to calling **linePark**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESS, LINEERR_NOTOWNER, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED,

LINEERR_INVALPARKMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

With directed park, the application determines the address at which it wants to park the call. With nondirected park, the switch determines the address and provides this to the application. In either case, a parked call can be unparked by specifying this address.

The parked call typically enters the *idle* state after it has been successfully parked and the application should then deallocate its handle to the call. If the application performs a **lineUnpark** on the parked call, a new call handle will be created for the unparked call even if the application has not deallocated its old call handle.

Some switches may remind the user after a call has been parked for some long amount of time. The application will see an *offering* call with a call reason set to *reminder*.

On a nondirected park, if the **dwTotalSize** member in the **VARSTRING** structure does not specify a sufficient amount of memory to receive the park address, the corresponding reply message returns a LINEERR_STRUCTURETOOSMALL error value. In such cases, there is no way to retrieve the complete park address. Note that when a LINEERR_STRUCTURETOOSMALL error value is returned, the **dwNeededSize** field of the NonDirAddress structure does not contain a valid value. If a LINEERR_STRUCTURETOOSMALL error value is received from a nondirected linePark, then increase the size of the buffer and call **linePark** again until it returns either success or a different LINEERR_XXX result.

See Also

LINE_REPLY, **lineUnpark**, **VARSTRING**

linePickup Overview

Overview

Overview

The **linePickup** function picks up a call alerting at the specified destination address and returns a call handle for the picked-up call. If invoked with NULL for the *lpszDestAddress* parameter, a group pickup is performed. If required by the device, *lpszGroupID* specifies the group ID to which the alerting station belongs.

```
LONG linePickup(

  HLINE hLine,
  DWORD dwAddressID,
  LPHCALL lphCall,
  LPCSTR lpszDestAddress,
  LPCSTR lpszGroupID
);
```

Parameters

hLine

A handle to the open line device on which a call is to be picked up.

dwAddressID

The address on *hLine* at which the pickup is to be originated.

lphCall

A pointer to a memory location where the handle to the picked up call will be returned. The application will be the initial sole owner of the call.

lpszDestAddress

A pointer to a NULL-terminated character buffer that contains the address whose call is to be picked up. The address is in standard dialable address format.

lpszGroupID

A pointer to a NULL-terminated character buffer containing the group ID to which the alerting station belongs. This parameter is required on some switches to pick up calls outside of the current pickup group.

Note that *lpszGroupID* can be specified by itself with a NULL pointer for *lpszDestAddress*. Alternatively, *lpszGroupID* can be specified in addition to *lpszDestAddress*, if required by the device.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESS, LINEERR_NOMEM, LINEERR_INVALADDRESSID, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALGROUPID, LINEERR_OPERATIONFAILED, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED.

Remarks

When a call has been picked up successfully, the application is notified by the LINE_CALLSTATE message about call state changes. The **LINECALLINFO** structure supplies information about the call that was picked up. It will list the reason for the call as *pickup*. This structure is available using **lineGetCallInfo**.

If LINEADDRCAPFLAGS_PICKUPCALLWAIT is TRUE, **linePickup** can be used to pick up a call for which the user has audibly detected the call-waiting signal but for which the provider is unable to perform the detection. This gives the user a mechanism to "answer" a waiting call even though the service provider was unable to detect the call-waiting signal. Both *lpszDestAddress* and *lpszGroupID* pointer parameters must be NULL to pick up a call-waiting call. The **linePickup** function will create a new call handle for the waiting call and pass that handle to the user. *dwAddressID* will most often be zero (particularly in single-line residential cases).

Once **linePickup** has been used to pick up the second call, **lineSwapHold** can be used to toggle between them. The **lineDrop** function can be used to drop one (and toggle to the other), and so forth. If the user wants to drop the current call and pick up the second call, they should call **lineDrop** when they get the call-waiting beep, wait for the second call to ring, and then call **lineAnswer** on the new call handle. The LINEADDRFEATURE_PICKUP flag in the **dwAddressFeatures** field in **LINEADDRESSSTATUS** indicates when pickup is actually possible.

See Also

LINE_CALLSTATE, LINE_REPLY, **LINEADDRESSSTATUS**, **lineAnswer**, **LINECALLINFO**, **lineDrop**, **lineGetCallInfo**, **lineSwapHold**

linePrepareAddToConference Overview

Overview

Overview

The **linePrepareAddToConference** function prepares an existing conference call for the addition of another party.

```
LONG linePrepareAddToConference(

  HCALL hConfCall,
  LPHCALL lphConsultCall,
  LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hConfCall

A handle to a conference call. The application must be an owner of this call. The call state of *hConfCall* must be *connected*.

lphConsultCall

A pointer to an HCALL handle. This location is then loaded with a handle identifying the consultation call to be added. Initially, the application will be the sole owner of this call.

lpCallParams

A pointer to call parameters to be used when establishing the consultation call. This parameter may be set to NULL if no special call setup parameters are desired.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

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.

Remarks

If LINEERR_INVALLINESTATE is returned, the line is currently not in a state in which this operation can be performed. A list of currently valid operations can be found in the **dwLineFeatures** field (of the type LINEFEATURE_) in the **LINEDEVSTATUS** structure. (Calling **lineGetLineDevStatus** updates the information in **LINEDEVSTATUS**.)

A conference call handle can be obtained with **lineSetupConference** or with **lineCompleteTransfer** that is resolved as a three-way conference call. The function **linePrepareAddToConference** 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**.

The consultation call can be canceled using **lineDrop**. It may also be possible for an application to swap between the consultation call and the held conference call with **lineSwapHold**.

See Also

LINE_REPLY, **lineAddToConference**, **lineCompleteTransfer**, **LINEDEVSTATUS**, **lineDrop**, **lineGetLineDevStatus**, **lineSetupConference**, **lineSwapHold**

lineProxyMessage Overview

Overview

Overview

The **lineProxyMessage** function is used by a registered proxy request handler to generate TAPI messages related to its role. For example, an ACD agent handler can use this function to generate LINE_AGENTSTATUS messages that will be received by all applications that have the specified line open. TAPI suppresses generation of the message on the *hLine* specified in the function parameters.

```
LONG lineProxyMessage(

  HLINE hLine,
  HCALL hCall,
  DWORD dwMsg,
  DWORD dwParam1,
  DWORD dwParam2,
  DWORD dwParam3
);
```

Parameters

hLine

A handle to the open line device. This will be converted by TAPI into the correct *hLine* for each application that receives the message.

hCall

If the message is related to a specific call (which it is not, in the case of LINE_AGENTSTATUS), specifies the proxy handler's handle to that call; shall be set to NULL for messages not related to a specific call. This will be converted by TAPI into the correct *hCall* for each application that receives the message.

dwMsg

The TAPI message to be generated. This must be a message that is permitted to be generated by proxy handlers.

dwParam1

dwParam2

dwParam3

The parameters associated with the message to be sent.

Return Values

Returns zero if the function is successful or one of these negative error values:

LINEERR_INVALLINEHANDLE, LINEERR_INVALCALLHANDLE, LINEERR_INVALPARAM, LINEERR_NOMEM, LINEERR_NOTREGISTERED, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

See Also

LINE_AGENTSTATUS

lineProxyResponse Overview

Overview

Overview

The **lineProxyResponse** function indicates completion of a proxy request by a registered proxy handler such as an ACD agent handler on a server.

```
LONG lineProxyResponse(

  HLINE hLine,
  LPLINEPROXYREQUEST lpProxyRequest,
  DWORD dwResult
);
```

Parameters

hLine

A handle to the open line device.

lpProxyRequest

The pointer to the proxy request buffer that was given to the application by TAPI in a LINE_PROXYREQUEST message. In the case of proxy requests that return data to the client application, the proxy handler shall have filled in the necessary structure in this buffer before calling this function. The **dwNeededSize** and **dwUsedSize** fields of the structure to be returned must have been set appropriately.

dwResult

The function result that is to be returned to the calling application in a LINE_REPLY message (generated automatically by TAPI). Must be 0 or one of the negative error values defined for the function invoked.

Return Values

Returns zero if the function is successful or one of these negative error values:

LINEERR_INVALLINEHANDLE, LINEERR_INVALPARAM, LINEERR_INVALPOINTER, LINEERR_NOMEM, LINEERR_NOTREGISTERED, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

lineRedirect Overview

Overview

Overview

The **lineRedirect** function redirects the specified offering call to the specified destination address.

```
LONG lineRedirect(

  HCALL hCall,
  LPCSTR lpszDestAddress,
  DWORD dwCountryCode
);
```

Parameters

hCall

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

lpszDestAddress

A pointer to the destination address. This follows the standard dialable number format.

dwCountryCode

The country code of the party the call is redirected to. If a value of zero is specified, a default is used by the implementation.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESS, LINEERR_NOTOWNER, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_INVALCOUNTRYCODE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Remarks

Call redirection allows an application to deflect an offering call to another address without first answering the call. Call redirect differs from call forwarding in that call forwarding is performed by the switch without the involvement of the application; redirection can be done on a call-by-call basis by the application, for example, driven by caller ID information. It differs from call transfer in that transferring a call requires the call first be answered.

After a call has been successfully redirected, the call typically transitions to idle.

Besides redirecting an incoming call, an application may have the option to accept the call using **lineAccept**, reject the call using **lineDrop**, or answer the call using **lineAnswer**. The availability of these operations is dependent on device capabilities.

See Also

LINE_REPLY, **lineAccept**, **lineAnswer**, **lineDrop**

lineRegisterRequestRecipient Overview

Overview

Overview

The **lineRegisterRequestRecipient** function registers the invoking application as a recipient of requests for the specified request mode.

```
LONG 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 is ignored if *bEnable* is set to zero. This parameter is examined by TAPI only for registration, not for deregistration. The *dwRegistrationInstance* value used while deregistering need not match the *dwRegistrationInstance* used while registering for a request mode.

dwRequestMode

The type or types of request for which the application registers. One or both bits may be set. This parameter uses the following LINEREQUESTMODE_ constants:

LINEREQUESTMODE_MAKECALL

A **tapiRequestMakeCall** request.

bEnable

If TRUE, the application registers; if FALSE, the application deregisters for the specified request modes.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALAPPHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALREQUESTMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

A telephony-enabled application can request that a call be placed on its behalf by invoking **tapiRequestMakeCall**. Additionally, other applications can request that information be logged with a given call. The **tapiRequestMakeCall** requests are queued by TAPI, and the highest priority application that has registered to handle the request is sent a LINE_REQUEST message with an indication of the

mode of the request that is pending. This application is typically the user's call-control application.

Next, the call-control application that receives this message invokes **lineGetRequest**, specifying the request mode and a buffer that is large enough to hold the request. The call-control application then interprets and executes the request.

The recipient application is also automatically deregistered for all requests when it does a **lineShutdown**.

See Also

LINE_REQUEST, **lineGetRequest**, **lineShutdown**, **tapiRequestMakeCall**

lineReleaseUserUserInfo Overview

Overview

Overview

The **lineReleaseUserUserInfo** function informs the service provider that the application has processed the user-to-user information contained in the **LINECALLINFO** structure, and that subsequently received user-to-user information can now be written into that structure. The service provider will send a LINE_CALLINFO message indicating LINECALLINFOSTATE_USERUSERINFO when new information is available.

```
LONG lineReleaseUserUserInfo(

  HCALL hCall
);
```

Parameters

hCall

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

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_RESOURCEUNAVAIL, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED, LINEERR_OPERATIONUNAVAIL.

Remarks

The **lineReleaseUserUserInfo** function allows the application to control the flow of incoming user-user information on an ISDN connection. When a new, complete user-user information message is received, the service provider informs the application using a LINE_CALLINFO message (specifying LINECALLINFOSTATE_USERUSERINFO). Any number of applications may examine the information (using **lineGetCallInfo**), but the application owning the call controls when the information is released so that subsequent information can be reported. The service provider will not overwrite previous user-user information in LINECALLINFO with newer information until after **lineReleaseUserUserInfo** has been called. It is the responsibility of the service provider to buffer subsequently received user-user information until the previous information is released by the application owning the call.

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so. The function will work the same way for all applications.

See Also

LINE_CALLINFO, LINE_REPLY, **LINECALLINFO**, **lineGetCallInfo**

lineRemoveFromConference Overview

Overview

Overview

The **lineRemoveFromConference** function removes the specified call from the conference call to which it currently belongs. The remaining calls in the conference call are unaffected.

```
LONG lineRemoveFromConference(

  HCALL hCall
);
```

Parameters

hCall

A handle to the call to be removed from the conference. The application must be an owner of this call. Call state of *hCall* must be *conferenced*.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_RESOURCEUNAVAIL, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED.

Remarks

This operation removes a party that currently belongs to a conference call. After the call has been successfully removed, it may be possible to further manipulate it using its handle. The availability of this operation and its result are likely to be limited in many implementations. For example, in many implementations, only the most recently added party may be removed from a conference, and the removed call may be automatically dropped (becomes idle). Consult the line's device capabilities to determine the available effects of removing a call from a conference.

If the removal of a participant from a conference is supported, the participant call, after it is removed from the conference, will enter the call-state listed in the **dwRemoveFromConfState** field in **LINEADDRESSCAPS**.

See Also

LINE_REPLY, **LINEADDRESSCAPS**

lineRemoveProvider Overview

Overview

Overview

The **lineRemoveProvider** function removes an existing Telephony Service Provider from the Telephony system.

```
LONG lineRemoveProvider(

  DWORD dwPermanentProviderID,
  HWND hwndOwner
);
```

Parameters

dwPermanentProviderID

The permanent provider ID of the service provider to be removed.

hwndOwner

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

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INIFILECORRUPT, LINEERR_NOMEM, LINEERR_INVALPARAM, LINEERR_OPERATIONFAILED.

Remarks

If the call to **TSPI_providerRemove** is successful, and the telephony system is active at the time, TAPI calls **TSPI_lineShutdown** and/or **TSPI_phoneShutdown** on the service provider (depending on which device types are active). Any line or phone handles still held by applications on associated devices are forcibly closed with LINE_CLOSE or PHONE_CLOSE messages (it is preferable for service providers themselves to issue these messages as part of **TSPI_providerRemove**, after verification with the user). The devices previously under the control of that provider are then marked as "unavailable", so that any future attempts by applications to reference them by device ID result in LINEERR_NODRIVER.

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so. The function will work the same way for all applications.

See Also

LINE_CLOSE, PHONE_CLOSE

lineSecureCall Overview

Overview

Overview

The **lineSecureCall** function secures the call from any interruptions or interference that may affect the call's media stream.

```
LONG lineSecureCall(

  HCALL hCall
);
```

Parameters

hCall

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

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_RESOURCEUNAVAIL, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED.

Remarks

A call can be secured to avoid interference. For example, in an analog environment, call-waiting tones may destroy a fax or modem session on the original call. The **lineSecureCall** function allows an existing call to be secured. The **lineMakeCall** function provides the option to secure the call from the time of call setup. The securing of a call remains in effect for the duration of the call.

See Also

LINE_REPLY, **lineMakeCall**

lineSendUserUserInfo Overview

Overview

Overview

The **lineSendUserUserInfo** function sends user-to-user information to the remote party on the specified call.

```
LONG lineSendUserUserInfo(

  HCALL hCall,
  LPCSTR lpsUserUserInfo,
  DWORD dwSize
);
```

Parameters

hCall

A handle to the call on which to send user-to-user information. The application must be an owner of the call. The call state of *hCall* must be *connected*, *offering*, *accepted*, or *ringback*.

lpsUserUserInfo

A pointer to a string containing user-to-user information to be sent to the remote party. User-to-user information is only sent if supported by the underlying network (see **LINEDEVCAPS**). The protocol discriminator field for the user-user information, if required, should appear as the first byte of the buffer pointed to by *lpsUserUserInfo*, and must be accounted for in *dwSize*.

dwSize

The size in bytes of the user-to-user information in *lpsUserUserInfo*.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_USERUSERINFOTOOBIG, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED.

Remarks

This function can be used to send user-to-user information at any time during a connected call. If the size of the specified information to be sent is larger than what may fit into a single network message (as in ISDN), the service provider is responsible for dividing the information into a sequence of chained network messages (using "more data").

User-to-user information can also be sent as part of call accept, call reject, and call redirect, and when making calls. User-to-user information can also be received. The received information is available through the call's call-information record. Whenever user-to-user information arrives after call offering or prior to call disconnect, a LINE_CALLINFO message with a *UserUserInfo* parameter will notify the application that user-to-user information in the call-information record has changed. If multiple network messages are chained, the information is assembled by the service provider and a single message is sent to the

application.

See Also

LINE_CALLINFO, LINE_REPLY, **LINEDEVCAPS**

lineSetAgentActivity Overview

Overview

Overview

The **lineSetAgentActivity** function sets the agent activity code associated with a particular address.

```
LONG lineSetAgentActivity(

  HLINE hLine,
  DWORD dwAddressID,
  DWORD dwActivityID
);
```

Parameters

hLine

Handle of the line device.

dwAddressID

Identifier of the address for which the agent activity code is to be changed.

dwActivityID

The new agent activity. The meaning of all values of this parameter are specific to the application and call center server.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALADDRESSID, LINEERR_INVALADDRESSSTATE, LINEERR_INVALAGENTACTIVITY, LINEERR_INVALLINEHANDLE, LINEERR_INVALPOINTER, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

lineSetAgentGroup Overview

Overview

Overview

The **lineSetAgentGroup** function sets the agent groups into which the agent is logged into on a particular address.

```
LONG lineSetAgentGroup(

  HLINE hLine,
  DWORD dwAddressID,
  LPLINEAGENTGROUPLIST lpAgentGroupList
);
```

Parameters

hLine

Handle of the line device.

dwAddressID

Identifier of the address for which the agent information is to be changed.

lpAgentList

Pointer to a **LINEAGENTGROUPLIST** structure identifying the groups into which the current agent is to be logged in on the address. If the pointer is NULL or the number of groups in the indicated structure is 0, then the agent is logged out of any ACD groups into which the agent is then logged in.

Note that the "Name" fields in the **LINEAGENTGROUPENTRY** items in the list are ignored for purposes of this function; the control of the logged-in groups is based on the group ID values only.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALADDRESSID, LINEERR_INVALADDRESSSTATE, LINEERR_INVALAGENTGROUP, LINEERR_INVALAGENTID, LINEERR_INVALAGENTSKILL, LINEERR_INVALAGENTSUPERVISOR, LINEERR_INVALLINEHANDLE, LINEERR_INVALPARAM, LINEERR_INVALPASSWORD, LINEERR_INVALPOINTER, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

See Also

LINEAGENTGROUPENTRY, **LINEAGENTGROUPLIST**

lineSetAgentState Overview

Overview

Overview

The **lineSetAgentState** function sets the agent state associated with a particular address.

```
LONG lineSetAgentState(

  HLINE hLine,
  DWORD dwAddressID,
  DWORD dwAgentState,
  DWORD dwNextAgentState
);
```

Parameters

hLine

Handle of the line device.

dwAddressID

Identifier of the address for which the agent information is to be changed.

dwAgentState

The new agent state. Must be one of the LINEAGENTSTATE_ constants, or 0 to leave the agent state unchanged and modify only the next state.

dwNextAgentState

The agent state that should be automatically set when the current call on the address becomes *idle*. For example, if it is known that after-call work must be performed, this field can be set to LINEAGENTSTATE_WORKAFTERCALL so that a new call will not be assigned to the agent after the current call. Must be one of the LINEAGENTSTATE_ constants, or 0 to use the default next state configured for the agent.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALADDRESSID, LINEERR_INVALADDRESSSTATE, LINEERR_INVALAGENTSTATE, LINEERR_INVALLINEHANDLE, LINEERR_INVALPARAM, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

lineSetAppPriority Overview

Overview

Overview

The **lineSetAppPriority** function allows an application to set its priority in the handoff priority list for a particular media mode or Assisted Telephony request mode or to remove itself from the priority list.

```
LONG lineSetAppPriority(

  LPCSTR lpszAppFilename,
  DWORD dwMediaMode,
  LPLINEEXTENSIONID const lpExtensionID,
  DWORD dwRequestMode,
  LPCSTR lpszExtensionName,
  DWORD dwPriority
);
```

Parameters

lpszAppFilename

A pointer to a string containing the application executable module filename (without directory information). In API versions 0x00020000 and greater, the parameter can specify a filename in either long or 8.3 filename format.

dwMediaMode

The media mode for which the priority of the application is to be set. The value may be one of the LINEMEDIAMODE_ constants; only a single bit may be on. The value 0 should be used to set the application priority for Assisted Telephony requests.

lpExtensionID

A pointer to structure of type **LINEEXTENSIONID**. This parameter is ignored.

dwRequestMode

If the *dwMediaMode* parameter is 0, 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 is ignored if *dwMediaMode* is non-zero.

lpszExtensionName

This parameter is ignored.

dwPriority

The new priority for the application. If the value 0 is passed, the application is removed from the priority list for the specified media or request mode (if it was already not present, no error is generated). If the value 1 is passed, the application is 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 is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INIFILECORRUPT, LINEERR_INVALREQUESTMODE, LINEERR_INVALAPPNAME,

LINEERR_NOMEM, LINEERR_INVALMEDIAMODE, LINEERR_OPERATIONFAILED, LINEERR_INVALPARAM, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER.

Remarks

If LINEERR_INVALMEDIAMODE is returned, the value specified in *dwMediaMode* is not 0 and not a LINEMEDIAMODE_ constant, or more than one bit is on in the parameter value.

This function updates to stored priority list. If the telephony system is initialized, it also sets the current, active priorities for applications then running; the new priority will be used on the next incoming call or **lineHandoff** based on media mode.

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so. The function will work the same way for all applications.

See Also

LINEEXTENSIONID, **lineHandoff**

lineSetAppSpecific Overview

Overview

Overview

The **lineSetAppSpecific** function enables an application to set the application-specific field of the specified call's call-information record.

```
LONG lineSetAppSpecific(

  HCALL hCall,
  DWORD dwAppSpecific
);
```

Parameters

hCall

A handle to the call whose application-specific field needs to be set. The application must be an owner of the call. The call state of *hCall* can be any state.

dwAppSpecific

The new content of the **dwAppSpecific** field for the call's **LINECALLINFO** structure. This value is not interpreted by the Telephony API.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_NOTOWNER, LINEERR_OPERATIONUNAVAIL, LINEERR_OPERATIONFAILED.

Remarks

The application-specific field in the **LINECALLINFO** data structure that exists for each call is not interpreted by the Telephony API or any of its service providers. Its usage is entirely defined by the applications. The field can be read from the **LINECALLINFO** record returned by **lineGetCallInfo**. However, **lineSetAppSpecific** must be used to set the field so that changes become visible to other applications. When this field is changed, all other applications with call handles are sent a LINE_CALLINFO message with an indication that the **dwAppSpecific** field has changed.

See Also

LINE_CALLINFO, **LINECALLINFO**, **lineGetCallInfo**

lineSetCallData Overview

Overview

Overview

The **lineSetCallData** function sets the **CallData** member in **LINECALLINFO**. Depending on the service provider implementation, the **CallData** member may be propagated to all applications having handles to the call, including those on other machines (through the server), and may travel with the call when it is transferred.

```
LONG lineSetCallData(

  HCALL hCall,
  LPVOID lpCallData,
  DWORD dwSize
);
```

Parameters

hCall

Handle to the call. The application must have OWNER privilege.

lpCallData

Address of the data to be copied to the **CallData** member in **LINECALLINFO**, replacing any existing data.

dwSize

Number of bytes of data to be copied. A value of 0 causes any existing data to be removed.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALCALLHANDLE, LINEERR_INVALCALLSTATE, LINEERR_INVALPARAM, LINEERR_INVALPOINTER, LINEERR_NOMEM, LINEERR_NOTOWNER, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

See Also

LINECALLINFO

lineSetCallParams Overview

Overview

Overview

The **lineSetCallParams** function allows an application to change bearer mode and/or the rate parameters of an existing call.

```
LONG lineSetCallParams(

  HCALL hCall,
  DWORD dwBearerMode,
  DWORD dwMinRate,
  DWORD dwMaxRate,
  LPLINEDIALPARAMS const lpDialParams
);
```

Parameters

hCall

A handle to the call whose parameters are to be changed. The application must be an owner of the call. The call state of *hCall* can be any state except *idle* and *disconnected.*

dwBearerMode

The new bearer mode for the call. This parameter can have only a single bit set, and it uses the following LINEBEARERMODE_ constants:

LINEBEARERMODE_VOICE

A regular 3.1 kHz analog voice-grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH

This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation, and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE

The multi-use mode defined by ISDN.

LINEBEARERMODE_DATA

The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA

The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING

Corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a media stream by the Telephony API).

LINEBEARERMODE_PASSTHROUGH

When a call is active in LINEBEARERMODE_PASSTHROUGH, the service provider gives direct access to the attached hardware for control by the application. This mode is used primarily by applications desiring temporary direct control over asynchronous modems, accessed through the Win32 comm functions, for the purpose of configuring or using special features not otherwise

supported by the service provider.

dwMinRate

A lower bound for the call's new data rate. The application is willing to accept a new rate as low as this one.

dwMaxRate

An upper bound for the call's new data rate. This is the maximum data rate the application can accept. If an exact data rate is required, *dwMinRate* and *dwMaxRate* should be equal.

lpDialParams

A pointer to the new dial parameters for the call, of type **LINEDIALPARAMS**. This parameter can be left NULL if the call's current dialing parameters are to be used.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_BEARERMODEUNAVAIL, LINEERR_NOTOWNER, LINEERR_INVALBEARERMODE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSTATE, LINEERR_RATEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALRATE, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Remarks

This operation is used to change the parameters of an existing call. Examples of its usage include changing the bearer mode and/or the data rate of an existing call.

See Also

LINE_REPLY, **LINEDIALPARAMS**

lineSetCallPrivilege Overview

Overview

Overview

The **lineSetCallPrivilege** function sets the application's privilege to the specified privilege.

```
LONG 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 the application wants to have for the specified call. Only a single flag can be set. This parameter uses the following LINECALLPRIVILEGE_ constants:

LINECALLPRIVILEGE_MONITOR

The application requests monitor privilege to the call. These privileges allow the application to monitor state changes and to query information and status about the call.

LINECALLPRIVILEGE_OWNER

The application requests owner privilege to the call. These privileges allow the application to manipulate the call in ways that affect the state of the call.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSTATE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALCALLPRIVILEGE, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

Remarks

If the application is the sole owner of a non-idle call and wants to change its privilege to monitor, a LINEERR_INVALCALLSTATE error is returned. If the application wants to, it can first drop the call using **lineDrop** to make the call transition to the *idle* state and then change its privilege.

See Also

lineDrop

lineSetCallQualityOfService Overview

Overview

Overview

The **lineSetCallQualityOfService** function allows the application to attempt to change the quality of service parameters (reserved capacity and performance guarantees) for an existing call. Except for basic parameter validation, this is a straight pass-through to a service provider.

```
LONG lineSetCallQualityOfService(

  HCALL hCall,
  LPVOID lpSendingFlowspec,
  DWORD dwSendingFlowspecSize,
  LPVOID lpReceivingFlowspec,
  DWORD dwReceivingFlowspecSize
);
```

Parameters

hCall

Handle to the call. The application must have OWNER privilege.

lpSendingFlowspec

Pointer to memory containing a WinSock2 **FLOWSPEC** structure followed by provider-specific data. The provider-specific portion following the **FLOWSPEC** structure must not contain pointers to other blocks of memory in the application process, because TAPI will not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the service provider.

dwSendingFlowspecSize

The total size in bytes of the **FLOWSPEC** structure and accompanying provider-specific data, equivalent to what would have been stored in SendingFlowspec.len in a WinSock2 **QOS** structure.

lpReceivingFlowspec

Pointer to memory containing a WinSock2 **FLOWSPEC** structure followed by provider-specific data. The provider-specific portion following the **FLOWSPEC** structure must not contain pointers to other blocks of memory in the application process, because TAPI will not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the service provider.

dwReceivingFlowspecSize

The total size in bytes of the **FLOWSPEC** and accompanying provider-specific data, equivalent to what would have been stored in ReceivingFlowspec.len in a WinSock2 **QOS** structure.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALCALLHANDLE, LINEERR_INVALCALLSTATE, LINEERR_INVALPARAM, LINEERR_INVALPOINTER, LINEERR_INVALRATE, LINEERR_NOMEM, LINEERR_NOTOWNER, LINEERR_OPERATIONUNAVAIL, LINEERR_OPERATIONFAILED, LINEERR_RATEUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

lineSetCallTreatment Overview

Overview

Overview

The **lineSetCallTreatment** function sets what sounds a party on a call that is unanswered or on hold will hear. Except for basic parameter validation, it is a straight pass-through by TAPI to the service provider.

```
LONG lineSetCallTreatment(

  HCALL hCall,
  DWORD dwCallTreatment
);
```

Parameters

hCall

Handle to the call. The application must have OWNER privilege.

dwCallTreatment

One of the call treatments supported on the address on which the call appears, as indicated by **LINEADDRESSCAPS**. LINEERR_INVALPARAM is returned if the specified treatment is not supported.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALCALLHANDLE, LINEERR_INVALCALLSTATE, LINEERR_INVALPARAM, LINEERR_NOMEM, LINEERR_NOTOWNER, LINEERR_OPERATIONFAILED, LINEERR_OPERATIONUNAVAIL, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

Remarks

The use of call treatment functionality should be limited to controlling of legacy equipment. New equipment is generally designed so that instead of call treatments being generated by external switching equipment, calls would be connected to PC-based audio sources and signals generating using standard Win32 functions such as the multimedia Wave API.

See Also

LINEADDRESSCAPS

lineSetCurrentLocation Overview

Overview

Overview

The **lineSetCurentLocation** function sets the location used as the context for address translation.

LONG lineSetCurrentLocation(

HLINEAPP *hLineApp*,
DWORD *dwLocation*
);

Parameters

hLineApp

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

dwLocation

Specifies a new value for the CurrentLocation entry in the [Locations] section in the registry. It must contain a valid permanent ID of a Location entry in the [Locations] section, as obtained from **lineGetTranslateCaps**. If it is valid, the CurrentLocation entry is updated.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INIFILECORRUPT, LINEERR_NOMEM, LINEERR_INVALAPPHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALLOCATION, LINEERR_RESOURCEUNAVAIL, LINEERR_NODRIVER, LINEERR_UNINITIALIZED.

See Also

lineGetTranslateCaps, **lineInitializeEx**

lineSetDevConfig Overview

Overview

Overview

The **lineSetDevConfig** function allows the application to restore the configuration of a media stream device on a line device to a setup previously obtained using **lineGetDevConfig**. For example, the contents of this structure could specify data rate, character format, modulation schemes, and error control protocol settings for a "datamodem" media device associated with the line.

```
LONG lineSetDevConfig(

  DWORD dwDeviceID,
  LPVOID const lpDeviceConfig,
  DWORD dwSize,
  LPCSTR lpszDeviceClass
);
```

Parameters

dwDeviceID

The line device to be configured.

lpDeviceConfig

A pointer to the opaque configuration data structure that was returned by **lineGetDevConfig** in the variable portion of the **VARSTRING** structure.

dwSize

The number of bytes in the structure pointed to by *lpDeviceConfig*. This value will have been returned in the **dwStringSize** field in the **VARSTRING** structure returned by **lineGetDevConfig**.

lpszDeviceClass

A pointer to a NULL-terminated ASCII string that specifies the device class of the device whose configuration is to be set. Valid device class strings are the same as those specified for the **lineGetID** function.

Return Values

Returns zero if the function is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INVALDEVICECLASS, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALPOINTER, LINEERR_OPERATIONFAILED, LINEERR_INVALPARAM, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINESTATE, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_NODEVICE.

Remarks

Call states are device specific.

Typically, an application will call **lineGetID** to identify the media stream device associated with a line, and then call **lineConfigDialog** to allow the user to set up the device configuration. It could then call **lineGetDevConfig** and save the configuration information in a phone book or other database associated with a particular call destination. When the user wants to call the same destination again, this function **lineSetDevConfig** can be used to restore the configuration settings selected by the user. The

lineSetDevConfig, **lineConfigDialog**, and **lineGetDevConfig** functions can be used, in that order, to allow the user to view and update the settings.

The exact format of the data contained within the structure is specific to the line and media stream API (device class), is undocumented, and is undefined. The application must treat it as "opaque" and not manipulate the contents directly. Generally, the structure can be sent using this function only to the same device from which it was obtained. Certain Telephony service providers may, however, permit structures to be interchanged between identical devices (that is, multiple ports on the same multi-port modem card). Such interchangeability is not guaranteed in any way, even for devices of the same device class.

Note that some service providers may permit the configuration to be set while a call is active, and others may not.

See Also

lineConfigDialog, **lineGetDevConfig**, **lineGetID**, **VARSTRING**

lineSetLineDevStatus Overview

The **lineSetLineDevStatus** function sets the line device status. Except for basic parameter validation, it is a straight pass-through to the service provider. The service provider will send a LINE_LINEDEVSTATUS message to inform applications of the new state, when set; TAPI does not synthesize these messages.

```
LONG WINAPI lineSetLineDevStatus(

  DWORD hLine,
  DWORD dwStatusToChange,
  DWORD fStatus
);
```

Parameters

hLine

Handle to the line device.

dwStatusToChange

One or more of the LINEDEVSTATUSFLAGS_ values.

fStatus

TRUE (-1) to turn on the indicated status bit(s), FALSE (0) to turn off.

Return Values

Returns a positive request identifier if the asynchronous operation starts; otherwise, one of these negative error values:

LINEERR_INVALLINEHANDLE, LINEERR_INVALLINESTATE, LINEERR_INVALPARAM, LINEERR_NOMEM, LINEERR_OPERATIONUNAVAIL, LINEERR_OPERATIONFAILED, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED.

lineSetMediaControl Overview

Overview

Overview

The **lineSetMediaControl** function enables and disables control actions on the media stream associated with the specified line, address, or call. Media control actions can be triggered by the detection of specified digits, media modes, custom tones, and call states.

```
LONG lineSetMediaControl(

  HLINE hLine,
  DWORD dwAddressID,
  HCALL hCall,
  DWORD dwSelect,
  LPLINEMEDIACONTROLDIGIT const lpDigitList,
  DWORD dwDigitNumEntries,
  LPLINEMEDIACONTROLMEDIA const lpMediaList,
  DWORD dwMediaNumEntries,
  LPLINEMEDIACONTROLTONE const lpToneList,
  DWORD dwToneNumEntries,
  LPLINEMEDIACONTROLCALLSTATE const lpCallStateList,
  DWORD dwCallStateNumEntries
);
```

Parameters

hLine

SAa handle to an open line device.

dwAddressID

An address on the given open line device.

hCall

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

dwSelect

Specifies whether the media control requested is associated with a single call, is the default for all calls on an address, or is the default for all calls on a line. This parameter can only have a single flag set, and it 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.

lpDigitList

A pointer to the array that contains the digits that are to trigger media control actions, of type **LINEMEDIACONTROLDIGIT**. Each time a digit in the digit list is detected, the specified media control action is carried out on the call's media stream.

Valid digits for pulse mode are '0' through '9'. Valid digits for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#'.

dwDigitNumEntries

The number of entries in the *lpDigitList*.

lpMediaList

A pointer to an array with entries of type **LINEMEDIACONTROLMEDIA**. The array has *dwMediaNumEntries* entries. Each entry contains a media mode to be monitored, media-type specific information (such as duration), and a media control field. If a media mode in the list is detected, the corresponding media control action is performed on the call's media stream.

dwMediaNumEntries

The number of entries in *lpMediaList*.

lpToneList

A pointer to an array with entries of type **LINEMEDIACONTROLTONE**. The array has *dwToneNumEntries* entries. Each entry contains a description of a tone to be monitored, duration of the tone, and a media-control field. If a tone in the list is detected, the corresponding media control action is performed on the call's media stream.

dwToneNumEntries

The number of entries in *lpToneList*.

lpCallStateList

A pointer to an array with entries are of type **LINEMEDIACONTROLCALLSTATE**. The array has *dwCallStateNumEntries* entries. Each entry contains a call state and a media control action. Whenever the given call transitions into one of the call states in the list, the corresponding media control action is invoked.

dwCallStateNumEntries

The number of entries in *lpCallStateList*.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESSID, LINEERR_NOMEM, LINEERR_INVALCALLHANDLE, LINEERR_NOTOWNER, LINEERR_INVALCALLSELECT, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATELIST, LINEERR_OPERATIONFAILED, LINEERR_INVALDIGITLIST, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINEHANDLE, LINEERR_UNINITIALIZED, LINEERR_INVALMEDIALIST, LINEERR_INVALPOINTER, LINEERR_INVALTONELIST.

Remarks

The **lineSetMediaControl** function is considered successful if media control has been correctly initiated, not when any media control has taken effect. Media control in progress is changed or is canceled by calling this function again with either different parameters or NULLs. If one or more of the parameters *lpDigitList*, *lpMediaList*, *lpToneList,* and *lpCallStateList* are NULL, then the corresponding digit, media mode, tone, or call state-triggered media control is disabled. To modify just a portion of the media control parameters while leaving the remaining settings in effect, the application should invoke

lineSetMediaControl supplying the previous parameters for those portions that must remain in effect, and new parameters for those parts that are to be modified.

If *hCall* is selected and the call terminates or the application deallocates its handle, media control on that call is canceled.

All applications that are owners of the call are in principle allowed to make media control requests on the call. Only a single media control request can be outstanding on a call across all applications that own the call. Each time **lineSetMediaControl** is called, the new request overrides any media control then in effect on the call, whether set by the calling application or any other owning application.

Depending on the service provider and other activities that compete for such resources, the amount of simultaneous detections that can be made may vary over time. If service provider resources are overcommitted, the LINEERR_RESOURCEUNAVAIL error is returned.

Whether or not media control is supported by the service provider is a device capability.

See Also

LINEMEDIACONTROLCALLSTATE, **LINEMEDIACONTROLDIGIT**, **LINEMEDIACONTROLMEDIA**, **LINEMEDIACONTROLTONE**

lineSetMediaMode Overview

Overview

Overview

The **lineSetMediaMode** function sets the media mode(s) of the specified call in its **LINECALLINFO** structure.

```
LONG lineSetMediaMode(

  HCALL hCall,
  DWORD dwMediaModes
);
```

Parameters

hCall

A handle to the call whose media mode is to be changed. The application must be an owner of the call. The call state of *hCall* can be any state.

dwMediaModes

The new media mode(s) for the call. As long as the UNKNOWN media mode flag is set, other media mode flags may be set as well. This is used to identify a call's media mode as not fully determined, but narrowed down to one of a small set of specified media modes. If the UNKNOWN flag is not set, only a single media mode can be specified. This parameter uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN

The target application is the one that handles calls of unknown media mode (unclassified calls).

LINEMEDIAMODE_INTERACTIVEVOICE

The target application is the one that handles calls with the interactive voice media mode (live conversations).

LINEMEDIAMODE_AUTOMATEDVOICE

Voice energy is present on the call, and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

The target application is the one that handles calls with the data modem media mode.

LINEMEDIAMODE_G3FAX

The target application is the one that handles calls with the group 3 fax media mode.

LINEMEDIAMODE_TDD

The target application is the one that handles calls with the TDD (Telephony Devices for the Deaf) media mode.

LINEMEDIAMODE_G4FAX

The target application is the one that handles calls with the group 4 fax media mode.

LINEMEDIAMODE_DIGITALDATA

The target application is the one that handles calls that are digital data calls.

LINEMEDIAMODE_TELETEX

The target application is the one that handles calls with the teletex media mode.

LINEMEDIAMODE_VIDEOTEX

The target application is the one that handles calls with the videotex media mode.

LINEMEDIAMODE_TELEX

The target application is the one that handles calls with the telex media mode.

LINEMEDIAMODE_MIXED

The target application is the one that handles calls with the ISDN mixed media mode.

LINEMEDIAMODE_ADSI

The target application is the one that handles calls with the ADSI (Analog Display Services Interface) media mode.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALMEDIAMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONUNAVAIL.

Remarks

The **lineSetMediaMode** function changes the call's media mode in its **LINECALLINFO** structure. Typical usage of this operation is either to set a call's media mode to a specific known media mode or to exclude possible media modes as long as the call's media mode is officially unknown (the UNKNOWN media mode flag is set).

See Also

LINECALLINFO

lineSetNumRings Overview

Overview

Overview

The **lineSetNumRings** function sets the number of rings that must occur before an incoming call is answered. This function can be used 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 all number of rings requested. It can be used by the application that answers inbound calls to determine the number of rings it should wait before answering the call.

```
LONG lineSetNumRings(

  HLINE hLine,
  DWORD dwAddressID,
  DWORD dwNumRings
);
```

Parameters

hLine

A handle to the open line device.

dwAddressID

An address on the line device.

dwNumRings

The number of rings before a call should be answered in order to honor the toll-saver requests from all applications.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALLINEHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALADDRESSID, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

The **lineGetNumRings** and **lineSetNumRings** functions, when used in combination, provide a mechanism to support the implementation of toll-saver features across multiple independent applications. If no application ever calls **lineSetNumRings**, **lineGetNumRings** will return 0xFFFFFFFF.

An application that is the owner of a call in the *offering* state and that received a LINE_LINEDEVSTATE *ringing* message should wait a number of rings equal to the number returned by **lineGetNumRings** before answering the call in order to honor the toll-saver settings across all applications. A separate LINE_LINEDEVSTATE *ringing* message is sent to the application for each ring cycle, so the application should count these messages. If this call disconnects before being answered, and another call comes in shortly thereafter, the LINE_CALLSTATE message should allow the application to determine that ringing is related to the second call.

If call classification is performed by TAPI by means of answering inbound calls of unknown media mode and filtering the media stream, TAPI will honor this number as well.

Note that this operation is purely informational and does not affect the state of any calls on the line device.

See Also

LINE_CALLSTATE, LINE_LINEDEVSTATE, **lineGetNumRings**

lineSetStatusMessages Overview

Overview

Overview

The **lineSetStatusMessages** enables an application to specify which notification messages the application wants to receive for events related to status changes for the specified line or any of its addresses.

```
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 those listed below have changed. The application should check the current device status to determine which items have 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 containing this constant. For example, in the United States, service providers send a message with this constant every six seconds.

LINEDEVSTATE_CONNECTED

The line was previously disconnected and is now connected to TAPI.

LINEDEVSTATE_DISCONNECTED

This line was previously connected and is now disconnected from TAPI.

LINEDEVSTATE_MSGWAITON

The "message waiting" indicator is turned on.

LINEDEVSTATE_MSGWAITOFF

The "message waiting" indicator is turned off.

LINEDEVSTATE_INSERVICE

The line is connected to TAPI. This happens when TAPI is first activated, or when the line wire is physically plugged in and in service at the switch while TAPI is active.

LINEDEVSTATE_OUTOFSERVICE

The line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE

Maintenance is being performed on the line at the switch. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_OPEN

The line has been opened by some application.

LINEDEVSTATE_CLOSE

The line has been closed by some application.

LINEDEVSTATE_NUMCALLS

The number of calls on the line device has changed.

LINEDEVSTATE_NUMCOMPLETIONS

The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_TERMINALS

The terminal settings have changed.

LINEDEVSTATE_ROAMMODE

The roam mode of the line device has changed.

LINEDEVSTATE_BATTERY

The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL

The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC

The line's device-specific information has changed.

LINEDEVSTATE_REINIT

Items have 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 are denied until applications have shut down their usage of TAPI. The *hDevice* parameter of the LINE_LINEDEVSTATE message is left NULL for this state change as it applies to any of the 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 are always delivered to the application.

LINEDEVSTATE_LOCK

The locked status of the line device has changed.

LINEDEVSTATE_REMOVED

Indicates that the device is being removed from the system by the service provider (most likely through user action, via a control panel or similar utility). A LINE_LINEDEVSTATE message with this value will normally be immediately followed by a LINE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in LINEERR_NODEVICE being returned to the application. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated API TAPI version 0x00010004 or above; applications negotiating a previous API version will 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_ constants:

LINEADDRESSSTATE_OTHER

Address-status items other than those listed below have changed. The application should check the current address status to determine which items have changed.

LINEADDRESSSTATE_DEVSPECIFIC

The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO

The address has changed to idle (it is now in use by zero stations).

LINEADDRESSSTATE_INUSEONE

The address has changed from idle or from being in use by many bridged stations to being in use by just one station.

LINEADDRESSSTATE_INUSEMANY

The monitored or bridged address has changed from being in use by one station to being used by more than one station.

LINEADDRESSSTATE_NUMCALLS

The number of calls on the address has changed. This is the result of 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 has changed (including the number of rings for determining a no-answer condition). The application should check the address status to determine details about the address's current forwarding status.

LINEADDRESSSTATE_TERMINALS

The terminal settings for the address have changed.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESSSTATE, LINEERR_OPERATIONFAILED, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINESTATE, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_OPERATIONUNAVAIL.

Remarks

TAPI defines a number of messages that notify applications about events occurring on lines and addresses. An application may not be interested in receiving all address and line status change messages. The **lineSetStatusMessages** function can be used to select which messages the application wants to receive. By default, address and line status reporting is disabled.

See Also

LINE_CLOSE, LINE_LINEDEVSTATE, **lineInitialize**, **lineInitializeEx**, **lineOpen**

lineSetTerminal Overview

Overview

Overview

The **lineSetTerminal** function enables an application to specify which terminal information related to the specified line, address, or call is to be routed. The **lineSetTerminal** function can be used while calls are in progress on the line to allow an application to route these events to different devices as required.

```
LONG lineSetTerminal(

  HLINE hLine,
  DWORD dwAddressID,
  HCALL hCall,
  DWORD dwSelect,
  DWORD dwTerminalModes,
  DWORD dwTerminalID,
  DWORD bEnable
);
```

Parameters

hLine

A handle to an open line device.

dwAddressID

An address on the given open line device.

hCall

A handle to a call. The call state of *hCall* can be any state, if *dwSelect* is CALL.

dwSelect

Specifies whether the terminal setting is requested for the line, the address, or just the specified call. If line or address is specified, events either apply to the line or address itself or serves as a default initial setting for all new calls on the line or address. 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.

dwTerminalModes

The class or classes of low-level events to be routed to the given terminal. This parameter uses the following LINETERMMODE_ constants:

LINETERMMODE_BUTTONS

The button presses from the terminal to the line.

LINETERMMODE_DISPLAY

The display events from the line to the terminal.

LINETERMMODE_LAMPS

The lamp lighting events from the line to the terminal.

LINETERMMODE_RINGER

The ring requests from the line to the terminal.

LINETERMMODE_HOOKSWITCH

The hookswitch events between the terminal and the line.

LINETERMMODE_MEDIATOLINE

This is the unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE

This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT

This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when routing of both unidirectional channels of a call's media stream cannot be controlled independently. Note that MEDIABIDIRECT is mutually exclusive with MEDIATOLINE and MEDIAFROMLINE

dwTerminalID

The device ID of the terminal device where the given events are to be routed. Terminal IDs are small integers in the range of 0 to one less than **dwNumTerminals**, where **dwNumTerminals**, and the terminal modes each terminal is capable of handling, are returned by **lineGetDevCaps**. Note that these terminal IDs have no relation to other device IDs and are defined by the service provider using device capabilities.

bEnable

If TRUE, *dwTerminalID* is valid and the specified event classes are routed to or from that terminal. If FALSE, these events are not routed to or from the terminal device with ID equal to *dwTerminalID*.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESSID, LINEERR_NOMEM, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSELECT, LINEERR_OPERATIONFAILED, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALTERMINALID, LINEERR_UNINITIALIZED, LINEERR_INVALTERMINALMODE.

Remarks

An application can use this function to route certain classes of low-level line events to the specified terminal device or to suppress the routing of these events. For example, voice may be routed to an audio I/O device (headset); lamps and display events may be routed to the local phone device, and button events and ringer events may be suppressed altogether.

This function can be called at any time, even when a call is active on the given line device. This allows a user to switch from using the local phone set to another audio I/O device. This function may be called multiple times to route the same events to multiple terminals simultaneously. To reroute events to a different terminal, the application should first disable routing to the existing terminal and then route the events to the new terminal.

Terminal ID assignments are made by the line's service provider. Device capabilities indicate only which terminal IDs the service provider has available. Service providers that do not support this type of event routing would indicate that they have no terminal devices (**dwNumTerminals** in **LINEDEVCAPS** would be zero).

Invoking **lineSetTerminal** on a line or address affects all existing calls on that line or address, but does not affect calls on other addresses. It also sets the default for future calls on that line or address. A line or address that has multiple connected calls active at one time may have different routing in effect for each call.

Disabling the routing of low-level events to a terminal when these events are not currently routed to or from that terminal will not necessarily generate an error so long after the function succeeds (the specified events are not routed to or from that terminal).

TAPI routes call progress tones and messages to the same location as set by the **lineSetTerminal** function for "media". For example, if audio signals are going to the phone, then so will busy signals (analog) or Q.931 messages indicating busy (digital).

See Also

LINE_REPLY, **LINEDEVCAPS**, **lineGetDevCaps**

lineSetTollList Overview

Overview

Overview

The **lineSetTollList** function manipulates the toll list.

LONG lineSetTollList(

HLINEAPP *hLineApp*,
DWORD *dwDeviceID*,
LPCSTR *lpszAddressIn*,
DWORD *dwTollListOption*
);

Parameters

hLineApp

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

dwDeviceID

The device ID for the line device upon which the call is intended to be dialed, so that variations in dialing procedures on different lines can be applied to the translation process.

lpszAddressIn

A pointer to a NULL-terminated ASCII string containing the address from which the prefix information is to be extracted for processing. This parameter must not be NULL, and it must be in the canonical address format.

dwTollListOption

The toll list operation to be performed. Only a single flag can be set. This parameter uses the following LINETOLLLISTOPTION_ constants:

LINETOLLLISTOPTION_ADD

Causes the prefix contained within the string pointed to by *lpszAddressIn* to be added to the toll list for the current location.

LINETOLLLISTOPTION_REMOVE

Causes the prefix to be removed from the toll list of the current location. If toll lists are not used or relevant to the country indicated in the current location, the operation has no affect.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INVALAPPHANDLE, LINEERR_NOMEM, LINEERR_INVALADDRESS, LINEERR_OPERATIONFAILED, LINEERR_INVALPARAM, LINEERR_RESOURCEUNAVAIL, LINEERR_INIFILECORRUPT, LINEERR_UNINITIALIZED, LINEERR_INVALLOCATION.

See Also

lineInitializeEx

lineSetupConference Overview

Overview

Overview

The **lineSetupConference** function sets up a conference call for the addition of the third party.

```
LONG lineSetupConference(

  HCALL hCall,
  HLINE hLine,
  LPHCALL lphConfCall,
  LPHCALL lphConsultCall,
  DWORD dwNumParties,
  LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hCall

The initial call that identifies the first party of a conference call. In some environments (as described in device capabilities), a call must exist to start a conference call, and the application must be an owner of this call. In other telephony environments, no call initially exists, *hCall* must be left NULL, and *hLine* must be specified to identify the line on which the conference call is to be initiated. The call state of *hCall* must be *connected*.

hLine

A handle to the line. This handle is used to identify the line device on which to originate the conference call if *hCall* is NULL. The *hLine* parameter is ignored if *hCall* is non-NULL.

lphConfCall

A pointer to an HCALL handle. This location is then loaded with a handle identifying the newly created conference call. The application will be the initial sole owner of this call. The call state of *hConfCall* is not applicable.

lphConsultCall

A pointer to an HCALL handle. When setting up a call for the addition of a new party, a new temporary call (consultation call) is automatically allocated. Initially, the application will be the sole owner for this call.

dwNumParties

The expected number of parties in the conference call. This number is passed to the service provider. The service provider is free to do as it pleases with this number: ignore it, use it as a hint to allocate the right size conference bridge inside the switch, and so on.

lpCallParams

A pointer to call parameters to be used when establishing the consultation call. This parameter may be set to NULL if no special call setup parameters are desired.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values

are:

LINEERR_BEARERMODEUNAVAIL, LINEERR_UNINITIALIZED, LINEERR_CALLUNAVAIL, LINEERR_INVALMEDIAMODE, LINEERR_CONFERENCEFULL, LINEERR_INVALPOINTER, LINEERR_INUSE, LINEERR_INVALRATE, LINEERR_INVALADDRESSMODE, LINEERR_NOMEM, LINEERR_INVALBEARERMODE, LINEERR_NOTOWNER, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLPARAMS, LINEERR_RATEUNAVAIL, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINESTATE, LINEERR_STRUCTURETOOSMALL, LINEERR_USERUSERINFOTOOBIG.

Remarks

If LINEERR_INVALLINESTATE is returned, the line is currently not in a state in which this operation can be performed. A list of currently valid operations can be found in the **dwLineFeatures** field (of the type **LINEFEATURE**) in the **LINEDEVSTATUS** structure. (Calling **lineGetLineDevStatus** updates the information in **LINEDEVSTATUS**.) If LINEERR_INVALMEDIAMODE is returned, check for supported media modes on the line in the **dwMediaModes** field in the **LINEDEVCAPS** structure.

The **lineSetupConference** function provides two ways to establish a new conference call, depending on whether a normal two-party call is required to pre-exist or not. When setting up a conference call from an existing two-party call, the *hCall* parameter is a valid call handle that is initially added to the conference call by the **lineSetupConference** request; *hLine* is ignored. On switches where conference call setup does not start with an existing call, *hCall* must be NULL and *hLine* must be specified to identify the line device on which to initiate the conference call. In either case, a consultation call is allocated for connecting to the party that is to be added to the call. The application can then use **lineDial** to dial the address of the other party.

The conference call typically transitions into the *onHoldPendingConference* state, the consultation call into the *dialtone* state, and the initial call (if there is one) into the *conferenced* state.

A conference call can also be set up by a **lineCompleteTransfer** that is resolved into a three-way conference. The application may be able to toggle between the consultation call and the conference call using **lineSwapHold**.

A consultation call can be canceled by invoking lineDroponit. When dropping a consultation call, the existing conference call typically transitions back to the *connected* state. The application should observe the LINE_CALLSTATE messages to determine exactly what happens to the calls. For example, if the conference call reverts back to a regular two-party call, the conference call will become idle and the original participant call may revert to *connected*.

If an application specifies the handle of the original call (*hCall*) in a call to the **lineUnhold** function, both the conference call and the consultation call typically go to idle.

See Also

LINE_CALLSTATE, **lineCompleteTransfer**, **LINEDEVCAPS**, **LINEDEVSTATUS**, **lineDial**, **lineGetLineDevStatus**, **lineSwapHold**, **lineUnhold**

lineSetupTransfer Overview

Overview

Overview

The **lineSetupTransfer** function initiates a transfer of the call specified by *hCall*. 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 *lphConsultCall*.

```
LONG lineSetupTransfer(

  HCALL hCall,
  LPHCALL lphConsultCall,
  LPLINECALLPARAMS const lpCallParams
);
```

Parameters

hCall

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

lphConsultCall

A pointer to an HCALL handle. This location is then loaded with a handle identifying the temporary consultation call. When setting up a call for transfer, another call (a consultation call) is automatically allocated to enable the application to dial the address (using **lineDial**) of the party to where the call is to be transferred. The originating party can carry on a conversation over this consultation call prior to completing the transfer. Call state of *hConsultCall* is not applicable.

This transfer procedure may not be valid for some line devices. The application may need to ignore the new consultation call and unhold an existing held call (using **lineUnhold**) to identify the destination of the transfer. On switches that support cross-address call transfer, the consultation call may exist on a different address than the call to be transferred. It may also be necessary that the consultation call be set up as an entirely new call, by **lineMakeCall**, to the destination of the transfer. Which forms of transfer are available are specified in the call's address capabilities.

lpCallParams

A pointer to call parameters to be used when establishing the consultation call. This parameter may be set to NULL if no special call setup parameters are desired.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_BEARERMODEUNAVAIL, LINEERR_INVALRATE, LINEERR_CALLUNAVAIL, LINEERR_NOMEM, LINEERR_INUSE, LINEERR_NOTOWNER, LINEERR_INVALADDRESSMODE, LINEERR_OPERATIONFAILED, LINEERR_INVALBEARERMODE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_RATEUNAVAIL, LINEERR_INVALCALLPARAMS, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALLINESTATE, LINEERR_UNINITIALIZED, LINEERR_INVALMEDIAMODE, LINEERR_USERUSERINFOTOOBIG, LINEERR_INVALPOINTER.

Remarks

The **lineSetupTransfer** function sets up the transfer of the call specified by *hCall*. The setup phase of a transfer establishes a consultation call that enables the application to send the address of the destination (the party to be transferred to) to the switch, while the call to be transferred is kept on hold. This new call is referred to as a consultation call (*hConsultCall*) and can be dropped or otherwise manipulated independently of the original call.

When the consultation call has reached the *dialtone* call state, the application may proceed transferring the call either by dialing the destination address and tracking its progress, or by unholding an existing call. The transfer of the original call to the selected destination is completed using **lineCompleteTransfer**.

While the consultation call exists, the original call typically transitions to the *onholdPendingTransfer* state. The application may be able to toggle between the consultation call and the original call by using **lineSwapHold**. A consultation call can be canceled by invoking **lineDrop** on it. After dropping a consultation call, the original call will typically transition back to the *connected* state. If the call state of the original call is *onHoldPendingTransfer*, the **lineUnhold** function can be used to recover the call. In this case, the call state of the consultation call is set to *idle*.

The application may also transfer calls in a single step, without having to deal with the intervening consultation call by using **lineBlindTransfer**.

See Also

LINE_REPLY, **lineBlindTransfer**, **lineCompleteTransfer**, **lineDial**, **lineDrop**, **lineMakeCall**, **lineSwapHold**, **lineUnhold**

lineShutdown Overview

Overview

Overview

The **lineShutdown** function shuts down the application's usage of the line abstraction of API.

LONG lineShutdown(

HLINEAPP *hLineApp*
);

Parameters

hLineApp

The application's usage handle for the line API.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED.

Remarks

If this function is called when the application has lines open or calls active, the call handles are deleted and TAPI automatically performs the equivalent of a **lineClose** on each open line. However, it is recommended that applications explicitly close all open lines before invoking **lineShutdown**. If shutdown is performed while asynchronous requests are outstanding, those requests will be canceled.

See Also

lineClose

lineSwapHold Overview

Overview

Overview

The **lineSwapHold** function swaps the specified active call with the specified call on consultation hold.

```
LONG lineSwapHold(

  HCALL hActiveCall,
  HCALL hHeldCall
);
```

Parameters

hActiveCall

The handle to the active call. The application must be an owner of the call. The call state of *hActiveCall* must be *connected*.

hHeldCall

The handle to the consultation call. The application must be an owner of the call. The call state of *hHeldCall* can be *onHoldPendingTransfer, onHoldPendingConference, onHold* .

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_RESOURCEUNAVAIL, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED.

Remarks

Swapping the active call with the call on consultation hold allows the application to alternate or toggle between these two calls. This is typical in call waiting.

See Also

LINE_REPLY

lineTranslateAddress Overview

Overview

Overview

The **lineTranslateAddress** function translates the specified address into another format.

```
LONG lineTranslateAddress(

  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  LPCSTR lpszAddressIn,
  DWORD dwCard,
  DWORD dwTranslateOptions,
  LPLINETRANSLATEOUTPUT lpTranslateOutput
);
```

Parameters

hLineApp

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

dwDeviceID

The device ID for the line device upon which the call is intended to be dialed, so that variations in dialing procedures on different lines can be applied to the translation process.

dwAPIVersion

Indicates the highest version of TAPI supported by the application (*not* necessarily the value negotiated by **lineNegotiateAPIVersion** on some particular line device).

lpszAddressIn

A pointer to a NULL-terminated ASCII string containing the address from which the information is to be extracted for translation. Must be in either 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 ASCII CR and LF characters, only the first address is translated, and the remainder of the string is returned to the application without modification.

dwCard

The credit card to be used for dialing. This field in only valid if the CARDOVERRIDE bit is set in *dwTranslateOptions*. This field specifies the permanent ID of a Card entry in the [Cards] section in the registry (as obtained from **lineTranslateCaps**) which should be used instead of the PreferredCardID 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 field is 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 the following LINETRANSLATEOPTION_ constants:

LINETRANSLATEOPTION_CARDOVERRIDE

If this bit is set, *dwCard* specifies the permanent ID of a Card entry in the [Cards] section in the

registry (as obtained from **lineTranslateCaps**) which should be used instead of the *PreferredCardID* 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. The *dwCard* field is ignored if the CARDOVERRIDE bit is not set.

LINETRANSLATEOPTION_CANCELCALLWAITING

If a Cancel Call Waiting string is defined for the location, setting this bit will cause that string to be inserted at the beginning of the dialable string. This is commonly used by data modem and fax applications to prevent interruption of calls by call waiting beeps. If no Cancel Call Waiting string is defined for the location, this bit has no affect. Note that applications using this bit are advised to also set the LINECALLPARAMFLAGS_SECURE bit in the **dwCallParamFlags** field of the **LINECALLPARAMS** structure passed in to **lineMakeCall** through the *lpCallParams* parameter, so that if the line device uses a mechanism other than dialable digits to suppress call interrupts that that mechanism will be invoked.

LINETRANSLATEOPTION_FORCELOCAL

If the number is local but would have been translated as a long distance call (LINETRANSLATERESULT_INTOLLLIST bit set in the **LINETRANSLATEOUTPUT** structure), this option will force it to be translated as local. This is a temporary override of the toll list setting.

LINETRANSLATEOPTION_FORCELD

If the address could potentially have been a toll call, but would have been translated as a local call (LINETRANSLATERESULT_NOTINTOLLLIST bit set in the **LINETRANSLATEOUT**PUT structure), this option will force it to be translated as long distance. This is a temporary override of the toll list setting.

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** field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

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.

See Also

lineInitializeEx, **LINECALLPARAMS**, **lineNegotiateAPIVersion**, **LINETRANSLATEOUTPUT**

lineTranslateDialog Overview

Overview

Overview

The **lineTranslateDialog** function displays an application-modal dialog which allows the user to change the current location, adjust location and calling card parameters, and see the effect on a phone number about to be dialed.

```
LONG lineTranslateDialog(

  HLINEAPP hLineApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  HWND hwndOwner,
  LPCSTR lpszAddressIn
);
```

Parameters

hLineApp

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

dwDeviceID

The device ID for the line device upon which the call is intended to be dialed, so that variations in dialing procedures on different lines can be applied to the translation process.

dwAPIVersion

Indicates the highest version of TAPI supported by the application (*not* necessarily the value negotiated by **lineNegotiateAPIVersion** on the line device indicated by *dwDeviceID*).

hwndOwner

A handle to a window to which the dialog is to be attached. Can be a NULL value to indicate that any window created during the function should have no owner window.

lpszAddressIn

A pointer to a NULL-terminated ASCII string containing a phone number which will be used, in the lower portion of the dialog, to show the effect of the user's changes to the location parameters. The number must be in canonical format; if non-canonical, the phone number portion of the dialog will not be displayed. This pointer can be left NULL, in which case the phone number portion of the dialog will not be displayed. If the *AddressIn* contains a subaddress or name field, or additional addresses separated from the first address by ASCII CR and LF characters, only the first address is used in the dialog.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

LINEERR_BADDEVICEID, LINEERR_INVALPARAM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_INVALPOINTER, LINEERR_INIFILECORRUPT, LINEERR_NODRIVER, LINEERR_INUSE, LINEERR_NOMEM, LINEERR_INVALADDRESS, LINEERR_INVALAPPHANDLE, LINEERR_OPERATIONFAILED.

Remarks

In API versions 0x00020000 and greater, it is possible for multiple instances of this dialog to be opened. In API versions less than 0x00020000, LINEERR_INUSE is returned if the dialog is already displayed by another application (cannot be open more than once). In these versions, TAPI brings the existing dialog to the front, and the error indicates that any particulars related to the address passed in by the current application have not been handled, because that address was not processed by the function.

The application must call **lineGetTranslateCaps** after this function to obtain any changes the user made to the telephony address translation parameters, and call **lineTranslateAddress** to obtain a dialable string based on the user's new selections.

If any function related to address translation (for example, **lineGetTranslateCaps** or **lineTranslateAddress**) returns LINEERR_INIFILECORRUPT, the application should call **lineTranslateDialog**. The **lineTranslateDialog** function will detect the errors and correct them, and report the action taken to the user. Note that LINEERR_INIFILECORRUPT will be returned the first time any of these functions are used after installation of Windows 95, because the parameters will be uninitialized (**lineTranslateDialog** will take care of initializing them, using the user-specified default country to select the default country code).

Although this is a new function which older applications would not be expected to call, for backward compatibility, they should not be prevented from doing so; the full range of API versions supported by TAPI (0x00010003 to 0x00010004) should work the same way.

See Also

lineGetTranslateCaps, **lineInitializeEx**, **lineNegotiateAPIVersion**, **lineTranslateAddress**

lineUncompleteCall Overview

Overview

Overview

The **lineUncompleteCall** function cancels the specified call completion request on the specified line.

```
LONG lineUncompleteCall(

  HLINE hLine,
  DWORD dwCompletionID
);
```

Parameters

hLine

A handle to the line device on which a call completion is to be canceled.

dwCompletionID

The completion ID for the request that is to be canceled.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALLINEHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCOMPLETIONID, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONUNAVAIL.

See Also

LINE_REPLY

lineUnhold Overview

Overview

Overview

The **lineUnhold** function retrieves the specified held call.

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*.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_RESOURCEUNAVAIL, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED.

See Also

LINE_REPLY

lineUnpark Overview

Overview

Overview

The **lineUnpark** function retrieves the call parked at the specified address and returns a call handle for it.

```
LONG lineUnpark(

  HLINE hLine,
  DWORD dwAddressID,
  LPHCALL lphCall,
  LPCSTR lpszDestAddress
);
```

Parameters

hLine

A handle to the open line device on which a call is to be unparked.

dwAddressID

The address on *hLine* at which the unpark is to be originated.

lphCall

A pointer to the location of type HCALL where the handle to the unparked call is returned. This handle is unrelated to any other handle which might have been previously associated with the retrieved call, such as the handle that might have been associated with the call when it was originally parked. The application will be the initial sole owner of this call.

lpszDestAddress

A pointer to a NULL-terminated character buffer that contains the address where the call is parked. The address is in standard dialable address format.

Return Values

Returns a positive request ID if the function will be completed asynchronously, or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALADDRESS, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALADDRESSID, LINEERR_OPERATIONFAILED, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM.

See Also

LINE_REPLY

Phone Device Functions

This section contains the telephony API phone device functions.

phoneClose Overview

Overview

Overview

The **phoneClose** function closes the specified open phone device.

```
LONG phoneClose(

  HPHONE hPhone
);
```

Parameters

hPhone

A handle to the open phone device to be closed. If the function is successful, the handle is no longer valid.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_OPERATIONFAILED, PHONEERR_RESOURCEUNAVAIL, PHONEERR_OPERATIONUNAVAIL, PHONEERR_UNINITIALIZED.

Remarks

After the open phone device has been successfully closed, the implementation sends a PHONE_CLOSE message to the application. Note that these messages can also be sent unsolicited as a result of the phone device being reclaimed somehow. An application should therefore be prepared to handle these unsolicited close messages. At the time the phone device is closed, any outstanding asynchronous replies are suppressed.

See Also

PHONE_CLOSE

phoneConfigDialog Overview

Overview

Overview

The **phoneConfigDialog** function causes the provider of the specified phone device to display a modal dialog (attached to the application's *hwndOwner*) that allows the user to configure parameters related to the phone device specified by *dwDeviceID*.

```
LONG phoneConfigDialog(

  DWORD dwDeviceID,
  HWND hwndOwner,
  LPCSTR lpszDeviceClass
);
```

Parameters

dwDeviceID

The phone device to be configured.

hwndOwner

A handle to a window to which the dialog is to be attached. Can be a NULL value to indicate that any window created during the function should have no owner window.

lpszDeviceClass

A pointer to a NULL-terminated string that identifies a device class name. This device class allows the application to select a specific subscreen of configuration information applicable to that device class. This parameter is optional and can be left NULL or empty, in which case the highest level configuration is selected.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_BADDEVICEID, PHONEERR_NOMEM, PHONEERR_INUSE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPARAM, PHONEERR_OPERATIONUNAVAIL, PHONEERR_INVALDEVICECLASS, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPOINTER, PHONEERR_UNINITIALIZED, PHONEERR_NODEVICE.

Remarks

The *lpszDeviceClass* parameter allows the application to select a specific subscreen of configuration information applicable to the device class in which the user is interested; the permitted strings are the same as for **phoneGetID**. For example, if the phone supports the wave API, passing "wave/in" as *lpszDeviceClass* would cause the provider to display the parameters related specifically to wave (or at least to start at the corresponding point in a multilevel configuration dialog chain, eliminating the need to search for relevant parameters).

The *lpszDeviceClass* parameter should be "tapi/phone", "", or NULL to cause the provider to display the highest level configuration for the phone.

See Also

phoneGetID

phoneDevSpecific Overview

Overview

Overview

The **phoneDevSpecific** function Is used as a general extension mechanism to enable a Telephony API implementation to provide features not described in the other TAPI functions. The meanings of these extensions are device specific.

```
LONG phoneDevSpecific(

  HPHONE hPhone,
  LPVOID lpParams,
  DWORD dwSize
);
```

Parameters

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. The contents of the parameter block are passed unchanged to or from the service provider by TAPI.

dwSize

The size in bytes of the parameter block area.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_OPERATIONUNAVAIL, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONFAILED.

Additional return values are device specific.

Remarks

This operation provides a generic parameter profile. The interpretation of the parameter block is device specific. Indications and replies that are device specific should use the PHONE_DEVSPECIFIC message.

A service provider can provide access to device-specific functions by defining parameters for use with this operation. Applications that want to make use of these device-specific extensions should consult the device-specific (vendor-specific) documentation that describes which extensions are defined. Note that an application that relies on these device-specific extensions will typically not be portable to work with other service-provider environments.

See Also

PHONE_DEVSPECIFIC, PHONE_REPLY

phoneGetButtonInfo Overview

Overview

Overview

The **phoneGetButtonInfo** function returns information about the specified button.

```
LONG phoneGetButtonInfo(

  HPHONE hPhone,
  DWORD dwButtonLampID,
  LPPHONEBUTTONINFO lpButtonInfo
);
```

Parameters

hPhone

A handle to the open phone device.

dwButtonLampID

A button on the phone device.

lpButtonInfo

A pointer to a variably sized structure of type **PHONEBUTTONINFO**. This data structure describes the mode, the function, and provides additional descriptive text corresponding to the button.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALBUTTONLAMPID, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPOINTER, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPHONESTATE, PHONEERR_STRUCTURETOOSMALL, PHONEERR_OPERATIONUNAVAIL, PHONEERR_UNINITIALIZED.

See Also

PHONEBUTTONINFO

phoneGetData Overview

Overview

Overview

The **phoneGetData** function uploads the information from the specified location in the open phone device to the specified buffer.

```
LONG phoneGetData(

  HPHONE hPhone,
  DWORD dwDataID,
  LPVOID lpData,
  DWORD dwSize
);
```

Parameters

hPhone

A handle to the open phone device.

dwDataID

Where in the phone device the buffer is to be uploaded from.

lpData

A pointer to the memory buffer where the data is to be uploaded.

dwSize

The size of the data buffer in bytes.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPHONESTATE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALDATAID, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONUNAVAIL.

Remarks

The function uploads a maximum of *dwSize* bytes from the phone device into the memory area pointed to by *lpData*. If *dwSize* is zero, nothing is copied. The size of each data area is listed in the phone's device capabilities.

phoneGetDevCaps Overview

Overview

Overview

The **phoneGetDevCaps** function queries a specified phone device to determine its telephony capabilities.

```
LONG phoneGetDevCaps(
  HPHONEAPP hPhoneApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  DWORD dwExtVersion,
  LPPHONECAPS lpPhoneCaps
);
```

Parameters

hPhoneApp

The handle to the application's registration with TAPI.

dwDeviceID

The phone device to be queried.

dwAPIVersion

The version number of the Telephony API to be used. The high-order word contains the major version number; the low-order word contains the minor version number. This number is obtained 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 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.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALAPPHANDLE, PHONEERR_INVALPOINTER, PHONEERR_BADDEVICEID, PHONEERR_OPERATIONFAILED, PHONEERR_INCOMPATIBLEAPIVERSION, PHONEERR_OPERATIONUNAVAIL, PHONEERR_INCOMPATIBLEEXTVERSION, PHONEERR_NOMEM, PHONEERR_STRUCTURETOOSMALL, PHONEERR_RESOURCEUNAVAIL, PHONEERR_NODRIVER, PHONEERR_UNINITIALIZED, PHONEERR_NODEVICE.

Remarks

Before using **phoneGetDevCaps**, the application must negotiate the TAPI version number to use, and optionally, the extension version to use.

TAPI and extension version numbers are those under which TAPI, Telephony DLL, and service provider must operate. If version ranges do not overlap, the application and API or service-provider versions are incompatible and an error is returned.

See Also

PHONECAPS, **phoneNegotiateAPIVersion**, **phoneNegotiateExtVersion**

phoneGetDisplay Overview

Overview

Overview

The **phoneGetDisplay** function returns the current contents of the specified phone display.

```
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**.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPOINTER, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPHONESTATE, PHONEERR_STRUCTURETOOSMALL, PHONEERR_OPERATIONUNAVAIL, PHONEERR_UNINITIALIZED, PHONEERR_NOMEM.

Remarks

The *lpDisplay* memory area should be at least (**dwDisplayNumRows** * **dwDisplayNumColumns**) elements in size to receive all of the display information. **dwDisplayNumRows** and **dwDisplayNumColumns** are available in the **PHONECAPS** structure returned by **phoneGetDevCaps**.

See Also

PHONECAPS, **phoneGetDevCaps**, **VARSTRING**

phoneGetGain Overview

Overview

Overview

The **phoneGetGain** function returns the gain setting of the microphone of the specified phone's hookswitch device.

```
LONG phoneGetGain(

  HPHONE hPhone,
  DWORD dwHookSwitchDev,
  LPDWORD lpdwGain
);
```

Parameters

hPhone

A handle to the open phone device.

dwHookSwitchDev

The hookswitch device whose gain level is queried. Note that *dwHookSwitchDev* can have only one bit set. This parameter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET

The phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER

The phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET

The phone's headset.

lpdwGain

A pointer to a DWORD-sized location containing the current gain setting of the hookswitch microphone component. The *dwGain* parameter specifies the volume level of the hookswitch device. This is a number in the range 0x00000000 (silence) to 0x0000FFFF (maximum volume). The actual granularity and quantization of gain settings in this range are service provider specific.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPHONESTATE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALHOOKSWITCHDEV, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONUNAVAIL.

phoneGetHookSwitch Overview

Overview

Overview

The **phoneGetHookSwitch** function returns the current hookswitch mode of the specified open phone device.

```
LONG phoneGetHookSwitch(

  HPHONE hPhone,
  LPDWORD lpdwHookSwitchDevs
);
```

Parameters

hPhone

A handle to the open phone device.

lpdwHookSwitchDevs

A pointer to a DWORD-sized location to be filled with the mode of the phone's hookswitch devices. If a bit position is FALSE, the corresponding hookswitch device is on-hook; if TRUE, the microphone and/or speaker part of the corresponding hookswitch device is offhook. To find out whether the microphone and/or speaker are enabled, the application can use **phoneGetStatus**. This parameter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET

The phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER

The phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET

The phone's headset.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPHONESTATE, PHONEERR_OPERATIONFAILED, PHONEERR_OPERATIONUNAVAIL, PHONEERR_UNINITIALIZED.

Remarks

After the hookswitch state of a device changes, and if hookswitch monitoring is enabled, the application is sent a PHONE_STATE message.

See Also

PHONE_STATE, **phoneGetStatus**

phoneGetIcon Overview

Overview

Overview

The **phoneGetIcon** function allows an application to retrieve a service phone device-specific (or provider-specific) icon that can be displayed to the user.

```
LONG phoneGetIcon(

  DWORD dwDeviceID,
  LPCSTR lpszDeviceClass,
  LPHICON lphIcon
);
```

Parameters

dwDeviceID

The phone device whose icon is requested.

lpszDeviceClass

A pointer to a NULL-terminated string that identifies a device class name. This device class allows the application to select a specific sub-icon applicable to that device class. This parameter is optional and can be left NULL or empty, in which case the highest-level icon associated with the phone device rather than a specified media stream device would be selected.

lphIcon

A pointer to a memory location in which the handle to the icon is returned.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_BADDEVICEID, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPOINTER, PHONEERR_OPERATIONFAILED, PHONEERR_INVALDEVICECLASS, PHONEERR_UNINITIALIZED, PHONEERR_NOMEM, PHONEERR_NODEVICE.

Remarks

The **phoneGetIcon** causes the provider to return a handle (in *lphIcon*) to an icon resource (obtained from **LoadIcon**) associated with the specified phone. The icon handle will be for a resource associated with the provider; the application must use **CopyIcon** if it wishes to reference the icon after the provider is unloaded, which is unlikely to happen as long as the application has the phone open.

The *lpszDeviceClass* parameter allows the provider to return different icons based on the type of service being referenced by the caller. The permitted strings are the same as for **phoneGetID**. For example, if the phone supports the Comm API, passing "COMM" as *lpszDeviceClass* causes the provider to return an icon related specifically to the Comm device functions of the service provider. The parameters "tapi/phone", "", or NULL may be used to request the icon for the phone service.

For applications using an API version less than 0x00020000, if the provider does not return an icon (whether because the given device class is invalid or the provider does not support icons), TAPI substitutes a generic Win32 Telephony phone device icon. For applications using API version 0x00020000 or greater, TAPI substitutes the default phone icon *only* if the *lpszDeviceClass* parameter is

"tapi/phone", "" or NULL. For any other device class, if the given device class is not valid or the provider does not support icons for the class, **phoneGetIcon** returns PHONEERR_INVALDEVICECLASS.

See Also

phoneGetID

phoneGetID Overview

Overview

Overview

The **phoneGetID** function returns a device ID for the given device class associated with the specified phone device.

```
LONG phoneGetID(

  HPHONE hPhone,
  LPVARSTRING lpDeviceID,
  LPCSTR lpszDeviceClass
);
```

Parameters

hPhone

A handle to an open phone device.

lpDeviceID

A pointer to a data structure of type **VARSTRING** where the device ID is returned. Upon successful completion of the request, this location is filled with the device ID. The format of the returned information depends on the method used by the device class (API) for naming devices.

lpszDeviceClass

A pointer to a NULL-terminated string that specifies the device class of the device whose ID is requested. Valid device class strings are those used in the SYSTEM.INI section to identify device classes.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALDEVICECLASS, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONFAILED, PHONEERR_STRUCTURETOOSMALL, PHONEERR_OPERATIONUNAVAIL.

Remarks

The **phoneGetID** function can be used to retrieve a phone device ID given a phone handle. It can also be used to obtain the device ID of the media device (for device classes such as COM, wave, MIDI, phone, line, or NDIS) associated with the opened phone device. The names of these device class are not case sensitive. This ID can then be used with the appropriate media API to select the corresponding device.

See Device Classes in TAPI for device class names.

A vendor that defines a device-specific media mode also needs to define the corresponding device-specific (proprietary) API to manage devices of the media mode. To avoid collisions on device class names assigned independently by different vendors, a vendor should select a name that uniquely identifies both the vendor and, following it, the media type. For example: "intel/video".

See Also

VARSTRING

phoneGetLamp Overview

Overview

Overview

The **phoneGetLamp** function returns the current lamp mode of the specified lamp.

```
LONG phoneGetLamp(

  HPHONE hPhone,
  DWORD dwButtonLampID,
  LPDWORD lpdwLampMode
);
```

Parameters

hPhone

A handle to the open phone device.

dwButtonLampID

The ID of the lamp to be queried.

lpdwLampMode

A pointer to a memory location that will hold the lamp mode status of the given lamp. Note that *lpdwLampMode* can have at most one bit set. This parameter uses the following PHONELAMPMODE_ constants:

PHONELAMPMODE_BROKENFLUTTER

Broken flutter is the superposition of flash and flutter.

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 is winking.

PHONELAMPMODE_UNKNOWN

The lamp mode is currently unknown.

PHONELAMPMODE_DUMMY

This value is used to describe a button/lamp position that has no corresponding lamp.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible

return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALBUTTONLAMPID, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPOINTER, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPHONESTATE, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONUNAVAIL.

Remarks

Phone sets that have multiple lamps per button should be modeled using multiple button/lamps pairs. Each extra button/lamp pair should use a DUMMY button.

phoneGetMessage Overview

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).

```
LONG phoneGetMessage(

  HPHONEAPP hPhoneApp,
  LPPHONEMESSAGE lpMessage,
  DWORD dwTimeout
);
```

Parameters

hPhoneApp

The handle returned by **phoneInitializeEx**. The application must have set the PHONEINITIALIZEEXOPTION_USEEVENT option in the **dwOptions** field of the **PHONEINITIALIZEEXPARAMS** structure.

lpMessage

A pointer to a **PHONEMESSAGE** structure. Upon successful return from this function, the structure will contain the next message which 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's time-out interval never elapses.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALAPPHANDLE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPOINTER, PHONEERR_NOMEM.

Remarks

If this function has been called with a non-zero timeout and the application calls **phoneShutdown** on another thread, this function will return immediately with PHONEERR_INVALAPPHANDLE.

If the timeout expires (or was zero) and no message could be fetched from the queue, the function returns with the error PHONEERR_OPERATIONFAILED.

See Also

PHONEINITIALIZEEXPARAMS, **PHONEMESSAGE**, **phoneShutdown**

phoneGetRing Overview

Overview

Overview

The **phoneGetRing** function enables an application to query the specified open phone device as to its current ring mode.

```
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.

lpdwVolume

The volume level with which the phone is ringing. This is a number in the range 0x00000000 (silence) to 0x0000FFFF (maximum volume). The actual granularity and quantization of volume settings in this range are service-provider specific.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPHONESTATE, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPOINTER, PHONEERR_OPERATIONFAILED, PHONEERR_OPERATIONUNAVAIL, PHONEERR_UNINITIALIZED.

Remarks

The service provider defines the actual audible ringing patterns corresponding to each of phone's ring modes.

phoneGetStatus Overview

Overview

Overview

The **phoneGetStatus** function enables an application to query the specified open phone device for its overall status.

```
LONG phoneGetStatus(

  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's status.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_OPERATIONFAILED, PHONEERR_STRUCTURETOOSMALL, PHONEERR_OPERATIONUNAVAIL, PHONEERR_UNINITIALIZED.

Remarks

An application can use this function to determine the current state of an open phone device. The status information describes information about the phone device's hookswitch devices, ringer, volume, display, and lamps of the open phone.

See Also

PHONESTATUS

phoneGetStatusMessages Overview

Overview

Overview

The **phoneGetStatusMessages** function returns which phone-state changes on the specified phone device will generate a callback to the application.

```
LONG phoneGetStatusMessages(

  HPHONE hPhone,
  LPDWORD lpdwPhoneStates,
  LPDWORD lpdwButtonModes,
  LPDWORD lpdwButtonStates
);
```

Parameters

hPhone

A handle to the open phone device to be monitored.

lpdwPhoneStates

A pointer to a PHONESTATE_ constant. These flags specify the set of phone status changes and events for which the application wishes to receive notification messages. The *lpdwPhoneStates* parameter can return a value with zero, one, or multiple bits set. Monitoring can be individually enabled and disabled for:

PHONESTATE_OTHER

Phone-status items other than those listed below have changed. The application should check the current phone status to determine which items have 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 connecting the phone to the computer is plugged in while TAPI is active.

PHONESTATE_DISCONNECTED

The connection between the phone device and TAPI was just broken. This happens when the wire connecting the phone set to the computer is unplugged while TAPI is active.

PHONESTATE_OWNER

The number of owners for the phone device has changed.

PHONESTATE_MONITORS

The number of monitors for the phone device has changed.

PHONESTATE_DISPLAY

The display of the phone has changed.

PHONESTATE_LAMP

A lamp of the phone has changed.

PHONESTATE_RINGMODE

The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME

The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH

The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME

The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN

The handset's microphone gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH

The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME

The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN

The speakerphone's microphone gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH

The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME

The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN

The headset's microphone gain setting has changed.

PHONESTATE_SUSPEND

The application's use of the phone is temporarily suspended.

PHONESTATE_RESUME

The application's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC

The phone's device-specific information has changed.

PHONESTATE_REINIT

Items have 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 is left NULL for this state change, as it applies to any of the phones in the system.

PHONESTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **PHONECAPS** structure have changed. The application should use **phoneGetDevCaps** to read the updated structure. If a service provider sends a PHONE_STATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will receive PHONE_STATE messages specifying PHONESTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI in order to obtain the updated information.

PHONESTATE_REMOVED

Indicates that the device is being removed from the system by the service provider (most likely through user action, via a control panel or similar utility). A PHONE_STATE message with this value will normally be immediately followed by a PHONE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in PHONEERR_NODEVICE being returned to the application. If a service provider sends a PHONE_STATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

lpdwButtonModes

A pointer to a . These flags specify the set of phone-button modes for which the application wishes to receive notification messages. The *lpdwButtonModes* parameter can return a value with zero, one, or multiple 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 twelve keypad buttons, '0' through '9', '*', and '#'.

PHONEBUTTONMODE_LOCAL

The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY

The button is a "soft" button associated with the phone's display. A phone set can have zero or more display buttons.

lpdwButtonStates

A pointer to a DWORD-sized location that contains flags specifying the set of phone button state changes for which the application wishes to receive notification messages. The *lpdwButtonStates* field can return a value with zero, one or multiple bits 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

Indicates that the up or down state of the button is not known at this time, but may become known at a future time.

PHONEBUTTONSTATE_UNAVAIL

Indicates that the up or down state of the button is not known to the service provider, and will not become known at a future time.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_OPERATIONFAILED, PHONEERR_UNINITIALIZED.

Remarks

An application can use **phoneGetStatusMessages** to query the generation of the corresponding messages. Message generation can be controlled by **phoneSetStatusMessages**. All phone status messages are disabled by default.

See Also

PHONE_CLOSE, PHONE_STATE, **PHONECAPS**, **phoneGetDevCaps**, **phoneSetStatusMessages**

phoneGetVolume Overview

Overview

Overview

The **phoneGetVolume** function returns the volume setting of the specified phone's hookswitch device.

```
LONG phoneGetVolume(

  HPHONE hPhone,
  DWORD dwHookSwitchDev,
  LPDWORD lpdwVolume
);
```

Parameters

hPhone

A handle to the open phone device.

dwHookSwitchDev

A single hookswitch device whose volume level is queried. This parameter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET

This is the phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER

This is the phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET

This is the phone's headset.

lpdwVolume

A pointer to a DWORD-sized location containing the current volume setting of the hookswitch device. *dwVolume* specifies the volume level of the hookswitch device. This is a number in the range 0x00000000 (silence) to 0x0000FFFF (maximum volume). The actual granularity and quantization of volume settings in this range are service-provider specific.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPHONESTATE, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPOINTER, PHONEERR_OPERATIONFAILED, PHONEERR_INVALHOOKSWITCHDEV, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONUNAVAIL.

phoneInitialize Overview

Overview

Overview

The **phoneInitialize** function is obsolete. It continues to be exported by TAPI.DLL and TAPI32.DLL for backward compatibility with applications using API versions 0x00010003 and 0x00010004.

Applications using API version 0x00020000 or greater must use **phoneInitializeEx** instead.

phoneInitializeEx Overview

The **phoneInitializeEx** function initializes the application's use of TAPI for subsequent use of the phone abstraction. It registers the application's specified notification mechanism and returns the number of phone devices available to the application. A phone device is any device that provides an implementation for the phone-prefixed functions in the Telephony API.

```
LONG 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's usage handle for TAPI.

hInstance

The instance handle of the client application or DLL. The application or DLL may pass NULL for this parameter, in which case TAPI will use 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**). This parameter is 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 displayable ASCII characters. If this parameter is not NULL, it contains an application-supplied name of the application. This name is provided in the **PHONESTATUS** structure to indicate, in a user-friendly way, which application has ownership of the phone device. If *lpszFriendlyAppName* is NULL, the application's module file name is 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 is filled with the number of phone devices 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 it is designed to support (for example, the same value it would pass into *dwAPIHighVersion* parameter of **phoneNegotiateAPIVersion**). Artificially high values must not be used; the value must be accurately set (for this release, to 0x00020000). TAPI will translate any newer messages or structures into values or formats supported by the application's version. Upon successful completion of this request, this location is filled with the highest API version supported by TAPI (for this release, 0x00020000), thereby allowing the application to detect and adapt to having been installed on a system with an older version of TAPI.

lpPhoneInitializeExParams

A pointer to a structure of type **PHONEINITIALIZEEXPARAMS** containing additional parameters used to establish the association between the application and TAPI (specifically, the application's selected event notification mechanism and associated parameters).

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALAPPNAME, PHONEERR_OPERATIONFAILED, PHONEERR_INIFILECORRUPT, PHONEERR_INVALPOINTER, PHONEERR_REINIT, PHONEERR_NOMEM, PHONEERR_INVALPARAM.

Remarks

Applications must select one of three mechanisms by which TAPI notifies the application of telephony events: Hidden Window, Event Handle, or Completion Port.

The Hidden Window mechanism is selected by specifying PHONEINITIALIZEEXOPTION_USEHIDDENWINDOW in the **dwOptions** field in the **PHONEINITIALIZEEXPARAMS** structure. In this mechanism (which is the only mechanism available to TAPI 1.*x* applications), TAPI creates a window in the context of the application during the **phoneInitializeEx** function, and subclasses the window so that all messages posted to it are handled by a WNDPROC in TAPI itself. When TAPI has a message to deliver to the application, TAPI posts a message to the hidden window. When the message is received (which can happen only when the application calls the Windows **GetMessage** API), Windows switches the process context to that of the application and invokes the WNDPROC in TAPI. TAPI then delivers the message to the application by calling the **PhoneCallbackProc**, a pointer to which the application provided as a parameter in its call to **phoneInitializeEx** (or **phoneInitialize**, for TAPI 1.3 and 1.4 applications). This mechanism requires the application to have a message queue (which is not desirable for service processes) and to service that queue regularly to avoid delaying processing of telephony events. The hidden window is destroyed by TAPI during the **phoneShutdown** function.

The Event Handle mechanism is selected by specifying PHONEINITIALIZEEXOPTION_USEEVENT in the **dwOptions** field in the **PHONEINITIALIZEEXPARAMS** structure. In this mechanism, TAPI creates an event object on behalf of the application, and returns a handle to the object in the **hEvent** field in **PHONEINITIALIZEEXPARAMS**. The application must not manipulate this event in any manner (for example, must not call **SetEvent**, **ResetEvent**, **CloseHandle**, and so on) or undefined behavior will result; the application may only wait on this event using functions such as **WaitForSingleObject** or **MsgWaitForMultipleObjects**. TAPI will signal this event whenever a telephony event notification is pending for the application; the application must call **phoneGetMessage** to fetch the contents of the message. The event is reset by TAPI when no events are pending. The event handle is closed and the event object destroyed by TAPI during the **phoneShutdown** function. The application is not required to wait on the event handle that is created; the application could choose instead to call **phoneGetMessage** and have it block waiting for a message to be queued.

The Completion Port mechanism is selected by specifying PHONEINITIALIZEEXOPTION_USECOMPLETION PORT in the **dwOptions** field in the **PHONEINITIALIZEEXPARAMS** structure. In this mechanism, whenever a telephony event needs to be sent to the application, TAPI will send it to the application using **PostQueuedCompletionStatus** to the completion port that the application specified in the **hCompletionPort** field in **PHONEINITIALIZEEXPARAMS**, tagged with the completion key that the application specified in the **dwCompletionKey** field in **PHONEINITIALIZEEXPARAMS**. The application must have previously created the completion port using **CreateIoCompletionPort**. The applications retrieves events using **GetQueuedCompletionStatus**. Upon return from **GetQueuedCompletionStatus**, the application will

have the specified **dwCompletionKey** written to the DWORD pointed to by the *lpCompletionKey* parameter, and a pointer to a **PHONEMESSAGE** structure returned to the location pointed to by *lpOverlapped*. After the application has processed the event, it is the application's responsibility to call **LocalFree** to release the memory used to contain the **PHONEMESSAGE** structure. Because the application created the completion port (thereby allowing it to be shared for other purposes), the application must close it; the application must not close the completion port until after calling **phoneShutdown**.

When a multithreaded application is using the Event Handle mechanism and more than one thread is waiting on the handle, or the Completion Port notification mechanism and more than one thread is waiting on the port, it is possible for telephony events to be processed out of sequence. This is not due to the sequence of delivery of events from TAPI, but would be caused by the time slicing of threads or the execution of threads on separate processors.

If PHONEERR_REINIT is returned and TAPI reinitialization has been requested, for example as a result of adding or removing a Telephony service provider, then **phoneInitializeEx** requests are rejected with this error until the last application shuts down its usage of the API (using **phoneShutdown**), at which time the new configuration becomes effective and applications are once again permitted to call **phoneInitializeEx**.

If the PHONEERR_INVALPARAM error value is returned, the specified *hInstance* parameter is invalid.

The application can refer to individual phone devices by using phone device IDs that range from zero to *dwNumDevs* minus one. An application should not assume that these phone devices are capable of any particular TAPI function without first querying their device capabilities by **phoneGetDevCaps**.

For information about the listing of service dependencies, see Service Dependencies.

See Also

phoneCallbackFunc, **phoneGetDevCaps**, **phoneGetMessage**, **PHONEINITIALIZEEXPARAMS**, **PHONEMESSAGE**, **phoneNegotiateAPIVersion**, **phoneShutdown**, **PHONESTATUS**

phoneCallbackFunc

The **phoneCallbackFunc** function is a placeholder for the application-supplied function name.

```
VOID FAR PASCAL phoneCallbackFunc(

  HANDLE hDevice,
  DWORD dwMsg,
  DWORD dwCallbackInstance,
  DWORD dwParam1,
  DWORD dwParam2,
  DWORD dwParam3
);
```

Parameters

hDevice

A handle to a phone device associated with the callback.

dwMsg

A line or call device message.

dwCallbackInstance

Callback instance data passed back to the application in the callback. This DWORD is not interpreted by TAPI.

dwParam1

A parameter for the message.

dwParam2

A parameter for the message.

dwParam3

A parameter for the message.

Remarks

For more information about the parameters passed to this callback function, see Messages later in this section.

All callbacks occur in the application's context. The callback function must reside in a dynamic-link library (DLL) or application module and be exported in the module-definition file.

phoneNegotiateAPIVersion Overview

Overview

Overview

The **phoneNegotiateAPIVersion** allows an application to negotiate an API version to use for the specified phone device.

```
LONG phoneNegotiateAPIVersion(

  HPHONEAPP hPhoneApp,
  DWORD dwDeviceID,
  DWORD dwAPILowVersion,
  DWORD dwAPIHighVersion,
  LPDWORD lpdwAPIVersion,
  LPPHONEEXTENSIONID lpExtensionID
);
```

Parameters

hPhoneApp

The handle to the application's registration with TAPI.

dwDeviceID

The phone device to be queried.

dwAPILowVersion

The least recent API version the application is compliant with. The high-order word is the major version number, the low-order word is the minor version number.

dwAPIHighVersion

The most recent API version the application is compliant with. The high-order word is the major version number, the low-order word is the minor version number.

lpdwAPIVersion

A pointer to a DWORD-sized location that contains the API version number that was negotiated. If negotiation is successful, this number will be in the range *dwAPILowVersion* and *dwAPIHighVersion*.

lpExtensionID

A pointer to a structure of type **PHONEEXTENSIONID**. If the service provider for the specified *dwDeviceID* supports provider-specific extensions, this structure is filled with the extension ID of these extensions when negotiation is successful. 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 is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALAPPHANDLE, PHONEERR_OPERATIONFAILED, PHONEERR_BADDEVICEID, PHONEERR_OPERATIONUNAVAIL, PHONEERR_NODRIVER, PHONEERR_NOMEM, PHONEERR_INVALPOINTER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INCOMPATIBLEAPIVERSION, PHONEERR_UNINITIALIZED, PHONEERR_NODEVICE.

Remarks

The **phoneNegotiateAPIVersion** function is used to negotiate the API version number to use with the specified phone device. It returns the extension ID supported by the phone device; zeros if no extensions are provided.

If the application wants to use the extensions defined by the returned Extension ID, it must call **phoneNegotiateExtVersion** to negotiate the extension version to use.

Use **phoneInitializeEx** to determine the number of phone devices present in the system. The device ID specified by *dwDeviceID* varies from zero to one less than the number of phone devices present.

The API version number negotiated is that under which TAPI can operate. If version ranges do not overlap, the application, API, or service-provider versions are incompatible and an error is returned.

See Also

PHONEEXTENSIONID, **phoneInitializeEx**, **phoneNegotiateExtVersion**

phoneNegotiateExtVersion Overview

Overview

Overview

The **phoneNegotiateExtVersion** function allows an application to negotiate an extension version to use with the specified phone device. This operation need not be called if the application does not support extensions.

```
LONG phoneNegotiateExtVersion(

  HPHONEAPP hPhoneApp,
  DWORD dwDeviceID,
  DWORD dwAPIVersion,
  DWORD dwExtLowVersion,
  DWORD dwExtHighVersion,
  LPDWORD lpdwExtVersion
);
```

Parameters

hPhoneApp

The handle to the application's registration with TAPI.

dwDeviceID

The phone device to be queried.

dwAPIVersion

The API version number that was negotiated for the specified phone device using **phoneNegotiateAPIVersion**.

dwExtLowVersion

The least recent extension version of the Extension ID returned by **phoneNegotiateAPIVersion** that the application is compliant with. The high-order word is the major version number; the low-order word is the minor version number.

dwExtHighVersion

The most recent extension version of the Extension ID returned by **phoneNegotiateAPIVersion** that the application is compliant with. The high-order word is the major version number; the low-order word is the minor version number.

lpdwExtVersion

A pointer to a DWORD-sized location that contains the extension version number that was negotiated. If negotiation is successful, this number will be in the range *dwExtLowVersion* and *dwExtHighVersion*.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALAPPHANDLE, PHONEERR_OPERATIONFAILED, PHONEERR_BADDEVICEID, PHONEERR_OPERATIONUNAVAIL, PHONEERR_NODRIVER, PHONEERR_NOMEM, PHONEERR_INCOMPATIBLEAPIVERSION, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INCOMPATIBLEEXTVERSION, PHONEERR_UNINITIALIZED,

PHONEERR_INVALPOINTER, PHONEERR_NODEVICE.

Remarks

The function **phoneNegotiateAPIVersion** is used to negotiate the API version number to use. It returns the extension ID supported by the phone device, or zeros if no extensions are provided.

If the application wants to use the extensions defined by the returned Extension ID, it must call **phoneNegotiateExtVersion** to negotiate the extension version to use.

Use **phoneInitializeEx** to determine the number of phone devices present in the system. The device ID specified by *dwDeviceID* varies from zero to one less than the number of phone devices present.

The extension version number negotiated is that under which the application and service provider must both operate. If version ranges do not overlap, the application and service-provider versions are incompatible and an error is returned.

See Also

phoneInitializeEx, **phoneNegotiateAPIVersion**

phoneOpen Overview

Overview

Overview

The **phoneOpen** function opens the specified phone device. A phone device can be opened using either owner privilege or monitor privilege. An application that opens the phone with owner privilege can control the phone's lamps, display, ringer, and hookswitch or hookswitches. An application that opens the phone device with monitor privilege is notified only about events that occur at the phone, such as hookswitch changes or button presses. Ownership of a phone device is exclusive. In other words, only one application can have a phone device opened with owner privilege at a time. The phone device can, however, be opened multiple times with monitor privilege.

```
LONG phoneOpen(

  HPHONEAPP hPhoneApp,
  DWORD dwDeviceID,
  LPHPHONE lphPhone,
  DWORD dwAPIVersion,
  DWORD dwExtVersion,
  DWORD dwCallbackInstance,
  DWORD dwPrivilege
);
```

Parameters

hPhoneApp

A handle to the application's registration with TAPI.

dwDeviceID

The phone device to be opened.

lphPhone

A pointer to an HPHONE handle, which 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 have agreed to operate. This number is obtained 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. This number is obtained from **phoneNegotiateExtVersion**.

dwCallbackInstance

User instance data passed back to the application with each message. This parameter is not interpreted by the Telephony API.

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 is informed about events and state changes occurring 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 is allowed to 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.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_ALLOCATED, PHONEERR_NODRIVER, PHONEERR_BADDEVICEID, PHONEERR_NOMEM, PHONEERR_INCOMPATIBLEAPIVERSION, PHONEERR_OPERATIONFAILED, PHONEERR_INCOMPATIBLEEXTVERSION, PHONEERR_OPERATIONUNAVAIL, PHONEERR_INVALAPPHANDLE, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPOINTER, PHONEERR_UNINITIALIZED, PHONEERR_INVALPRIVILEGE, PHONEERR_REINIT, PHONEERR_INUSE, PHONEERR_NODEVICE, PHONEERR_INIFILECORRUPT.

Remarks

When opening a phone device with monitor privileges, the application is sent messages when events occur that change the status of the phone. Messages sent to the application include PHONE_BUTTON and PHONE_STATE. The latter provides an indication of the phone's status item that has changed.

When opening a phone with owner privilege, the phone device can be manipulated in ways that affect the state of the phone device. An application should only open a phone using owner privilege if it actively wants to manipulate the phone device, and it should close the phone device when done to allow other applications to control the phone.

When an application opens a phone device, it must specify the negotiated API version and, if it wants to use the phone's extensions, the phone's device-specific extension version. This version number should have been obtained with the function **phoneNegotiateAPIVersion** and **phoneNegotiateExtVersion**. Version numbering allows the mix and match of different application versions with different API versions and service-provider versions.

See Also

PHONE_BUTTON, PHONE_STATE, **phoneNegotiateAPIVersion**, **phoneNegotiateExtVersion**

phoneSetButtonInfo Overview

Overview

Overview

The **phoneSetButtonInfo** function sets information about the specified button on the specified phone.

```
LONG phoneSetButtonInfo(

  HPHONE hPhone,
  DWORD dwButtonLampID,
  LPPHONEBUTTONINFO const lpButtonInfo
);
```

Parameters

hPhone

A handle to the open phone device. The application must be the owner of the phone device.

dwButtonLampID

A button on the phone device.

lpButtonInfo

A pointer to a variably sized structure of type **PHONEBUTTONINFO**. This data structure describes the mode, the function, and provides additional descriptive text about the button.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALBUTTONLAMPID, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPHONEHANDLE, PHONEERR_STRUCTURETOOSMALL, PHONEERR_INVALPOINTER, PHONEERR_UNINITIALIZED, PHONEERR_NOTOWNER, PHONEERR_NOMEM, PHONEERR_OPERATIONUNAVAIL, PHONEERR_RESOURCEUNAVAIL.

See Also

PHONE_REPLY, **PHONEBUTTONINFO**

phoneSetData Overview

Overview

Overview

The **phoneSetData** function downloads the information in the specified buffer to the opened phone device at the selected data ID.

```
LONG phoneSetData(

  HPHONE hPhone,
  DWORD dwDataID,
  LPVOID const lpData,
  DWORD dwSize
);
```

Parameters

hPhone

A handle to the open phone device. The application must be the owner of the phone.

dwDataID

Where in the phone device the buffer is to be downloaded.

lpData

A pointer to the memory location where the data is to be downloaded from.

dwSize

The size of the buffer in bytes.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_OPERATIONUNAVAIL, PHONEERR_NOTOWNER, PHONEERR_NOMEM, PHONEERR_INVALDATAID, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPHONESTATE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPOINTER, PHONEERR_UNINITIALIZED.

Remarks

The **phoneSetData** function downloads a maximum of *dwSize* bytes from *lpData* to the phone device. The format of the data, its meaning to the phone device, and the meaning of the data ID are service-provider specific. The data in the buffer or the selection of a data ID may act as commands to the phone device.

See Also

PHONE_REPLY

phoneSetDisplay Overview

Overview

Overview

The **phoneSetDisplay** function causes the specified string to be displayed on the specified open phone device.

```
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 on the display where the new text is to be displayed.

dwColumn

The column position on the display where the new text is to be displayed.

lpsDisplay

A pointer to the memory location where the display content is stored. The display information must have the format specified in the **dwStringFormat** field of the phone's device capabilities.

dwSize

The size in bytes of the information pointed to by *lpsDisplay*.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_OPERATIONUNAVAIL, PHONEERR_NOTOWNER, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPHONESTATE, PHONEERR_UNINITIALIZED, PHONEERR_INVALPOINTER, PHONEERR_NOMEM, PHONEERR_INVALPARAM, PHONEERR_RESOURCEUNAVAIL.

Remarks

The specified display information is written to the phone's display, starting at the specified positions. This operation overwrites previously displayed information. If the amount of information exceeds the size of the display, the information will be truncated. The amount of information that can be displayed is at most (**dwNumRows** * **dwNumColumns**) elements in size. **dwNumRows** and **dwNumColumns** are available in the **PHONECAPS** structure, which is returned by **phoneGetDevCaps**; they are zero-based.

See Also

PHONE_REPLY, **PHONECAPS**, **phoneGetDevCaps**

phoneSetGain Overview

Overview

Overview

The **phoneSetGain** function sets the gain of the microphone of the specified hookswitch device to the specified gain level.

```
LONG phoneSetGain(

  HPHONE hPhone,
  DWORD dwHookSwitchDev,
  DWORD dwGain
);
```

Parameters

hPhone

A handle to the open phone device. The application must be the owner of the phone.

dwHookSwitchDev

The hookswitch device whose microphone's gain is to be set. The *dwHookswitchDev* parameter can only have a single bit set. This parameter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET

The phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER

The phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET

The phone's headset.

dwGain

A pointer to a DWORD-sized location containing the current gain setting of the device. The *dwGain* parameter specifies the gain level of the hookswitch device. This is a number in the range 0x00000000 (silence) to 0x0000FFFF (maximum volume). The actual granularity and quantization of gain settings in this range are service-provider specific. A value for *dwGain* that is out of range is set to the nearest value in the range.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_NOTOWNER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPHONESTATE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALHOOKSWITCHDEV, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONUNAVAIL.

See Also

PHONE_REPLY

phoneSetHookSwitch Overview

Overview

Overview

The **phoneSetHookSwitch** function sets the hook state of the specified open phone's hookswitch devices to the specified mode. Only the hookswitch state of the hookswitch devices listed is affected.

```
LONG phoneSetHookSwitch(

  HPHONE hPhone,
  DWORD dwHookSwitchDevs,
  DWORD dwHookSwitchMode
);
```

Parameters

hPhone

A handle to the open phone device. The application must be the owner of the phone.

dwHookSwitchDevs

The device or devices whose hookswitch mode is to be set. This parameter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET

The phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER

The phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET

The phone's headset.

dwHookSwitchMode

The hookswitch mode to set. The *dwHookSwitchMode* parameter can have only a single bit set. This parameter uses the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK

The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC

The device's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER

The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER

The device's microphone and speaker are both active.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number

if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_OPERATIONUNAVAIL, PHONEERR_NOTOWNER, PHONEERR_NOMEM, PHONEERR_INVALHOOKSWITCHDEV, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALHOOKSWITCHMODE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALPHONESTATE, PHONEERR_UNINITIALIZED.

Remarks

The hookswitch mode is the same for all devices specified. If different settings are desired, this function can be invoked multiple times with a different set of parameters. A PHONE_STATE message is sent to the application after the hookswitch state has changed.

See Also

PHONE_REPLY, PHONE_STATE

phoneSetLamp Overview

Overview

Overview

The **phoneSetLamp** function causes the specified lamp to be lit on the specified open phone device in the specified lamp mode.

```
LONG phoneSetLamp(

  HPHONE hPhone,
  DWORD dwButtonLampID,
  DWORD dwLampMode
);
```

Parameters

hPhone

A handle to the open phone device. The application must be the owner of the phone.

dwButtonLampID

The button whose lamp is to be lit.

dwLampMode

How the lamp is to be lit. The *dwLampMode* parameter can have only a single bit set. This parameter uses the following PHONELAMPMODE_ constants:

PHONELAMPMODE_BROKENFLUTTER

Broken flutter is the superposition of flash and flutter.

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 is winking.

PHONELAMPMODE_DUMMY

This value is used to describe a button/lamp position that has no corresponding lamp.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_OPERATIONUNAVAIL, PHONEERR_NOTOWNER, PHONEERR_NOMEM, PHONEERR_INVALBUTTONLAMPID, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPHONESTATE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALLAMPMODE, PHONEERR_UNINITIALIZED.

See Also

PHONE_REPLY

phoneSetRing Overview

Overview

Overview

The **phoneSetRing** function rings the specified open phone device using the specified ring mode and volume.

```
LONG phoneSetRing(

  HPHONE hPhone,
  DWORD dwRingMode,
  DWORD dwVolume
);
```

Parameters

hPhone

A handle to the open phone device. The application must be the owner of the phone device.

dwRingMode

The ringing pattern with which to ring the phone. This parameter must be within the range of 0 to the value of the **dwNumRingModes** field in the **PHONECAPS** structure. If **dwNumRingModes** is 0, the ring mode of the phone cannot be controlled; if **dwNumRingModes** is 1, a value of 0 for *dwRingMode* indicates that the phone should not be rung (silence), and other values from 1 to **dwNumRingModes** are valid ring modes for the phone device.

dwVolume

The volume level with which the phone is ringing. This is a number in the range 0x00000000 (silence) to 0x0000FFFF (maximum volume). The actual granularity and quantization of volume settings in this range are service-provider specific. A value for *dwVolume* that is out of range is set to the nearest value in the range.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_NOTOWNER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPHONESTATE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALRINGMODE, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONUNAVAIL.

Remarks

The service provider defines the actual audible ringing patterns corresponding to each of the phone's ring modes.

See Also

PHONE_REPLY, **PHONECAPS**

phoneSetStatusMessages Overview

Overview

Overview

The **phoneSetStatusMessages** function enables an application to monitor the specified phone device for selected status events.

```
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 wishes to 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 listed below have changed. The application should check the current phone status to determine which items have 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 connecting the phone to the PC is plugged in while TAPI is active.

PHONESTATE_DISCONNECTED

The connection between the phone device and TAPI was just broken. This happens when the wire connecting the phone set to the PC is unplugged while TAPI is active.

PHONESTATE_OWNER

The number of owners for the phone device has changed.

PHONESTATE_MONITORS

The number of monitors for the phone device has changed.

PHONESTATE_DISPLAY

The display of the phone has changed.

PHONESTATE_LAMP

A lamp of the phone has changed.

PHONESTATE_RINGMODE

The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME

The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH

The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME

The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN

The handset's microphone gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH

The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME

The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN

The speakerphone's microphone gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH

The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME

The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN

The headset's microphone gain setting has changed.

PHONESTATE_SUSPEND

The application's use of the phone is temporarily suspended.

PHONESTATE_RESUME

The application's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC

The phone's device-specific information has changed.

PHONESTATE_REINIT

Items have 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 are denied until applications have shut down their usage of TAPI. The *hDevice* parameter of the PHONE_STATE message is left NULL for this state change, because it applies to any of the lines in the system. Because of the critical nature of PHONESTATE_REINIT, such messages cannot be masked, so the setting of this bit is ignored and the messages are always delivered to the application.

PHONESTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **PHONECAPS** structure have changed. The application should use **phoneGetDevCaps** to read the updated structure. If a service provider sends a PHONE_STATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x0001004 or above; applications negotiating a previous API version will receive PHONE_STATE messages specifying PHONESTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI in order to obtain the updated information.

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). A PHONE_STATE message with this value will normally be immediately followed by a PHONE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in PHONEERR_NODEVICE being returned to the application. If a service provider sends a PHONE_STATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

dwButtonModes

The set of phone-button modes for which the application wishes to 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 twelve keypad buttons, '0' through '9', '*', and '#'.

PHONEBUTTONMODE_LOCAL

The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY

The button is a "soft" button associated with the phone's display. A phone set can have zero or more display buttons.

dwButtonStates

The set of phone-button state changes for which the application wants to receive notification messages. If the *dwButtonModes* parameter is 0, *dwButtonStates* is ignored. If *dwButtonModes* has one or more bits set, this parameter must also 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

Indicates that the up or down state of the button is not known at this time, but may become known at a future time.

PHONEBUTTONSTATE_UNAVAIL

Indicates that the up or down state of the button is not known to the service provider, and will not become known at a future time.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_INVALPHONESTATE, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALBUTTONMODE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALBUTTONSTATE, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONUNAVAIL.

Remarks

An application can use the **phoneSetStatusMessages** function to enable or disable the generation of the corresponding messages. All phone status messages are disabled by default.

See Also

PHONE_CLOSE, PHONE_STATE, **PHONECAPS**, **phoneGetDevCaps**, **phoneInitialize**, **phoneInitializeEx**, **phoneOpen**

phoneSetVolume Overview

Overview

Overview

The **phoneSetVolume** sets the volume of the speaker component of the specified hookswitch device to the specified level.

```
LONG phoneSetVolume(

  HPHONE hPhone,
  DWORD dwHookSwitchDev,
  DWORD dwVolume
);
```

Parameters

hPhone

A handle to the open phone device. The application must be the owner of the phone.

dwHookSwitchDev

The hookswitch device whose speaker's volume is to be set.

PHONEHOOKSWITCHDEV_HANDSET

The phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER

The phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET

The phone's headset.

dwVolume

The current volume setting of the device. The *dwVolume* parameter specifies the volume level of the hookswitch device. This is a number in the range 0x00000000 (silence) to 0x0000FFFF (maximum volume). The actual granularity and quantization of volume settings in this range are service-provider specific. A value for *dwVolume* that is out of range is set to the nearest value in the range.

Return Values

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding PHONE_REPLY message is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALPHONEHANDLE, PHONEERR_NOMEM, PHONEERR_NOTOWNER, PHONEERR_RESOURCEUNAVAIL, PHONEERR_INVALPHONESTATE, PHONEERR_OPERATIONFAILED, PHONEERR_INVALHOOKSWITCHDEV, PHONEERR_UNINITIALIZED, PHONEERR_OPERATIONUNAVAIL.

See Also

PHONE_REPLY

phoneShutdown Overview

Overview

Overview

The **phoneShutdown** function shuts down the application's usage of TAPI's phone abstraction.

```
LONG phoneShutdown(

  HPHONEAPP hPhoneApp
);
```

Parameters

hPhoneApp

The application's usage handle for TAPI.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

PHONEERR_INVALAPPHANDLE, PHONEERR_NOMEM, PHONEERR_UNINITIALIZED, PHONEERR_RESOURCEUNAVAIL.

Remarks

If this function is called when the application has open phone devices. These devices will be closed.

Assisted Telephony Functions

The functions in this section are used by Assisted Telephony.

tapiRequestMakeCall Overview

Overview

Overview

The **tapiRequestMakeCall** function requests the establishment of a voice call. A call-manager application is responsible for establishing the call on behalf of the requesting application, which is then controlled by the user's call-manager application.

```
LONG tapiRequestMakeCall(

  LPCSTR lpszDestAddress,
  LPCSTR lpszAppName,
  LPCSTR lpszCalledParty,
  LPCSTR lpszComment
);
```

Parameters

lpszDestAddress

A pointer to a memory location where the NULL-terminated destination address of the call request is located. The address can use the canonical address format (address formats are discussed in Line Devices Overview) or the dialable address format. Validity of the specified address is not checked by this operation. The maximum length of the address is TAPIMAXDESTADDRESSSIZE characters, which includes the NULL terminator.

lpszAppName

A pointer to a memory location where the ASCII NULL-terminated user-friendly application name of the call request is located. This pointer may be left NULL if the application does not wish to supply an application name. The maximum length of the address is TAPIMAXAPPNAMESIZE characters, which includes the NULL terminator. Longer strings are truncated.

lpszCalledParty

A pointer to a memory location where the ASCII NULL-terminated called party name for the called party of the call is located. This pointer may be left NULL if the application does not wish to supply this information. The maximum length of the string is TAPIMAXCALLEDPARTYSIZE characters, which includes the NULL terminator. Longer strings are truncated.

lpszComment

A pointer to a memory location where the ASCII NULL-terminated comment about the call is located. This pointer may be left NULL if the application does not wish to supply a comment. The maximum length of the address is TAPIMAXCOMMENTSIZE characters, which includes the NULL terminator. Longer strings are truncated.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible error return value are:

TAPIERR_NOREQUESTRECIPIENT, TAPIERR_INVALDESTADDRESS, TAPIERR_REQUESTQUEUEFULL, TAPIERR_INVALPOINTER.

Remarks

A telephony-enabled application can request that a call be placed on its behalf by invoking

tapiRequestMakeCall, providing only the destination address for the call. This request is forwarded to the user's call-control application, which places the call on behalf of the original application. A default call-control application is provided as part of Win32 Telephony. Users can replace this with a call-control application of their choice.

Invoking **tapiRequestMakeCall** when no call control application is running returns the TAPIERR_NOREQUESTRECIPIENT error indication. If the call control application is not running, TAPI will attempt to launch the highest-priority call control application (which is listed for RequestMakeCall in the registry). Invoking this function when the Assisted TAPI request queue is full returns the TAPIERR_REQUESTQUEUEFULL error.

tapiRequestMediaCall Overview

Overview

The **tapiRequestMediaCall** function is nonfunctional in Win32-based applications and obsolete for all classes of Windows-based applications. It should not be used.

tapiGetLocationInfo Overview

Overview

Overview

The **tapiGetLocationInfo** function returns the country code and city (area) code which the user has set in the current location parameters in the Telephony control panel. The application may use this information to assist the user in forming proper canonical telephone numbers, such as by offering these as defaults when new numbers are entered in a phone book entry or database record.

```
LONG tapiGetLocationInfo(

  LPCSTR lpszCountryCode,
  LPCSTR lpszCityCode
);
```

Parameters

lpszCountryCode

A pointer to a memory location where a NULL-terminated ASCII string specifying the country code for the current location is to be returned. The application should allocate at least 8 bytes of storage at this location to hold the string (TAPI will not return more than 8 bytes, including the terminating NULL). An empty string (\0) will be returned if the country code has not been set for the current location.

lpszCityCode

A pointer to a memory location where a NULL-terminated ASCII string specifying the city (area) code for the current location is to be returned. The application should allocate at least 8 bytes of storage at this location to hold the string (TAPI will not return more than 8 bytes, including the terminating NULL). An empty string (\0) will be returned if the city code has not been set for the current location.

Return Values

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are TAPIERR_REQUESTFAILED.

tapiRequestDrop Overview

Overview

The **tapiRequestDrop** function is nonfunctional in Win32 -based applications and obsolete for all classes of Windows-based applications. It should not be used.

Messages

This section contains the reference for line device, phone device, and assisted telephony messages.

Line Device Messages

This section contains a list of the messages in the Telephony API. Messages are used to notify the application of the occurrence of asynchronous events. All of these messages are sent to the application via the message notification mechanism the applicaiton specified in **lineInitializeEx**.

The message always contains a handle to the relevant object (phone, line, or call). The application can determine the type of the handle from the message type.

Certain messages are used to notify the application about a change in an object's status. These messages provide the object handle and give an indication of which status item has changed. The application can call the appropriate "get status" function of the object to obtain the object's full status.

When an event occurs, messages may be sent to zero, one, or more applications. The target applications for a message are determined by a number of different factors including the meaning of the message, the application's privilege to the object, whether or not the application initiated the request for which the message is a direct result, and the message masking that has been set by the application. Note the following points about messages:

- Asynchronous reply messages are only sent to the application that originated the request and cannot be masked.
- Messages that signal the completion of digit or tone generation or the gathering of digits are only sent to the application that requested the digit or tone generation.
- Messages that indicate line or address state changes are sent to all applications that have opened the line, so long as the message has been enabled via **lineSetStatusMessages**.
- Messages that indicate call state and call information changes are sent to all applications that have a handle to the call.
- Messages that signal a digit detection, tone detection, or media mode detection are sent to the application(s) that requested monitoring of that event.

LINE_ADDRESSSTATE Overview

The LINE_ADDRESSSTATE message is 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.

```
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 opening the line.

dwParam1

The address ID 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 listed below have changed. The application should check the current address status to determine which items have changed.

LINEADDRESSSTATE_DEVSPECIFIC

The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO

The address has changed to idle (it is now in use by zero stations).

LINEADDRESSSTATE_INUSEONE

The address has 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 has 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 is the result of 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 has changed, including the number of rings for determining a no answer condition. The application should check the address status to determine details about the address's current forwarding status.

LINEADDRESSSTATE_TERMINALS

The terminal settings for the address have changed.

LINEADDRESSSTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINEADDRESSCAPS** structure for the address have changed. The application should use **lineGetAddressCaps** to read the updated structure. Applications which support API versions less than 0x00010004 will receive a LINEDEVSTATE_REINIT message, requiring them to shutdown and reinitialize their connection to TAPI in order to obtain the updated information.

dwParam3

Unused.

Return Values

No return value.

Remarks

The LINE_ADDRESSSTATE message is sent to any application that has opened the line device and that has enabled this message. The sending of this message for the various status items can be controlled and queried using **lineGetStatusMessages** and **lineSetStatusMessages**. By default, address status reporting is disabled.

See Also

LINEADDRESSCAPS, **lineGetAddressCaps**, **lineGetAddressStatus**, **lineGetStatusMessages**, **lineSetStatusMessages**

LINE_AGENTSPECIFIC Overview

The LINE_AGENTSPECIFIC message is sent when the status of an ACD agent changes on a line the application currently has open. The application can invoke **lineGetAgentStatus** to determine the current status of the agent.

```
LINE_AGENTSPECIFIC
    dwDevice = (DWORD) hLine;
    dwCallbackInstance = (DWORD) dwInstanceData;
    dwParam1 = (DWORD) dwAgentExtensionIDIndex;
    dwParam2 = (DWORD) dwHandlerSpecific1;
    dwParam3 = (DWORD) dwHandlerSpecific2;
```

Parameters

dwDevice

The application's handle to the line device.

dwCallbackInstance

The callback instance supplied when opening the call's line.

dwParam1

The index into the array of handler extension IDs in **LINEAGENTCAPS** structure of the handler extension with which the message is associated:

dwParam2

Specific to the handler extension. Generally, this value will be used to cause the application to invoke a **lineAgentSpecific** function to fetch further details of the message.

dwParam3

Specific to the handler extension.

Return Values

No return value.

Remarks

The LINE_AGENTSPECIFIC message is not sent to applications which support older versions of TAPI.

See Also

LINEAGENTCAPS, **lineAgentSpecific**, **lineGetAgentStatus**

LINE_AGENTSTATUS Overview

The LINE_AGENTSTATUS message is sent when the status of an ACD agent changes on a line the application currently has open. The application can invoke **lineGetAgentStatus** to determine the current status of the agent.

```
LINE_AGENTSTATUS
dwDevice = (DWORD) hLine;
    dwCallbackInstance = (DWORD) dwInstanceData;
    dwParam1 = (DWORD) dwAddressID;
    dwParam2 = (DWORD) AgentStatus;
    dwParam3 = (DWORD) AgentStatusDetail;
```

Parameters

dwDevice

The application's handle to the line device on which the agent status has changed.

dwCallbackInstance

The callback instance supplied when opening the call's line.

dwParam1

Identifier of the address on the line on which the agent status changed.

dwParam2

Specifies the agent status that has changed; can be a combination of LINEAGENTSTATUS_ constant values:

dwParam3

If *dwParam2* includes the LINEAGENTSTATUS_STATE bit, then *dwParam3* indicates the new value of **dwState** member in **LINEAGENTSTATUS**. Otherwise, this parameter is set to 0.

Return Values

No return value.

Remarks

The LINE_AGENTSTATUS message is not sent to applications which support older versions of TAPI.

See Also

LINEAGENTSTATUS, **lineGetAgentStatus**

LINE_APPNEWCALL Overview

The LINE_APPNEWCALL message is sent to inform an application when a new call handle has been 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 passed into the function).

```
LINE_APPNEWCALL
dwDevice = (DWORD) hLine;
    dwCallbackInstance = (DWORD) dwInstanceData;
    dwParam1 = (DWORD) dwAddressID;
    dwParam2 = (DWORD) hCall;
    dwParam3 = (DWORD) dwPrivilege;
```

Parameters

dwDevice

The application's handle to the line device on which the call has been created.

dwCallbackInstance

The callback instance supplied when opening the call's line.

dwParam1

Identifier of the address on the line on which the call appears.

dwParam2

The application's handle to the new call.

dwParam3

The applications privilege to the new call (LINECALLPRIVILEGE_OWNER or LINECALLPRIVILEGE_MONITOR).

Return Values

No return value.

Comments and Backward Compatibility

Applications supporting TAPI version 0x00020000 or above are sent a LINE_APPNEWCALL message whenever the application is spontaneously given a handle to a new call. Because the message includes the *hLine* and *dwAddressID* on which the call exists, the application can readily create a *new* call object in the correct context. The LINE_APPNEWCALL message will always be immediately followed by a LINE_CALLSTATE message indicating the initial state of the call.

Older applications (which negotiated an API version prior to 0x00020000) are sent only a LINE_CALLSTATE message, as documented in previous versions of the API. Such applications would create a *new* call object upon receiving a LINE_CALLSTATE message which has *dwParam3* set to a non-zero value and containing a call handle not presently known by the application. The disadvantages are that (a) the application must call **lineGetCallInfo** to determine the *hLine* and *dwAddressID* associated with the call, (b) the application must scan all known call handles to determine that the call is a new call, and (c) it is possible, under certain conditions, for the application to think it is receiving a new call handle when in reality it has just deallocated its handle to the call (for example, the application has just deallocated a call handle, but a LINE_CALLSTATE message giving the application ownership of the call due to a **lineHandoff** from another application was already in the application's TAPI message queue).

See Also

LINE_CALLSTATE, **lineGetCallInfo**, **lineHandoff**

LINE_CALLINFO Overview

The LINE_CALLINFO message is sent when the call information about the specified call has changed. The application can invoke **lineGetCallInfo** to determine the current call information.

```
LINE_CALLINFO
dwDevice = (DWORD) hCall;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) CallInfoState;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

A handle to the call.

dwCallbackInstance

The callback instance supplied when opening the call's line.

dwParam1

The call information item that has changed. Can be a combination of these values:

LINECALLINFOSTATE_OTHER

Information items other than those listed below have changed. The application should check the current call information to determine which items have changed.

LINECALLINFOSTATE_DEVSPECIFIC

The device-specific field of the call-information record has changed.

LINECALLINFOSTATE_BEARERMODE

The bearer mode field of the call-information record has changed.

LINECALLINFOSTATE_RATE

The rate field of the call-information record has changed.

LINECALLINFOSTATE_MEDIAMODE

The media mode field of the call-information record has changed.

LINECALLINFOSTATE_APPSPECIFIC

The application-specific field of the call-information record has changed.

LINECALLINFOSTATE_CALLID

The call ID field of the call-information record has changed.

LINECALLINFOSTATE_RELATEDCALLID

The related call ID field of the call-information record has changed.

LINECALLINFOSTATE_ORIGIN

The origin field of the call-information record has changed.

LINECALLINFOSTATE_REASON

The reason field of the call-information record has changed.

LINECALLINFOSTATE_COMPLETIONID

The completion ID field of the call-information record has changed.

LINECALLINFOSTATE_NUMOWNERINCR

The number of owner fields in the call-information record was increased.

LINECALLINFOSTATE_NUMOWNERDECR

The number of owner fields in the call-information record was decreased.

LINECALLINFOSTATE_NUMMONITORS

The number of monitors fields in the call-information record has changed.

LINECALLINFOSTATE_TRUNK

The trunk field of the call information record has changed.

LINECALLINFOSTATE_CALLERID

One of the callerID-related fields of the call information record has changed.

LINECALLINFOSTATE_CALLEDID

One of the calledID-related fields of the call information record has changed.

LINECALLINFOSTATE_CONNECTEDID

One of the connectedID-related fields of the call information record has changed.

LINECALLINFOSTATE_REDIRECTIONID

One of the redirectionID-related fields of the call information record has changed.

LINECALLINFOSTATE_REDIRECTINGID

One of the redirectingID-related fields of the call information record has changed.

LINECALLINFOSTATE_DISPLAY

The display field of the call information record has changed.

LINECALLINFOSTATE_USERUSERINFO

The user-to-user information of the call information record has changed.

LINECALLINFOSTATE_HIGHLEVELCOMP

The high-level compatibility field of the call information record has changed.

LINECALLINFOSTATE_LOWLEVELCOMP

The low-level compatibility field of the call information record has changed.

LINECALLINFOSTATE_CHARGINGINFO

The charging information of the call information record has changed.

LINECALLINFOSTATE_TERMINAL

The terminal mode information of the call information record has changed.

LINECALLINFOSTATE_DIALPARAMS

The dial parameters of the call information record has changed.

LINECALLINFOSTATE_MONITORMODES

One or more of the digit, tone, or media monitoring fields in the call information record has changed.

dwParam2

Unused.

dwParam3

Unused.

Return Values

No return value.

Remarks

A LINE_CALLINFO message with a *NumOwnersIncr*, *NumOwnersDecr*, and/or *NumMonitorsChanged* indication is sent to applications that already have a handle for the call. This can be the result of another application changing ownership or monitorship to a call with **lineOpen**, **lineClose**, **lineShutdown**, **lineSetCallPrivilege**, **lineGetNewCalls**, and **lineGetConfRelatedCalls**.

These LINE_CALLINFO messages are not sent when a notification of a new call is provided in a LINE_CALLSTATE message, because the call information already reflects the correct number of owners and monitors at the time the LINE_CALLSTATE messages are sent. LINE_CALLINFO messages are also suppressed in the case where a call is offered by TAPI to monitors through the LINECALLSTATE_UNKNOWN mechanism.

Note The application which causes a change in the number of owners or monitors (for example, by invoking **lineDeallocateCall** or **lineSetCallPrivilege**) will not itself receive a message indicating that the change has been done.

No LINE_CALLINFO messages are sent for a call after the call has entered the *idle* state. Specifically, changes in the number of owners and monitors are not reported as applications deallocate their handles for the idle call.

See Also

lineClose, **lineDeallocateCall**, **lineGetCallInfo**, **lineGetConfRelatedCalls**, **lineGetNewCalls**, **lineOpen**, **lineSetCallPrivilege**, **lineShutdown**

LINE_CALLSTATE Overview

The LINE_CALLSTATE message is sent when the status of the specified call has changed. Several such messages will typically be received during the lifetime of a call. Applications are notified of new incoming calls with this message; the new call will be in the *offering* state. The application can use **lineGetCallStatus** to retrieve more detailed information about the current status of the call.

```
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 supplied when opening the call's line.

dwParam1

The new call state. This parameter must be one and only one of the following LINECALLSTATE_ values:

LINECALLSTATE_IDLE

The call is idle—no call actually exists.

LINECALLSTATE_OFFERING

The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the offering state does not automatically alert the user. Alerting is done by the switch instructing the line to ring; it does not affect any call states.

LINECALLSTATE_ACCEPTED

The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also indicates that alerting to both parties has started.

LINECALLSTATE_DIALTONE

The call is receiving 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 being sent to the switch over the call. Note that **lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK

The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY

The call is receiving a busy tone. Busy tone indicates that the call cannot be completed—either a circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO

Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED

The call has been established and the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING

Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD

The call is on hold by the switch.

LINECALLSTATE_CONFERENCED

The call is currently a member of a multiparty conference call.

LINECALLSTATE_ONHOLDPENDCONF

The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENTRANSFER

The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED

The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN

The state of the call is not known. This may be due to limitations of the call-progress detection implementation.

dwParam2

Call-state-dependent information.

If *dwParam1* is LINECALLSTATE_BUSY, *dwParam2* contains details about the busy mode. This parameter uses the following LINEBUSYMODE_ constants:

LINEBUSYMODE_STATION

The busy signal indicates that the called party's station is busy. This is usually signaled by means of a "normal" busy tone.

LINEBUSYMODE_TRUNK

The busy signal indicates that a trunk or circuit is busy. This is usually signaled with a "long" busy tone.

LINEBUSYMODE_UNKNOWN

The busy signal's specific mode is currently unknown, but may become known later.

LINEBUSYMODE_UNAVAIL

The busy signal's specific mode is unavailable and will not become known.

If *dwParam1* is LINECALLSTATE_CONNECTED, *dwParam2* contains details about the connected mode. This parameter uses the following LINECONNECTEDMODE_ constants:

LINECONNECTEDMODE_ACTIVE

Indicates that the call is connected at the current station (the current station is a participant in the call).

LINECONNECTEDMODE_INACTIVE

Indicates that the call is active at one or more other stations, but the current station is not a participant in the call.

If *dwParam1* is LINECALLSTATE_DIALTONE, *dwParam2* contains the details about the dial tone mode. This parameter uses the following LINEDIALTONEMODE_ constants:

LINEDIALTONEMODE_NORMAL

This is a "normal" dial tone, which typically is a continuous tone.

LINEDIALTONEMODE_SPECIAL

This is a special dial tone indicating that a certain condition is currently in effect.

LINEDIALTONEMODE_INTERNAL

This is an internal dial tone, as within a PBX.

LINEDIALTONEMODE_EXTERNAL

This is an external (public network) dial tone.

LINEDIALTONEMODE_UNKNOWN

The dial tone mode is currently unknown, but may become known later.

LINEDIALTONEMODE_UNAVAIL

The dial tone mode is unavailable and will not become known.

If *dwParam1* is LINECALLSTATE_OFFERING, *dwParam2* contains details about the connected mode. This parameter uses the following LINEOFFERINGMODE_ constants:

LINEOFFERINGMODE_ACTIVE

Indicates that the call is alerting at the current station (will be accompanied by LINEDEVSTATE_RINGING messages), and if any application is set up to automatically answer, it may do so.

LINEOFFERINGMODE_INACTIVE

Indicates that the call is being offered at more than one station, but the current station is not alerting (for example, it may be an attendant station where the offering status is advisory, such as blinking a light).

If *dwParam1* is LINECALLSTATE_SPECIALINFO, *dwParam2* contains the details about the special information mode. This parameter uses the following LINESPECIALINFO_ constants:

LINESPECIALINFO_NOCIRCUIT

This special information tone precedes a "no circuit" or emergency announcement (trunk blockage category).

LINESPECIALINFO_CUSTIRREG

This special information tone precedes a vacant number, AIS, Centrex number change and nonworking station, access code not dialed or dialed in error, or manual intercept operator

message (customer irregularity category).

LINESPECIALINFO_REORDER

This special information tone precedes a reorder announcement (equipment irregularity category).

LINESPECIALINFO_UNKNOWN

Specifics about the special information tone are currently unknown but may become known later.

LINESPECIALINFO_UNAVAIL

Specifics about the special information tone are unavailable and will not become known.

If *dwParam1* is LINECALLSTATE_DISCONNECTED, *dwParam2* contains details about the disconnect mode. This parameter uses the following LINEDISCONNECTMODE_ constants:

LINEDISCONNECTMODE_NORMAL

This is a "normal" disconnect request by the remote party, the call was terminated normally.

LINEDISCONNECTMODE_UNKNOWN

The reason for the disconnect request is unknown.

LINEDISCONNECTMODE_REJECT

The remote user has rejected the call.

LINEDISCONNECTMODE_PICKUP

The call was picked up from elsewhere.

LINEDISCONNECTMODE_FORWARDED

The call was forwarded by the switch.

LINEDISCONNECTMODE_BUSY

The remote user's station is busy.

LINEDISCONNECTMODE_NOANSWER

The remote user's station does not answer.

LINEDISCONNECTMODE_NODIALTONE

A dial tone was not detected within a service-provider defined timeout, at a point during dialing when one was expected (such as at a "W" in the dialable string). This can also occur without a service-provider-defined timeout period or without a value specified in the **dwWaitForDialTone** member of the **LINEDIALPARAMS** structure.

LINEDISCONNECTMODE_BADADDRESS

The destination address in invalid.

LINEDISCONNECTMODE_UNREACHABLE

The remote user could not be reached.

LINEDISCONNECTMODE_CONGESTION

The network is congested.

LINEDISCONNECTMODE_INCOMPATIBLE

The remote user's station equipment is incompatible for the type of call requested.

LINEDISCONNECTMODE_UNAVAIL

The reason for the disconnect is unavailable and will not become known later.

If *dwParam1* is LINECALLSTATE_CONFERENCED, *dwParam2* contains the *hConfCall* of the parent call of the conference of which the subject *hCall* is a member. If the call specified in *dwParam2* was not previously considered by the application to be a parent conference call (*hConfCall)*, the application must do so as a result of this message. If the application does not have a handle to the parent call of the conference (because it has previously called **lineDeallocateCall** on that handle) *dwParam2* will be set to *null*.

dwParam3

If zero, this parameter indicates that there has been no change in the application's privilege for the call.

If non-zero, it specifies the application's privilege to the call. This will occur in the following situations: (1) The first time that the application is given a handle to this call; (2) When the application is the target of a call handoff (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.

Return Values

No return value.

Remarks

This message is sent to any application that has a handle for the call. The LINE_CALLSTATE message also notifies applications that monitor calls on a line about the existence and state of outbound calls established by other applications or manually by the user (for example, on an attached phone device). The call state of such calls reflects the actual state of the call, which will not be *offering*. By examining the call state, the application can determine whether the call is an inbound call that needs to be answered or not.

A LINE_CALLSTATE message with an unknown call state may be sent to a monitoring application as the result of a successful **lineMakeCall**, **lineForward**, **lineUnpark**, **lineSetupTransfer**, **linePickup**, **lineSetupConference**, or **linePrepareAddToConference** that has been requested by another application. At the same time that the requesting application is sent a LINE_REPLY (success) for the requested operation, any monitoring applications on the line will be sent the LINE_CALLSTATE (unknown) message. A LINE_CALLSTATE indicating the "real" call state of the newly generated call will be sent (using information provided by the service provider) to the requesting and monitoring applications shortly thereafter.

A LINE_CALLSTATE (unknown) message is sent to monitoring applications only if **lineCompleteTransfer** causes calls to be resolved into a three-way conference.

For backward compatibility, older applications will not be expecting any particular value in *dwParam2* of a LINECALLSTATE_CONFERENCED message. TAPI will therefore pass the parent call *hConfCall* in *dwParam2* regardless of the API version of the application receiving the message. In the case of a conference call initiated by the service provider, the older application will not be aware that the parent call has become a conference call unless it happens to spontaneously examine other information (for example, call **lineGetConfRelatedCalls**)

This message cannot be disabled.

See Also

LINE_REPLY, **lineCompleteTransfer**, **lineDeallocateCall**, **LINEDIALPARAMS**, **lineForward**, **lineGenerateDigits**, **lineGetCallStatus**, **lineGetConfRelatedCalls**, **lineMakeCall**, **linePickup**, **linePrepareAddToConference**, **lineSetupTransfer**, **lineUnpark**

LINE_CLOSE Overview

The LINE_CLOSE message is sent when the specified line device has been forcibly closed. The line device handle or any call handles for calls on the line are no longer valid once this message has been sent.

```
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 supplied when opening the line.

dwParam1

Unused.

dwParam2

Unused.

dwParam3

Unused.

Return Values

No return value.

Remarks

The LINE_CLOSE message is only sent to any application after an open line has been forcibly closed. This may be done to prevent a single application from monopolizing a line device for too long. Whether or not the line can be reopened immediately after a forced close is device-specific.

A line device may also be forcibly closed after the user has modified the configuration of that line or its driver. If the user wants the configuration changes to be effective immediately (as opposed to after the next system restart), and they affect the application's current view of the device (such as a change in device capabilities), then a service provider may forcibly close the line device.

LINE_CREATE Overview

The LINE_CREATE message is sent to inform the application of the creation of a new line device.

```
dwDevice = (DWORD) 0;
    dwCallbackInstance = (DWORD) 0;
    dwParam1 = (DWORD) idDevice;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

Unused.

dwCallbackInstance

Unused.

dwParam1

The *dwDeviceID* of the newly-created device.

dwParam2

Unused.

dwParam3

Unused.

Return Values

No return value.

Remarks

Older applications (which negotiated TAPI version 0x00010003) are sent a LINE_LINEDEVSTATE message specifying LINEDEVSTATE_REINIT, which requires them to shut down their use of the API and call **lineInitialize** again to obtain the new number of devices. Unlike previous versions of TAPI, however, this version does not require *all* applications to shut down before allowing applications to reinitialize; reinitialization can take place immediately when a new device is created (complete shutdown is still required when a service provider is removed from the system).

Applications supporting TAPI version 0x00010004 or above are sent a LINE_CREATE message. This informs them of the existence of the new device and its new device ID. The application can then choose whether or not to attempt working with the new device at its leisure. This message will be sent to all applications supporting this or subsequent versions of the API which have called **lineInitialize** or **lineInitializeEx**, including those that do not have any line devices open at the time.

See Also

LINE_LINEDEVSTATE, **lineInitialize**, **lineInitializeEx**

LINE_DEVSPECIFIC Overview

The LINE_DEVSPECIFIC message is sent to notify the application about device-specific events occurring on a line, address, or call. The meaning of the message and the interpretation of the parameters are device specific.

```
LINE_DEVSPECIFIC
dwDevice = (DWORD) hLineOrCall;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) DeviceSpecific1;
    dwParam2 = (DWORD) DeviceSpecific2;
    dwParam3 = (DWORD) DeviceSpecific3;
```

Parameters

dwDevice

A handle to either a line device or call. This is device specific.

dwCallbackInstance

The callback instance supplied when opening the line.

dwParam1

Device specific.

dwParam2

Device specific.

dwParam3

Device specific.

Return Values

No return value.

Remarks

The LINE_DEVSPECIFIC message is used by a service provider in conjunction with the **lineDevSpecific** function. Its meaning is device specific.

LINE_DEVSPECIFICFEATURE Overview

The LINE_DEVSPECIFICFEATURE message is sent to notify the application about device-specific events occurring on a line, address, or call. The meaning of the message and the interpretation of the parameters are device specific.

```
LINE_DEVSPECIFICFEATURE
dwDevice = (DWORD) hLineOrCall;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) DeviceSpecific1;
    dwParam2 = (DWORD) DeviceSpecific2;
    dwParam3 = (DWORD) DeviceSpecific3;
```

Parameters

dwDevice

A handle to either a line device or call. This is device specific.

dwCallbackInstance

The callback instance supplied when opening the line.

dwParam1

Device specific.

dwParam2

Device specific.

dwParam3

Device specific.

Return Values

No return value.

Remarks

The LINE_DEVSPECIFICFEATURE message is used by a service provider in conjunction with the **lineDevSpecificFeature** function. Its meaning is device specific.

LINE_GATHERDIGITS Overview

The LINE_GATHERDIGITS message is sent when the current buffered digit-gathering request has terminated or is canceled. The digit buffer may be examined after this message has been received by the application.

```
LINE_GATHERDIGITS
dwDevice = (DWORD) hCall;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) GatherTermination;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

A handle to the call.

dwCallbackInstance

The callback instance supplied when opening the line.

dwParam1

The reason why digit gathering was terminated. This parameter must be one and only one of the following LINEGATHERTERM_ constants:

LINEGATHERTERM_BUFFERFULL

The requested number of digits has been gathered. The buffer is full.

LINEGATHERTERM_TERMDIGIT

One of the termination digits matched a received digit. The matched termination digit is the last digit in the buffer.

LINEGATHERTERM_FIRSTTIMEOUT

The first digit timeout expired. The buffer contains no digits.

LINEGATHERTERM_INTERTIMEOUT

The inter-digit timeout expired. The buffer contains at least one digit.

LINEGATHERTERM_CANCEL

The request was canceled by this application, by another application, or because the call terminated.

dwParam2

Unused.

dwParam3

The "tick count" (number of milliseconds since Windows started) at which the digit gathering completed. For API versions prior to 0x0002000, this parameter is unused.

Return Values

No return value.

Remarks

The LINE_GATHERDIGITS message is only sent to the application that initiated the digit gathering on the call using **lineGatherDigits**.

If the **lineGatherDigits** function is used to cancel a previous request to gather digits, TAPI sends a LINE_GATHERDIGITS message with *dwParam1* set to LINEGATHERTERM_CANCEL to the application indicating that the originally specified buffer contains the digits gathered up to the cancellation.

Because the timestamp specified by *dwParam3* may have been generated on a computer other than the one on which the application is executing, it is useful only for comparison to other similarly timestamped messages generated on the same line device (LINE_GENERATE, LINE_MONITORDIGITS, LINE_MONITORMEDIA, LINE_MONITORTONE), in order to determine their *relative* timing (separation between events). The tick count can "wrap around" after approximately 49.7 days; applications must take this into account when performing calculations.

If the service provider does not generate the timestamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a timestamp at the point closest to the service provider generating the event so that the synthesized timestamp is as accurate as possible.

See Also

LINE_GENERATE, LINE_MONITORDIGITS, LINE_MONITORMEDIA, LINE_MONITORTONE, **lineGatherDigits**

LINE_GENERATE Overview

The LINE_GENERATE message is sent to notify the application that the current digit or tone generation has terminated. Only one such generation request can be in progress an a given call at any time. This message is also sent when digit or tone generation is canceled.

```
LINE_GENERATE
dwDevice = (DWORD) hCall;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) GenerateTermination;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

A handle to the call.

dwCallbackInstance

The callback instance supplied when opening the line.

dwParam1

The reason why digit or tone generation was terminated. This parameter must be one and only one of the following LINEGENERATETERM_ constants:

LINEGENERATETERM_DONE

The requested number of digits have been generated, or the requested tones have been generated for the requested duration.

LINEGENERATETERM_CANCEL

The digit or tone generation request was canceled by this application, by another application, or because the call terminated.

dwParam2

Unused.

dwParam3

The "tick count" (number of milliseconds since Windows started) at which the digit or tone generation completed. For API versions prior to 0x00020000, this parameter is unused.

Return Values

No return value.

Remarks

The LINE_GENERATE message is only sent to the application that requested the digit or tone generation.

Because the timestamp specified by *dwParam3* may have been generated on a computer other than the one on which the application is executing, it is useful only for comparison to other similarly timestamped messages generated on the same line device (LINE_GATHERDIGITS, LINE_MONITORDIGITS, LINE_MONITORMEDIA, LINE_MONITORTONE), in order to determine their *relative* timing (separation between events). The tick count can "wrap around" after approximately 49.7 days; applications must take this into account when performing calculations.

If the service provider does not generate the timestamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a timestamp at the point closest to the service provider generating the event so that the synthesized timestamp is as accurate as possible.

See Also

LINE_GATHERDIGITS, LINE_MONITORDIGITS, LINE_MONITORMEDIA, LINE_MONITORTONE

LINE_LINEDEVSTATE Overview

The LINE_LINEDEVSTATE message is sent when the state of a line device has changed. The application can invoke **lineGetLineDevStatus** to determine the new status of the line.

```
LINE_LINEDEVSTATE
dwDevice = (DWORD) hLine;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) DeviceState;
    dwParam2 = (DWORD) DeviceStateDetail1;
    dwParam3 = (DWORD) DeviceStateDetail2;
```

Parameters

dwDevice

A handle to the line device. This parameter is NULL when *dwParam1* is LINEDEVSTATE_REINIT.

dwCallbackInstance

The callback instance supplied when opening the line. 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 has changed. The parameter can be a combination of the following LINEDEVSTATE_ constants:

LINEDEVSTATE_OTHER

Device-status items other than those listed below have changed. The application should check the current device status to determine which items have changed.

LINEDEVSTATE_RINGING

The switch tells the line to alert the user. Service providers notify applications on each ring cycle by sending messages containing this constant. For example, in the United States, service providers send a message with this constant every six seconds.

LINEDEVSTATE_CONNECTED

The line was previously disconnected and is now connected to TAPI.

LINEDEVSTATE_DISCONNECTED

This line was previously connected and is now disconnected from TAPI.

LINEDEVSTATE_MSGWAITON

The "message waiting" indicator is turned on.

LINEDEVSTATE_MSGWAITOFF

The "message waiting" indicator is turned off.

LINEDEVSTATE_NUMCOMPLETIONS

The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_INSERVICE

The line is connected to TAPI. This happens when TAPI is first activated, or when the line wire is physically plugged in and in service at the switch while TAPI is active.

LINEDEVSTATE_OUTOFSERVICE

The line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE

Maintenance is being performed on the line at the switch. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_OPEN

The line has been opened by another application.

LINEDEVSTATE_CLOSE

The line has been closed by another application.

LINEDEVSTATE_NUMCALLS

The number of calls on the line device has changed.

LINEDEVSTATE_TERMINALS

The terminal settings have changed.

LINEDEVSTATE_ROAMMODE

The roaming state of the line device has changed.

LINEDEVSTATE_BATTERY

The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL

The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC

The line's device-specific information has changed.

LINEDEVSTATE_REINIT

Items have 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. The *dwDevice* parameter is left NULL for this state change as it applies to any of the lines in the system.

LINEDEVSTATE_LOCK

The locked status of the line device has changed. (For more information, refer to the description of the LINEDEVSTATUSFLAGS_LOCKED bit of the LINEDEVSTATUSFLAGS_ constants.)

LINEDEVSTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINEDEVCAPS** structure for the address have changed. The application should use **lineGetDevCaps** to read the updated structure.

LINEDEVSTATE_CONFIGCHANGE

Indicates that configuration changes have been made to one or more of the media devices associated with the line device. The application, if it desires, may use **lineGetDevConfig** to read the updated information.

LINEDEVSTATE_TRANSLATECHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINETRANSLATECAPS** structure have changed. The application should use **lineGetTranslateCaps** to read the updated structure.

LINEDEVSTATE_COMPLCANCEL

Indicates that the call completion identified by the completion ID contained in parameter *dwParam2* of the LINE_LINEDEVSTATE message has been externally canceled and is no longer considered valid (if that value were to be passed in a subsequent call to **lineUncompleteCall**, the function would fail with LINEERR_INVALCOMPLETIONID).

LINEDEVSTATE_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). A LINE_LINEDEVSTATE message with this value will normally be immediately followed by a LINE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in LINEERR_NODEVICE being returned to the application. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

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 are numbers in the range one to **dwNumRingModes**, where **dwNumRingModes** is a line device capability.

If *dwParam1* is LINEDEVSTATE_REINIT, and the message was issued by TAPI as a result of translation of a new API message into a REINIT message, then *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 is 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 the message was issued by TAPI as a result of translation of a new API message into a REINIT message, then *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 ID, if *dwParam2* is LINE_CREATE).

Return Values

No return value.

Remarks

The sending of the LINE_LINEDEVSTATE message can be controlled with **lineSetStatusMessages**. An application can indicate status item changes about which it wants to be notified. By default, all status reporting will be disabled except for LINEDEVSTATE_REINIT, which cannot be disabled. This message is sent to all applications that have a handle to the line, including those that called **lineOpen** with the *dwPrivileges* parameter set to LINECALLPRIVILEGE_NONE, LINECALLPRIVILEGE_OWNER, LINECALLPRIVILEGE_MONITOR, or permitted combinations of these.

See Also

LINE_CLOSE, LINE_CREATE, **LINEDEVCAPS**, **lineGetDevCaps**, **lineGetDevConfig**, **lineGetTranslateCaps**, **lineInitialize**, **lineOpen**, **lineSetStatusMessages**, **lineShutdown**,

LINETRANSLATECAPS, **lineUncompleteCall**

LINE_MONITORDIGITS Overview

The LINE_MONITORDIGITS message is sent when a digit is detected. The sending of this message is controlled with the **lineMonitorDigits** function.

```
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 supplied when opening the call's line.

dwParam1

The low-order byte contains the last digit received in ASCII.

dwParam2

The digit mode that was detected. This parameter must be one and only one of the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE

Detect digits as audible clicks that are the result of rotary pulse sequences. Valid digits for pulse are '0' through '9'.

LINEDIGITMODE_DTMF

Detect digits as DTMF tones. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

LINEDIGITMODE_DTMFEND

Detect and provide application notification of DTMF down edges. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

dwParam3

The "tick count" (number of milliseconds since Windows started) at which the specified digit was detected. For API versions prior to 0x00020000, this parameter is unused.

Return Values

No return value.

Remarks

The LINE_MONITORDIGITS message is sent to the application that has enabled digit monitoring.

Because this timestamp may have been generated on a computer other than the one on which the application is executing, it is useful only for comparison to other similarly timestamped messages generated on the same line device (LINE_GATHERDIGITS, LINE_GENERATE, LINE_MONITORMEDIA, LINE_MONITORTONE), in order to determine their *relative* timing (separation between events). The tick count can "wrap around" after approximately 49.7 days; applications must take this into account when

performing calculations.

If the service provider does not generate the timestamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a timestamp at the point closest to the service provider generating the event so that the synthesized timestamp is as accurate as possible.

See Also

LINE_GATHERDIGITS, LINE_GENERATE, LINE_MONITORMEDIA, LINE_MONITORTONE, **lineMonitorDigits**

LINE_MONITORMEDIA Overview

The LINE_MONITORMEDIA message is sent when a change in the call's media mode is detected. The sending of this message is controlled with the **lineMonitorMedia** function.

```
LINE_MONITORMEDIA
     dwDevice = (DWORD) hCall;
     dwCallbackInstance = (DWORD) hCallback;
     dwParam1 = (DWORD) MediaMode;
     dwParam2 = (DWORD) 0;
     dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

A handle to the call.

dwCallbackInstance

The callback instance supplied when opening the line.

dwParam1

The new media mode. This parameter must be one and only one of the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_INTERACTIVEVOICE

The presence of voice energy has been detected and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE

The presence of voice energy has been detected and the call is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

A data modem session has been detected.

LINEMEDIAMODE_G3FAX

A group 3 fax has been detected.

LINEMEDIAMODE_TDD

A TDD (Telephony Devices for the Deaf) session has been detected.

LINEMEDIAMODE_G4FAX

A group 4 fax has been detected.

LINEMEDIAMODE_DIGITALDATA

Digital data has been detected.

LINEMEDIAMODE_TELETEX

A teletex session has been detected. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX

A videotex session has been detected. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX

A telex session has been detected. Telex is one the telematic services.

LINEMEDIAMODE_MIXED

A mixed session has been detected. Mixed is one the telematic services.

LINEMEDIAMODE_ADSI

An ADSI (Analog Display Services Interface) session has been detected.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

dwParam2

Unused.

dwParam3

The "tick count" (number of milliseconds since Windows started) at which the specified media was detected. For API versions prior to 0x00020000, this parameter is unused.

Return Values

No return value.

Remarks

The LINE_MONITORMEDIA message is sent to the application that has enabled media monitoring for the media mode detected.

Because this timestamp may have been generated on a computer other than the one on which the application is executing, it is useful only for comparison to other similarly timestamped messages generated on the same line device (LINE_GATHERDIGITS, LINE_GENERATE, LINE_MONITORDIGITS, LINE_MONITORTONE), in order to determine their *relative* timing (separation between events). The tick count can "wrap around" after approximately 49.7 days; applications must take this into account when performing calculations.

If the service provider does not generate the timestamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a timestamp at the point closest to the service provider generating the event so that the synthesized timestamp is as accurate as possible.

See Also

LINE_GATHERDIGITS, LINE_GENERATE, LINE_MONITORDIGITS, LINE_MONITORTONE

LINE_MONITORTONE Overview

The LINE_MONITORTONE message is sent when a tone is detected. The sending of this message is controlled with the **lineMonitorTones** function.

```
LINE_MONITORTONE
    dwDevice = (DWORD) hCall;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) dwAppSpecific;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

A handle to the call.

dwCallbackInstance

The callback instance supplied when opening the call's line.

dwParam1

The application-specific **dwAppSpecific** field of the **LINEMONITORTONE** structure for the tone that was detected.

dwParam2

Unused.

dwParam3

The "tick count" (number of milliseconds since Windows started) at which the tone was detected. For API versions prior to 0x00020000, this parameter is unused.

Return Values

No return value.

Remarks

The LINE_MONITORTONE message is only sent to the application that has requested the tone be monitored.

Because this timestamp may have been generated on a computer other than the one on which the application is executing, it is useful only for comparison to other similarly timestamped messages generated on the same line device (LINE_GATHERDIGITS, LINE_GENERATE, LINE_MONITORDIGITS, LINE_MONITORTONE), in order to determine their *relative* timing (separation between events). The tick count can "wrap around" after approximately 49.7 days; applications must take this into account when performing calculations.

If the service provider does not generate the timestamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a timestamp at the point closest to the service provider generating the event so that the synthesized timestamp is as accurate as possible.

See Also

LINE_GATHERDIGITS, LINE_GENERATE, LINE_MONITORDIGITS, LINE_MONITORTONE, **LINEMONITORTONE**, **lineMonitorTones**

LINE_PROXYREQUEST Overview

The LINE_PROXYREQUEST message delivers a request to a registered proxy function handler.

```
LINE_PROXYREQUEST
     dwDevice = (DWORD) hLine;
     dwCallbackInstance = (DWORD) dwInstanceData;
     dwParam1 = (DWORD) lpProxyRequest;
     dwParam2 = (DWORD) 0;
     dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

The application's handle to the line device on which the agent status has changed.

dwCallbackInstance

The callback instance supplied when opening the call's line.

dwParam1

Pointer to a **LINEPROXYREQUEST** structure containing the request to be processed by the proxy handler application.

dwParam2

Reserved.

dwParam3

Reserved.

Return Values

No return value.

Remarks

The LINE_PROXYREQUEST message is sent only to the first application that registered to handle proxy requests of the type being delivered.

The application should process the request contained in the proxy buffer and call **lineProxyResponse** to return data or deliver results. Processing of the request should be done within the context of the application's TAPI callback function *only* if it can be performed immediately, without waiting for response from any other entity. If the application needs to communicate with other entities (for example, a service provider to handle PBX-based ACD, or any other system service which might result in blocking), then the request should be queued within the application and the callback function exited to avoid delaying the receipt of further TAPI messages by the application.

At the time the LINE_PROXYREQUEST is delivered to the proxy handler, TAPI has already returned a positive *dwRequestID* function result to the original application and unblocked the calling thread to continue execution. The application is awaiting a LINE_REPLY message, which is automatically generated when the proxy handler application calls **lineProxyResponse**.

The application shall not free the memory pointed to by *lpProxyRequest*. TAPI will free the memory during the execution of **lineProxyResponse**. The application shall call **lineProxyResponse** exactly once for each LINE_PROXYREQUEST message.

If the application receives a LINE_CLOSE message while it has pending proxy requests, it should call **lineProxyResponse** for each pending request, passing in an appropriate *dwResult* value (such as LINEERR_OPERATIONFAILED).

See Also

LINE_CLOSE, LINE_REPLY, **LINEPROXYREQUEST**, **lineProxyResponse**

LINE_REMOVE Overview

The LINE_REMOVE message is sent to inform an application of the removal (deletion from the system) of a line device. Generally, this is not used for temporary removals, such as extraction of PCMCIA devices, but only for permanent removals in which the device would no longer be reported by the service provider if TAPI were reinitialized.

```
LINE_REMOVE
    dwDevice = (DWORD) 0;
    dwCallbackInstance = (DWORD) 0;
    dwParam1 = (DWORD) dwDeviceID;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

Reserved; set to 0.

dwCallbackInstance

Reserved; set to 0.

dwParam1

Identifier of the line device that was removed.

dwParam2

Reserved; set to 0.

dwParam3

Reserved; set to 0.

Return Values

No return value.

Comments and Backward Compatibility

Applications supporting TAPI version 0x00020000 or above are sent a LINE_REMOVE message. This informs them that the device has been removed from the system. The LINE_REMOVE message will have been preceded by a LINE_CLOSE message on each line handle, if the application had the line open. This message will be sent to all applications supporting TAPI version 0x00020000 or above which have called **lineInitializeEx**, including those that do not have any line devices open at the time.

Older applications are sent a LINE_LINEDEVSTATE message specifying LINEDEVSTATE_REMOVED, followed by a LINE_CLOSE message. Unlike the LINE_REMOVE message, however, these older applications can receive these messages only if they have the line open when it is removed. If they do not have the line open, their only indication that the device was removed would be receiving a LINEERR_NODEVICE when they attempt to access the device.

After a device has been removed, any attempt to access the device by its device ID will result in a LINEERR_NODEVICE error. After all TAPI applications have shutdown so that TAPI can restart, when TAPI is reinitialized, the removed device will no longer occupy a device ID.

Implementation Note It is TAPI that will return this LINEERR_NODEVICE; after a LINE_REMOVE message is received from a service provider; no further calls will be made to that

service provider using that line device ID.

See Also

LINE_CLOSE, LINE_LINEDEVSTATE, **lineInitializeEx**

LINE_REPLY Overview

The LINE_REPLY message is sent to report the results of function calls that completed asynchronously.

```
LINE_REPLY
     dwDevice = (DWORD) 0;
     dwCallbackInstance = (DWORD) hCallback;
     dwParam1 = (DWORD) idRequest;
     dwParam2 = (DWORD) Status;
     dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

Not used.

dwCallbackInstance

Returns the application's callback instance.

dwParam1

The request ID 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

Unused.

Return Values

No return value.

Remarks

Functions that operate asynchronously return a positive request ID value to the application. This request ID is returned with the reply message to identify the request that was completed. The other parameter for the LINE_REPLY message carries the success or failure indication. Possible errors are the same as those defined by the corresponding function. This message cannot be disabled.

In some cases, an application may fail to receive the LINE_REPLY message corresponding to a call to an asynchronous function. This occurs if the corresponding call handle is deallocated before the message has been received.

LINE_REQUEST Overview

The LINE_REQUEST message is sent to report the arrival of a new request from another application.

```
LINE_REQUEST
     dwDevice = (DWORD) 0;
     dwCallbackInstance = (DWORD) hRegistration;
     dwParam1 = (DWORD) RequestMode;
     dwParam2 = (DWORD) RequestModeDetail1;
     dwParam3 = (DWORD) RequestModeDetail2;
```

Parameters

dwDevice

Not used.

dwCallbackInstance

The registration instance of the application specified on **lineRegisterRequestRecipient**.

dwParam1

The request mode of the newly pending request. This parameter uses the following LINEREQUESTMODE_ constants:

LINEREQUESTMODE_MAKECALL

A **tapiRequestMakeCall** request.

dwParam2

If *dwParam1* is set to LINEREQUESTMODE_DROP, *dwParam2* contains the *hWnd* of the application requesting the drop. Otherwise, *dwParam2* is unused.

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 unused.

Return Values

No return value.

Remarks

The LINE_REQUEST message is sent to the highest priority application that has registered for the corresponding request mode. This message indicates the arrival of an Assisted Telephony request of the specified request mode. If *dwParam1* is LINEREQUESTMODE_MAKECALL or LINEREQUESTMODE_MEDIACALL, the application can invoke **lineGetRequest** using the corresponding request mode to receive the request. If *dwParam1* is LINEREQUESTMODE_DROP, the message contains all of the information the request recipient needs in order to perform the request.

See Also

lineGetRequest, **lineRegisterRequestRecipient**, **tapiRequestMakeCall**

Phone Device Messages

The following section contains the reference for phone device messages.

PHONE_BUTTON Overview

The PHONE_BUTTON message is sent to notify the application that button press monitoring is enabled if it has detected a button press on the local phone.

```
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 application's callback instance provided when opening the phone device.

dwParam1

The button/lamp ID of the button that was pressed. Note that button IDs 0 through 11 are always the KEYPAD buttons, with '0' being button ID 0, '1' being button ID 1 (and so on through button ID 9), and with '*' being button ID 10, and '#' being button ID 11. Additional information about a button ID is available with **phoneGetDevCaps** and **phoneGetButtonInfo**.

dwParam2

The button mode of the button. 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 twelve keypad buttons, '0' through '9', '*', and '#'.

PHONEBUTTONMODE_LOCAL

The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY

The button is a "soft" button associated with the phone's 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

Indicates that the up or down state of the button is not known at this time, but may become known at a future time.

PHONEBUTTONSTATE_UNAVAIL

Indicates that the up or down state of the button is not known to the service provider, and will not become known at a future time.

Return Values

No return value.

Remarks

A PHONE_BUTTON message is sent whenever a button changes state. An application is guaranteed that for each button down event, it will eventually be sent a corresponding button up event. A service provider that is incapable of detecting the actual button up is required to generate the button up message shortly after the button down message for each button press.

See Also

phoneGetButtonInfo, **phoneGetDevCaps**

PHONE_CLOSE Overview

The PHONE_CLOSE message is sent when an open phone device has been forcibly closed as part of resource reclamation. The device handle is no longer valid once this message has been sent.

```
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 has been sent

dwCallbackInstance

The application's callback instance provided when opening the phone device.

dwParam1

Unused.

dwParam2

Unused.

dwParam3

Unused.

Return Values

No return value.

Remarks

The PHONE_CLOSE message is only sent to an application after an open phone has been forcibly closed. This may be done to prevent a single application from monopolizing a phone device for too long. Whether the phone can be reopened immediately after a forced close is device specific.

An open phone device may also be forcibly closed after the user has modified the configuration of that phone or its driver. If the user wants the configuration changes to be effective immediately (as opposed to after the next system restart), and these changes affect the application's current view of the device (such as a change in device capabilities), then a service provider may forcibly close the phone device.

PHONE_CREATE Overview

The PHONE_CREATE message is sent to inform applications of the creation of a new phone device.

```
PHONE_CREATE
hPhone = (HPHONE) hPhoneDev;
    dwCallbackInstance = (DWORD) 0;
    dwParam1 = (DWORD) idDevice;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

hPhone

Unused.

dwCallbackInstance

Unused.

dwParam1

The *dwDeviceID* of the newly-created device.

dwParam2

Unused.

dwParam3

Unused.

Return Values

No return value.

Remarks

Applications that negotiated API versoin 0x00010003 are sent a PHONE_STATE message specifying PHONESTATE_REINIT, which requires them to shut down their use of the API and call **phoneInitialize** again to obtain the new number of devices. However, TAPI version 1.4 and above do not require *all* applications to shut down before allowing applications to reinitialize; reinitialization can take place immediately when a new device is created.

Applications supporting TAPI version 0x00010004 or above are sent a PHONE_CREATE message. This informs them of the existence of the new device and its new device ID. The application can then choose whether or not to attempt working with the new device at its leisure.

See Also

PHONE_STATE, **phoneInitialize**, **phoneInitializeEx**

PHONE_DEVSPECIFIC Overview

The implementation sends the PHONE_DEVSPECIFIC message to an application to notify the application about device-specific events occurring at the phone. The meaning of the message and the interpretation of the parameters is implementation-defined.

```
PHONE_DEVSPECIFIC
hPhone = (HPHONE) hPhoneDevice;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) DeviceSpecific1;
    dwParam2 = (DWORD) DeviceSpecific2;
    dwParam3 = (DWORD) DeviceSpecific3;
```

Parameters

hPhone

A handle to the phone device.

dwCallbackInstance

The application's callback instance provided when opening the phone device.

dwParam1

Device specific.

dwParam2

Device specific.

dwParam3

Device specific.

Return Values

No return value.

PHONE_REMOVE Overview

The PHONE_REMOVE message is sent to inform an application of the removal (deletion from the system) of a phone device. Generally, this is not used for temporary removals, such as extraction of PCMCIA devices, but only for permanent removals in which the device would no longer be reported by the service provider if TAPI were reinitialized.

```
PHONE_REMOVE
    dwDevice = (DWORD) 0;
    dwCallbackInstance = (DWORD) 0;
    dwParam1 = (DWORD) dwDeviceID;
    dwParam2 = (DWORD) 0;
    dwParam3 = (DWORD) 0;
```

Parameters

dwDevice

Reserved; set to 0.

dwCallbackInstance

Reserved; set to 0.

dwParam1

Identifier of the phone device that was removed.

dwParam2

Reserved; set to 0.

dwParam3

Reserved; set to 0.

Return Values

No return value.

Comments and Backward Compatibility

Applications TAPI version 0x00020000 and above are sent a PHONE_REMOVE message. This informs them that the device has been removed from the system. The PHONE_REMOVE message will have been preceded by a PHONE_CLOSE message on each phone handle, if the application had the phone open. This message will be sent to all applications supporting TAPI version 0x00020000 or above which have called **phoneInitializeEx**, including those that do not have any phone devices open at the time.

Older applications (which negotiated TAPI version 0x00010004 or below) are sent a PHONE_STATE message specifying PHONESTATE_REMOVED, followed by a PHONE_CLOSE message. Unlike the PHONE_REMOVE message, however, these older applications can receive these messages only if they have the phone open when it is removed. If they do not have the phone open, their only indication that the device was removed would be receiving a PHONEERR_NODEVICE when they attempt to access the device.

After a device has been removed, any attempt to access the device by its device ID will result in a PHONEERR_NODEVICE error. After all TAPI applications have shutdown so that TAPI can restart, when TAPI is reinitialized, the removed device will no longer occupy a device ID.

Implementation Note It is TAPI that will return this PHONEERR_NODEVICE after a

PHONE_REMOVE message is received from a service provider; no further calls will be made to that service provider using that phone device ID.

See Also

PHONE_CLOSE, PHONE_STATE, **phoneInitialize**, **phoneInitializeEx**

PHONE_REPLY Overview

The PHONE_REPLY message is sent to an application' to report the results of function call that completed asynchronously.

```
PHONE_REPLY
hPhone = (HPHONE) 0;
    dwCallbackInstance = (DWORD) hCallback;
    dwParam1 = (DWORD) idRequest;
    dwParam2 = (DWORD) Status;
    dwParam3 = (DWORD) 0;
```

Parameters

hDevice

Unused.

dwCallbackInstance

Returns the application's callback instance.

dwParam1

The request ID 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

Unused.

Return Values

No return value.

Remarks

Functions that operate asynchronously return a positive request ID value to the application. This request ID is returned with the reply message to identify the request that was completed. The other parameter for the PHONE_REPLY message carries the success or failure indication. Possible errors are the same as those defined by the corresponding function. This message cannot be disabled.

PHONE_STATE Overview

TAPI sends the PHONE_STATE message to an application whenever the status of a phone device changes.

```
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 application's callback instance provided when opening the phone device.

dwParam1

The phone state that has changed. This parameter uses the following PHONESTATE_ constants:

PHONESTATE_OTHER

Phone-status items other than those listed below have changed. The application should check the current phone status to determine which items have 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 connecting the phone to the computer is plugged in with TAPI active.

PHONESTATE_DISCONNECTED

The connection between the phone device and TAPI was just broken. This happens when the wire connecting the phone set to the computer is unplugged while TAPI is active.

PHONESTATE_OWNER

The number of owners for the phone device has changed.

PHONESTATE_MONITORS

The number of monitors for the phone device has changed.

PHONESTATE_DISPLAY

The display of the phone has changed.

PHONESTATE_LAMP

A lamp of the phone has changed.

PHONESTATE_RINGMODE

The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME

The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH

The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME

The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN

The handset's microphone gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH

The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME

The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN

The speakerphone's microphone gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH

The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME

The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN

The headset's microphone gain setting has changed.

PHONESTATE_SUSPEND

The application's use of the phone device is temporarily suspended.

PHONESTATE_RESUME

The application's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC

The phone's device-specific information has changed.

PHONESTATE_REINIT

Items have 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 is left NULL for this state change as it applies to any of the phones in the system.

PHONESTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **PHONECAPS** structure have changed. The application should use **phoneGetDevCaps** to read the updated structure.

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). A PHONE_STATE message with this value will normally be immediately followed by a PHONE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in PHONEERR_NODEVICE being returned to the application. If a service provider sends a PHONE_STATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or higher; applications negotiating a previous API version will not receive any notification.

dwParam2

Phone-state-dependent information detailing the status change. This field is not used if multiple flags are set in *dwParam1*, because multiple status items have changed. The application should invoke **phoneGetStatus** to obtain complete set of information.

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 ID of the lamp that has 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 device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC

The device's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER

The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER

The device's microphone and speaker are both active.

dwParam3

Unused.

Return Values

No return value.

Remarks

Sending the PHONE_STATE message to the application can be controlled and queried using **phoneSetStatusMessages** and **phoneGetStatusMessages**. By default, this message is disabled for all state changes except for PHONESTATE_REINIT, which cannot be disabled. This message is sent to all applications that have a handle to the phone, including those that called **phoneOpen** with the *dwPrivileges* parameter set to PHONEPRIVILEGE_OWNER or PHONEPRIVILEGE_MONITOR.

A PHONE_STATE message with an *Owners* and/or *Monitors* indication is sent to applications that already have a handle for the phone. This can be the result of another application changing ownership or monitorship of the phone device with **phoneOpen**, **phoneClose** or **phoneShutdown**.

See Also

PHONE_CLOSE, **PHONECAPS**, **phoneClose**, **phoneGetDevCaps**, **phoneGetStatus**, **phoneGetStatusMessages**, **phoneInitialize**, **phoneInitializeEx**, **phoneOpen**, **phoneSetStatusMessages**, **phoneShutdown**

Assisted Telephony Messages

The following messages are used by Assisted Telephony.

Formatted Error Messages

This section discusses the **TAPIERROR_FORMATMESSAGE()** macro. This macro generates an identifier for standard TAPI error message text strings that can be obtained by Win32 applications using the Win32 **FormatMessage()** function.

This mechanism should be used only for displaying information on errors for which the application has no defined method of recovery (that is, unexpected or internal errors). In most cases (unlike the simplified example below), it is desirable to include additional text informing the user of actions the application will take (or the user should take) as a result of the unhandled error.

If the application gets an error result from any TAPI function, the error value can be passed through the **TAPIERROR_FORMATMESSAGE()** macro, which in turn generates the real value to be passed to **FormatMessage()**. **FormatMessage()** produces an error string corresponding to the error in a buffer. For example:

```
lResult = lineClose(hLine);

if (lResult < 0)
{
        FormatMessage(FORMAT_MESSAGE_FROM_HMODULE,
                    (LPCVOID)GetModuleHandle("TAPI32.DLL"),
                    TAPIERROR_FORMATMESSAGE(lResult),
                    0,
                    (LPTSTR)pBuf,
                    BUFSIZE,
                    NULL);
        MessageBox(hWnd,pBuf,"TAPI ERROR",MB_OK);
}
```

TAPI_REPLY

The TAPI_REPLY message is nonfunctional in Win32 -based applications and obsolete for all classes of Windows-based applications. It should not be used.

Structures

This section contains the reference for structures for line devices and phone devices.

Line Device Structures

The following topics describe the data structures used by the Telephony API. They are listed in alphabetical order.

Memory Allocation

Memory for all data structures used by the API must be allocated by the application. The application passes a pointer to the API function that returns the information, and the function fills the data structure with the requested information. If the operation is asynchronous, then the information is not available until the asynchronous reply message indicates success.

All data structures used to pass information between the application and the Telephony API are *flattened*. This means that data structures do not contain pointers to substructures that contain variably sized components of information. Instead, data structures that are used to pass variable amounts of information back to the application have the following meta structure:

```
DWORD  dwTotalSize;
DWORD  dwNeededSize;
DWORD  dwUsedSize;
  <fixed size fields>
DWORD  dw<VarSizeField1>Size;
DWORD  dw<VarSizeField1>Offset;
  <fixed size fields>
DWORD  dw<VarSizeField2>Size;
DWORD  dw<VarSizeField2>Offset;
  <common extensions>
  <var sized field1>
  <var sized field2>
```

The **dwTotalSize** field is the size in bytes allocated to this data structure. It marks the end of the data structure and is set by the application before it invokes the function that uses this data structure. The function will not read or write beyond this size. An application must set the **dwTotalSize** field to indicate the total number of bytes allocated for TAPI to return the contents of the structure.

The **dwNeededSize** field is filled in by TAPI. It indicates how many bytes are needed to return all the information requested. The existence of variably sized fields often makes it impossible for the application to estimate the data structure size it needs to allocate. This field simply returns the number of bytes actually needed for the information. This number could be smaller than, equal to, or larger than **dwTotalSize**, and it includes space for the **dwTotalSize** field itself. If larger, the returned structure is only partially filled. If the fields the application is interested in are available in the partial structure, nothing else must be done. Otherwise, the application should allocate a structure at least the size of **dwNeededSize** and invoke the function again. Usually, enough space will be available this time to return all the information, although it is possible the size could have increased again.

The **dwUsedSize** field is filled in by TAPI if it returns information to the application to indicate the actual size in bytes of the portion of the data structure that contains useful information. If, for example, a structure that was allocated was too small and the truncated field is a variably sized field, **dwNeededSize** will be larger than **dwTotalSize**, and the truncated field will be left empty. The **dwUsedSize** field could therefore be smaller than **dwTotalSize**. Partial field values are not returned.

Following this header is the fixed part of the data structure. It contains regular fields and size/offset pairs that describe the actual variably sized fields. The offset field contains the offset in bytes of the variably sized field from the beginning of the record. The size field contains the size in bytes of the variably sized field. If a variably sized field is empty, then the size field is zero and the offset is set to zero. Variably sized fields that would be truncated if the total structure size is insufficient are left empty. That is, their size field is set to zero and the offset is set to zero. The variably sized fields follow the fixed fields.

Variably Sized Data Structures

When variably sized data structures are used to transmit information between TAPI and the application, the application is responsible for allocating the necessary memory. The amount of memory allocated must be at least large enough for the fixed portion of the data structure, and is set by the application in the **dwTotalSize** field of the data structure. The **dwUsedSize** and **dwNeededSize** fields are filled in by TAPI. If **dwTotalSize** is less than the size of the fixed portion, then LINEERR/ PHONEERR_STRUCTURETOOSMALL is returned. If a function returns success, then all the fields in the fixed portion have been filled in. The **dwUsedSize** and **dwNeededSize** fields can be compared to determine if all variable parts have been filled in, and how much space would be required to fill them all in.

If **dwNeededSize** is equal to **dwUsedSize**, then all fixed and variable parts have been filled in. If **dwNeededSize** is larger than **dwUsedSize**, some variable parts may have been filled in, but exactly which variably sized fields have been filled in is undefined. No variable part is ever truncated, and variable parts that would have been truncated due to insufficient space are indicated by having both of their corresponding "Offset" and "Size" parts set to zero. If these are not both zero (and no error was returned), they indicate the offset and size of valid, nontruncated variable-part data.

An application can always guarantee that all variable parts are filled in by allocating and indicating **dwNeededSize** bytes for the structure and calling the "Get" function again until the function returns success and **dwNeededSize** "covers" **dwUsedSize**. This should happen on the second try except for race conditions that cause changes in the size of variable parts between calls, which should be a rare occurrence.

Note All ASCII, DBCS, and Unicode strings that occur in variable-sized structures should be NULL-terminated according to normal C string handling conventions.

Extensibility

Provisions are made for extending constants and structures both in a device-independent way and in a device-specific (vendor-specific) way. In constants that are scalar enumerations, a range of values is reserved for future common extensions. The remainder of values are identified as device specific. A vendor can define meanings for these values in any way desired. Their interpretation is keyed to the *extension ID* provided in the **LINEDEVCAPS** data structure. For constants that are defined as bit flags, a range of low-order bits are reserved, where the high-order bits can be extension specific. It is recommended that extended values and bit arrays use bits from the highest value or high-order bit down. This leaves the option to move the border between the common portion and extension portion if there is a need to do so in the future. Extensions to data structures are assigned a variably sized field with size/offset being part of the fixed part. TAPI describes for each data structure what device-specific extensions are allowed.

In addition to recognizing a specific extension ID, the application must negotiate the extension version number that the application and the service provider will operate under. This is done in the second version negotiation phase of the **lineGetDevCaps** function.

An extension ID is a globally unique identifier. There is no central registry for extension IDs. Instead, they are generated locally by the manufacturer by a utility that is available with the toolkit. The number is made up of parts such as a unique LAN address, time of day, and random number, to guarantee global uniqueness. Globally Unique Identifiers are designed to be distinguishable from HP/DEC universally unique identifiers and are thus fully compatible with them.

LINEADDRESSCAPS Overview

The **LINEADDRESSCAPS** structure describes the capabilities of a specified address.

```
typedef struct lineaddresscaps_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwLineDeviceID;

    DWORD  dwAddressSize;
    DWORD  dwAddressOffset;

    DWORD  dwDevSpecificSize;
    DWORD  dwDevSpecificOffset;

    DWORD  dwAddressSharing;
    DWORD  dwAddressStates;
    DWORD  dwCallInfoStates;
    DWORD  dwCallerIDFlags;
    DWORD  dwCalledIDFlags;
    DWORD  dwConnectedIDFlags;
    DWORD  dwRedirectionIDFlags;
    DWORD  dwRedirectingIDFlags;
    DWORD  dwCallStates;
    DWORD  dwDialToneModes;
    DWORD  dwBusyModes;
    DWORD  dwSpecialInfo;
    DWORD  dwDisconnectModes;

    DWORD  dwMaxNumActiveCalls;
    DWORD  dwMaxNumOnHoldCalls;
    DWORD  dwMaxNumOnHoldPendingCalls;
    DWORD  dwMaxNumConference;
    DWORD  dwMaxNumTransConf;

    DWORD  dwAddrCapFlags;
    DWORD  dwCallFeatures;
    DWORD  dwRemoveFromConfCaps;
    DWORD  dwRemoveFromConfState;
    DWORD  dwTransferModes;
    DWORD  dwParkModes;

    DWORD  dwForwardModes;
    DWORD  dwMaxForwardEntries;
    DWORD  dwMaxSpecificEntries;
    DWORD  dwMinFwdNumRings;
    DWORD  dwMaxFwdNumRings;

    DWORD  dwMaxCallCompletions;
    DWORD  dwCallCompletionConds;
    DWORD  dwCallCompletionModes;
    DWORD  dwNumCompletionMessages;
```

```
    DWORD  dwCompletionMsgTextEntrySize;
    DWORD  dwCompletionMsgTextSize;
    DWORD  dwCompletionMsgTextOffset;
    DWORD  dwAddressFeatures;

    DWORD  dwPredictiveAutoTransferStates;
    DWORD  dwNumCallTreatments;
    DWORD  dwCallTreatmentListSize;
    DWORD  dwCallTreatmentListOffset;
    DWORD  dwDeviceClassesSize;
    DWORD  dwDeviceClassesOffset;
    DWORD  dwMaxCallDataSize;
    DWORD  dwCallFeatures2;
    DWORD  dwMaxNoAnswerTimeout;
    DWORD  dwConnectedModes;
    DWORD  dwOfferingModes;
    DWORD  dwAvailableMediaModes;
} LINEADDRESSCAPS, FAR *LPLINEADDRESSCAPS;
```

Members

dwTotalSize

The total size in bytes 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.

dwLineDeviceID

The device ID of the line device with which this address is associated.

dwAddressSize

dwAddressOffset

The size in bytes of the variably sized address field and the offset in bytes from the beginning of this data structure.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized device-specific field and the offset in bytes from the beginning of this data structure.

dwAddressSharing

The sharing mode of the address. Values are:

LINEADDRESSSHARING_PRIVATE

An address with *private* sharing mode is only assigned to a single line or station.

LINEADDRESSSHARING_BRIDGEDEXCL

An address with a *bridged-exclusive* sharing mode is assigned to one or more other lines or stations. The *exclusive* portion refers to the fact that only one of the bridged parties can be connected with a remote party at any given time.

LINEADDRESSSHARING_BRIDGEDNEW

An address with a *bridged-new* sharing mode is assigned to one or more other lines or stations. The *new* portion refers to the fact that activities by the different bridged parties result in the creation of new calls on the address.

LINEADDRESSSHARING_BRIDGEDSHARED

An address with a *bridged-shared* sharing mode is also assigned to one or more other lines or stations. The *shared* portion refers to the fact that if one of the bridged parties is connected with a remote party, the remaining bridged parties can share in the conversation (as in a conference) by activating that call appearance.

LINEADDRESSSHARING_MONITORED

An address with a monitored address mode simply monitors the status of that address. The status is either idle or in use. The message LINE_ADDRESSSTATE notifies the application about these changes.

dwAddressStates

This field contains the address states changes for which the application may get notified in the LINE_ADDRESSSTATE message. It uses the following LINEADDRESSSTATE_ constants:

LINEADDRESSSTATE_OTHER

Address status items other than those listed below have changed. The application should check the current address status to determine which items have changed.

LINEADDRESSSTATE_DEVSPECIFIC

The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO

The address has changed to idle (it is not in use by any stations).

LINEADDRESSSTATE_INUSEONE

The address has changed from being idle or from being in use by many bridged stations to being in use by just one station.

LINEADDRESSSTATE_INUSEMANY

The monitored or bridged address has changed to being in use by one station to being used by more than one station.

LINEADDRESSSTATE_NUMCALLS

The number of calls on the address has changed. This is the result of 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 has changed, including the number of rings for determining a "no answer" condition. The application should check the address status to retrieve details about the address's current forwarding status.

LINEADDRESSSTATE_TERMINALS

The terminal settings for the address have changed.

LINEADDRESSSTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINEADDRESSCAPS** structure for the address have changed. The application should use **lineGetAddressCaps** to read the updated structure. If a service provider sends a

LINE_ADDRESSSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINEDEVSTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI in order to obtain the updated information.

dwCallInfoStates

This field describes the call information elements that are meaningful for all calls on this address. An application may get notified about changes in some of these states in LINE_CALLINFO messages. It uses the following LINECALLINFOSTATE_ constants:

LINECALLINFOSTATE_OTHER

Call information items other than those listed below have changed. The application should check the current call information to determine which items have changed.

LINECALLINFOSTATE_DEVSPECIFIC

The device-specific field of the call information.

LINECALLINFOSTATE_BEARERMODE

The bearer mode field of the call information record.

LINECALLINFOSTATE_RATE

The rate field of the call information record.

LINECALLINFOSTATE_MEDIAMODE

The media-mode field of the call information record.

LINECALLINFOSTATE_APPSPECIFIC

The application-specific field of the call information record.

LINECALLINFOSTATE_CALLID

The call ID field of the call information record.

LINECALLINFOSTATE_RELATEDCALLID

The related call ID field of the call information record.

LINECALLINFOSTATE_ORIGIN

The origin field of the call information record.

LINECALLINFOSTATE_REASON

The reason field of the call information record.

LINECALLINFOSTATE_COMPLETIONID

The completion ID field of the call information record.

LINECALLINFOSTATE_NUMOWNERINCR

The number of owner field in the call information record was increased.

LINECALLINFOSTATE_NUMOWNERDECR

The number of owner field in the call information record was decreased.

LINECALLINFOSTATE_NUMMONITORS

The number of monitors field in the call information record has changed.

LINECALLINFOSTATE_TRUNK

The trunk field of the call information record has changed.

LINECALLINFOSTATE_CALLERID

One of the callerID-related fields of the call information record has changed.

LINECALLINFOSTATE_CALLEDID

One of the calledID-related fields of the call information record has changed.

LINECALLINFOSTATE_CONNECTEDID

One of the connectedID-related fields of the call information record has changed.

LINECALLINFOSTATE_REDIRECTIONID

One of the redirectionID-related fields of the call information record has changed.

LINECALLINFOSTATE_REDIRECTINGID

One of the redirectingID-related fields of the call information record has changed.

LINECALLINFOSTATE_DISPLAY

The display field of the call information record.

LINECALLINFOSTATE_USERUSERINFO

The user-to-user information of the call information record.

LINECALLINFOSTATE_HIGHLEVELCOMP

The high-level compatibility field of the call information record.

LINECALLINFOSTATE_LOWLEVELCOMP

The low-level compatibility field of the call information record.

LINECALLINFOSTATE_CHARGINGINFO

The charging information of the call information record.

LINECALLINFOSTATE_TERMINAL

The terminal mode information of the call information record.

LINECALLINFOSTATE_DIALPARAMS

The dial parameters of the call information record.

LINECALLINFOSTATE_MONITORMODES

One or more of the digit, tone, or media monitoring fields in the call information record has changed.

dwCallerIDFlags

dwCalledIDFlags

dwConnectedIDFlags

dwRedirectionIDFlags

dwRedirectingIDFlags

These fields describe the various party ID information types that may be provided for calls on this address. It uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED

Caller ID information for the call has been blocked by the caller but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA

Caller ID information for the call is not available because it is not propagated all the way by the network.

LINECALLPARTYID_NAME

The caller ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the caller ID name variably sized field.

LINECALLPARTYID_ADDRESS

The caller ID information for the call is the caller's number and is provided in the caller ID variably sized field.

LINECALLPARTYID_PARTIAL

Caller ID information for the call is valid but is limited to partial number information.

LINECALLPARTYID_UNKNOWN

Caller ID information is currently unknown; it may become known later.

LINECALLPARTYID_UNAVAIL

Caller ID information is unavailable and will not become known later.

dwCallStates

This field describes the various call states that can possibly be reported for calls on this address. This parameter uses the following LINECALLSTATE_ constants:

LINECALLSTATE_IDLE

The call is idle−no call exists.

LINECALLSTATE_OFFERING

The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the offering state does not automatically alert the user; alerting is done by the switch instructing the line to ring. It does not affect any call states.

LINECALLSTATE_ACCEPTED

The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.

LINECALLSTATE_DIALTONE

The call is receiving 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 being sent to the switch over the call. Note that the operation **lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK

The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY

The call is receiving a busy tone. Busy tone indicates that the call cannot be completed−either a

circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO

Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED

The call has been established and the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING

Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD

The call is on hold by the switch.

LINECALLSTATE_CONFERENCED

The call is currently a member of a multiparty conference call.

LINECALLSTATE_ONHOLDPENDCONF

The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENDTRANSF

The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED

The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN

The state of the call is not known. This may be due to limitations of the call progress detection implementation.

dwDialToneModes

This field describes the various dial tone modes that can possibly be reported for calls made on this address. This field is meaningful only if the *dialtone* call state can be reported. It uses the following LINEDIALTONEMODE_ constants:

LINEDIALTONEMODE_NORMAL

This is a "normal" dial tone, which typically is a continuous tone.

LINEDIALTONEMODE_SPECIAL

This is a special dial tone indicating a certain condition is currently in effect.

LINEDIALTONEMODE_INTERNAL

This is an internal dial tone, as within a PBX.

LINEDIALTONEMODE_EXTERNAL

This is an external (public network) dial tone.

LINEDIALTONEMODE_UNKNOWN

The dial tone mode is currently unknown but may become known later.

LINEDIALTONEMODE_UNAVAIL

The dial tone mode is unavailable and will not become known.

wBusyModes

This field describes the various busy modes that can possibly be reported for calls made on this address. This field is meaningful only if the *busy* call state can be reported. It uses the following LINEBUSYMODE_ constants:

LINEBUSYMODE_STATION

The busy signal indicates that the called party's station is busy. This is usually signaled with a "normal" busy tone.

LINEBUSYMODE_TRUNK

The busy signal indicates that a trunk or circuit is busy. This is usually signaled with a "long" busy tone.

LINEBUSYMODE_UNKNOWN

The busy signal's specific mode is currently unknown but may become known later.

LINEBUSYMODE_UNAVAIL

The busy signal's specific mode is unavailable and will not become known.

dwSpecialInfo

This field describes the various special information types that can possibly be reported for calls made on this address. This field is meaningful only if the *specialInfo* call state can be reported. It uses the following LINESPECIALINFO_ constants:

LINESPECIALINFO_NOCIRCUIT

This special information tone precedes a no-circuit or emergency announcement (trunk blockage category).

LINESPECIALINFO_CUSTIRREG

This special information tone precedes a vacant number, AIS, Centrex number change and nonworking station, access code not dialed or dialed in error, manual intercept operator message (customer irregularity category).

LINESPECIALINFO_REORDER

This special information tone precedes a reorder announcement (equipment irregularity category).

LINESPECIALINFO_UNKNOWN

Specifics about the special information tone are currently unknown but may become known later.

LINESPECIALINFO_UNAVAIL

Specifics about the special information tone are unavailable and will not become known.

dwDisconnectModes

This field describes the various disconnect modes that can possibly be reported for calls made on this address. This field is meaningful only if the *disconnected* call state can be reported. It uses the following LINEDISCONNECTMODE_ constants:

LINEDISCONNECTMODE_NORMAL

This is a "normal" disconnect request by the remote party; the call was terminated normally.

LINEDISCONNECTMODE_UNKNOWN

The reason for the disconnect request is unknown.

LINEDISCONNECTMODE_REJECT

The remote user has rejected the call.

LINEDISCONNECTMODE_PICKUP

The call was picked up from elsewhere.

LINEDISCONNECTMODE_FORWARDED

The call was forwarded by the switch.

LINEDISCONNECTMODE_BUSY

The remote user's station is busy.

LINEDISCONNECTMODE_NOANSWER

The remote user's station does not answer.

LINEDISCONNECTMODE_NODIALTONE

A dial tone was not detected within a service-provider defined timeout, at a point during dialing when one was expected (such as at a "W" in the dialable string). This can also occur without a service-provider-defined timeout period or without a value specified in the **dwWaitForDialTone** member of the **LINEDIALPARAMS** structure.

LINEDISCONNECTMODE_BADADDRESS

The destination address in invalid.

LINEDISCONNECTMODE_UNREACHABLE

The remote user could not be reached.

LINEDISCONNECTMODE_CONGESTION

The network is congested.

LINEDISCONNECTMODE_INCOMPATIBLE

The remote user's station equipment is incompatible with the type of call requested.

LINEDISCONNECTMODE_UNAVAIL

The reason for the disconnect is unavailable and will not become known later.

dwMaxNumActiveCalls

This field contains the maximum number of active call appearances that the address can handle. This number does not include calls on hold or calls on hold pending transfer or conference.

dwMaxNumOnHoldCalls

This field contains the maximum number of call appearances at the address that can be on hold.

dwMaxNumOnHoldPendingCalls

This field contains the maximum number of call appearances at the address that can be on hold pending transfer or conference.

dwMaxNumConference

This field contains the maximum number of parties that can be conferenced in a single conference call on this address.

dwMaxNumTransConf

This field specifies the number of parties (including "self") that can be added in a conference call that is initiated as a generic consultation call using **lineSetupTransfer**.

dwAddrCapFlags

This field contains a series of packed bit flags that describe a variety of address capabilities. It uses the following LINEADDRCAPFLAGS_ constants:

LINEADDRCAPFLAGS_FWDNUMRINGS

Specifies whether the number of rings for a "no answer" can be specified when forwarding calls on no answer.

LINEADDRCAPFLAGS_PICKUPGROUPID

Specifies whether or not a group ID is required for call pickup.

LINEADDRCAPFLAGS_SECURE

Specifies whether or not calls on this address can be made secure at call-setup time.

LINEADDRCAPFLAGS_BLOCKIDDEFAULT

Specifies whether the network by default sends or blocks caller ID information when making a call on this address. If TRUE, ID information is blocked by default; if FALSE, ID information is transmitted by default.

LINEADDRCAPFLAGS_BLOCKIDOVERRIDE

Specifies whether the default setting for sending or blocking of caller ID information can be overridden per call. If TRUE, override is possible; if FALSE, override is not possible.

LINEADDRCAPFLAGS_DIALED

Specifies whether a destination address can be dialed on this address for making a call. TRUE if a destination address must be dialed; FALSE if the destination address is fixed (as with a "hot phone").

LINEADDRCAPFLAGS_ORIGOFFHOOK

Specifies whether the originating party's phone can automatically be taken offhook when making calls.

LINEADDRCAPFLAGS_DESTOFFHOOK

Specifies whether the called party's phone can automatically be forced offhook when making calls.

LINEADDRCAPFLAGS_FWDCONSULT

Specifies whether call forwarding involves the establishment of a consultation call.

LINEADDRCAPFLAGS_SETUPCONFNULL

Specifies whether setting up a conference call starts out with an initial call (FALSE) or with no initial call (TRUE).

LINEADDRCAPFLAGS_AUTORECONNECT

Specifies whether dropping a consultation call automatically reconnects to the call on consultation hold. TRUE if reconnect happens automatically; otherwise, FALSE.

LINEADDRCAPFLAGS_COMPLETIONID

Specifies whether the completion IDs returned by **lineCompleteCall** are useful and unique. TRUE is useful; otherwise, FALSE.

LINEADDRCAPFLAGS_TRANSFERHELD

Specifies whether a (hard) held call can be transferred. Often, only calls on consultation hold may be able to be transferred.

LINEADDRCAPFLAGS_CONFERENCEHELD

Specifies whether a (hard) held call can be conferenced to. Often, only calls on consultation hold

may be able to be added to as a conference call.

LINEADDRCAPFLAGS_ PARTIALDIAL

Specifies whether partial dialing is available.

LINEADDRCAPFLAGS_FWDSTATUSVALID

Specifies whether the forwarding status in the **LINEADDRESSSTATUS** structure for this address is valid.

LINEADDRCAPFLAGS_ FWDINTEXTADDR

Specifies whether internal and external calls can be forwarded to different forwarding addresses. This flag is meaningful only if forwarding of internal and external calls can be controlled separately. It is TRUE if internal and external calls can be forwarded to different destination addresses; otherwise, FALSE.

LINEADDRCAPFLAGS_ FWDBUSYNAADDR

Specifies whether call forwarding for busy and for no answer can use different forwarding addresses. This flag is meaningful only if forwarding for busy and for no answer can be controlled separately. It is TRUE if forwarding for busy and for no answer can use different destination addresses; otherwise, FALSE.

LINEADDRCAPFLAGS_ACCEPTTOALERT

TRUE if an offering call must be accepted using **lineAccept** to start alerting the users at both ends of the call; otherwise, FALSE. Typically, this is only used with ISDN.

LINEADDRCAPFLAGS_CONFDROP

TRUE if invoking **lineDrop** on a conference call parent also has the side effect of dropping (disconnecting) the other parties involved in the conference call; FALSE if dropping a conference call still allows the other parties to talk among themselves.

LINEADDRCAPFLAGS_ PICKUPCALLWAIT

TRUE if **linePickup** can be used to pick up a call detected by the user as a "call waiting" call; otherwise FALSE.

dwCallFeatures

This field specifies the switching capabilities or features available for all calls on this address using the LINECALLFEATURE_ constants. This member represents the call-related features which may possibly be available on an address (static availability as opposed to dynamic availability). Invoking a supported feature requires the call to be in the proper state and the underlying line device to be opened in a compatible mode. A zero in a bit position indicates that the corresponding feature is never available. A one indicates that the corresponding feature may be available if the application has the right privileges to the call and the call is in the appropriate state for the operation to be meaningful. This field allows an application to discover which call features can be (and which can never be) supported by the address.

dwRemoveFromConfCaps

This field specifies the address's capabilities for removing calls from a conference call. It uses the following LINEREMOVEFROMCONF_ constants:

LINEREMOVEFROMCONF_NONE

Parties cannot be removed from the conference call.

LINEREMOVEFROMCONF_LAST

Only the most recently added party can be removed from the conference call.

LINEREMOVEFROMCONF_ANY

Any participating party can be removed from the conference call.

dwRemoveFromConfState

This field uses the LINECALLSTATE_ constants to specify the state of the call after it has been removed from a conference call.

dwTransferModes

This field specifies the address's capabilities for resolving transfer requests. It uses the following LINETRANSFERMODE_ constants:

LINETRANSFERMODE_TRANSFER

Resolve the initiated transfer by transferring the initial call to the consultation call.

LINETRANSFERMODE_CONFERENCE

Resolve the initiated transfer by conferencing all three parties into a three-way conference call. A conference call is created and returned to the application.

dwParkModes

This field specifies the different call park modes available at this address. It uses the LINEPARKMODE_ constants:

LINEPARKMODE_DIRECTED

Specifies directed call park. The address where the call is to be parked must be supplied to the switch.

LINEPARKMODE_NONDIRECTED

Specifies nondirected call park. The address where the call is parked is selected by the switch and provided by the switch to the application.

dwForwardModes

This field specifies the different modes of forwarding available for this address. It 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.

LINEFORWARDMODE_UNCONDINTERNAL

Forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

LINEFORWARDMODE_UNCONDEXTERNAL

Forward all external calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

LINEFORWARDMODE_UNCONDSPECIFIC

Forward all calls that originated at a specified address unconditionally (selective call forwarding).

LINEFORWARDMODE_BUSY

Forward all calls on busy, irrespective of their origin. Use this value when forwarding for internal and external calls 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.

LINEFORWARDMODE_BUSYSPECIFIC

Forward all calls that originated at a specified address on busy (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/no answer, irrespective of their origin. Use this value when forwarding for internal and external calls on busy and on no answer cannot be controlled separately.

LINEFORWARDMODE_BUSYNAINTERNAL

Forward all internal calls on busy/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/no answer. Use this value when call forwarding on busy and on no answer cannot be controlled separately for internal calls.

LINEFORWARDMODE_BUSYNASPECIFIC

Forward all calls that originated at a specified address on busy/no answer (selective call forwarding).

dwMaxForwardEntries

Specifies the maximum number of entries that can be passed to **lineForward** in the *lpForwardList* parameter.

dwMaxSpecificEntries

Specifies the maximum number of entries in the *lpForwardList* parameter passed to **lineForward** that can contain forwarding instructions based on a specific caller ID (selective call forwarding). This field is zero if selective call forwarding is not supported.

dwMinFwdNumRings

Specifies the minimum number of rings that can be set to determine when a call is officially considered "no answer."

dwMaxFwdNumRings

Specifies the maximum number of rings that can be set to determine when a call is officially

considered "no answer." If this number of rings cannot be set, then **dwMinFwdNumRings** and **dwMaxNumRings** will be equal.

dwMaxCallCompletions

Specifies the maximum number of concurrent call completion requests that can be outstanding on this line device. Zero implies that call completion is not available.

dwCallCompletionCond

Specifies the different call conditions under which call completion can be requested. This field uses the following LINECALLCOMPLCOND_ constants:

LINECALLCOMPLCOND_BUSY

Complete the call under the busy condition.

LINECALLCOMPLCOND_NOANSWER

Complete the call under the ringback no answer condition.

dwCallCompletionModes

Specifies the way in which the call can be completed. This field uses the following LINECALLCOMPLCOND_ constants:

LINECALLCOMPLMODE_CAMPON

Queues the call until the call can be completed.

LINECALLCOMPLMODE_CALLBACK

Requests the called station to return the call when it returns to idle.

LINECALLCOMPLMODE_INTRUDE

Adds the application to the existing call at the called station if busy (barge in).

LINECALLCOMPLMODE_MESSAGE

Leave a short predefined message for the called station (Leave Word Calling). A specific message can be identified.

dwNumCompletionMessages

Specifies the number of call completion messages that can be selected from when using the LINECALLCOMPLMODE_MESSAGE option. Individual messages are identified by values in the range 0 through one less than **dwNumCompletionMessages**.

dwCompletionMsgTextEntrySize

Specifies the size in bytes of each of the call completion text descriptions pointed at by **dwCompletionMsgTextSize/Offset**.

dwCompletionMsgTextSize

dwCompletionMsgTextOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing descriptive text about each of the call completion messages. Each message is **dwCompletionMsgTextEntrySize** bytes long. The string format of these textual descriptions is indicated by **dwStringFormat** in the line's device capabilities.

dwAddressFeatures

This field specifies the features available for this address using the LINEADDRFEATURE_ constants. Invoking a supported feature requires the address to be in the proper state and the underlying line device to be opened in a compatible mode. A zero in a bit position indicates that the corresponding

feature is never available. A one indicates that the corresponding feature may be available if the address is in the appropriate state for the operation to be meaningful. This field allows an application to discover which address features can be (and which can never be) supported by the address.

dwPredictiveAutoTransferStates

The call state or states upon which a call made by a predictive dialer can be set to automatically transfer the call to another address; one or more of the LINECALLSTATE_ constants. The value 0 indicates automatic transfer based on call state is unavailable.

dwNumCallTreatments

The number of entries in the array of **LINECALLTREATMENTENTRY** structures delimited by **dwCallTreatmentSize** and **dwCallTreatmentOffset**.

dwCallTreatmentListSize

dwCallTreatmentListOffset

The total size in bytes and offset from the beginning of **LINEADDRESSCAPS** of an array of **LINECALLTREATMENTENTRY** structures, indicating the call treatments supported on the address (which can be selected using **lineSetCallTreatment**). The value will be **dwNumCallTreatments** times SIZEOF (**LINECALLTREATMENTENTRY**).

dwDeviceClassesSize

dwDeviceClassesOffset

Length in bytes and offset from the beginning of **LINEADDRESSCAPS** of a string consisting of the device class identifiers supported on this address for use with **lineGetID**, separated by nulls; the last class indentifier is followed by two nulls.

dwMaxCallDataSize

The maximum number of bytes that an application can set in **LINECALLINFO** using **lineSetCallData**.

dwCallFeatures2

Specifies additional switching capabilities or features available for all calls on this address using the LINECALLFEATURE2_ constants. It is an extension of the **dwCallFeatures** member.

dwMaxNoAnswerTimeout

The maximum value in seconds that can be set in the **dwNoAnswerTimeout** member in **LINECALLPARAMS** when making a call. A value of 0 indicates that automatic abandonment of unanswered calls is not supported by the service provider, or that the timeout value is not adjustable by applications.

dwConnectedModes

Specifies the LINECONNECTEDMODE_ values that may appear in the **dwCallStateMode** member of **LINECALLSTATUS** and in LINE_CALLSTATE messages for calls on this address.

dwOfferingModes

Specifies the LINEOFFERINGMODE_ values that may appear in the **dwCallStateMode** member of **LINECALLSTATUS** and in LINE_CALLSTATE messages for calls on this address.

dwAvailableMediaModes

Indicates the media modes that can be invoked on new calls created on this address, when the **dwAddressFeatures** member indicates that new calls are possible. If this field is zero, it indicates that the service provider either does not know or cannot indicate which media modes are available, in which case any or all of the media modes indicated in the **dwMediaModes** field in **LINEDEVCAPS** may be available.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

Older applications will have been compiled without this field in the **LINEADDRESSCAPS** structure, and using a **SIZEOF LINEADDRESSCAPS** smaller than the new size. The application passes in a *dwAPIVersion* parameter with the **lineGetAddressCaps** function, which can be used for guidance by TAPI in handling this situation. If the application passes in a **dwTotalSize** less than the size of the fixed portion of the structure as defined in the **dwAPIVersion** specified, LINEERR_STRUCTURETOOSMALL will be returned. If sufficient memory has been allocated by the application, before calling **TSPI_lineGetAddressCaps**, TAPI will set the **dwNeededSize** and **dwUsedSize** fields to the fixed size of the structure as it existed in the specified API version.

New service providers (which support the new API version) must examine the API version passed in. If the API version is less than the highest version supported by the provider, the service provider must not fill in fields not supported in older API versions, as these would fall in the variable portion of the older structure.

New applications must be cognizant of the API version negotiated, and not examine the contents of fields in the fixed portion beyond the original end of the fixed portion of the structure for the negotiated API version.

The members **dwPredictiveAutoTransferStates** through **dwAvailableMediaModes** are available only to applications that request an API version of 0x00020000 or greater when calling **lineGetAddressCaps**.

See Also

LINE_ADDRESSSTATE, LINE_CALLINFO, LINE_CALLSTATE, LINE_LINEDEVSTATE, **LINEADDRESSSTATUS**, **LINECALLINFO**, **LINECALLPARAMS**, **LINECALLSTATUS**, **LINECALLTREATMENTENTRY**, **LINEDEVCAPS**, **LINEDIALPARAMS**, **lineCompleteCall**, **lineForward**, **lineGenerateDigits**, **lineGetAddressCaps**, **lineGetID**, **lineSetCallData**, **lineSetCallTreatment**

LINEADDRESSSTATUS Overview

The **LINEADDRESSSTATUS** structure describes the current status of an address.

```
typedef struct lineaddressstatus_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwNumInUse;
    DWORD  dwNumActiveCalls;
    DWORD  dwNumOnHoldCalls;
    DWORD  dwNumOnHoldPendCalls;
    DWORD  dwAddressFeatures;

    DWORD  dwNumRingsNoAnswer;
    DWORD  dwForwardNumEntries;
    DWORD  dwForwardSize;
    DWORD  dwForwardOffset;

    DWORD  dwTerminalModesSize;
    DWORD  dwTerminalModesOffset;

    DWORD  dwDevSpecificSize;
    DWORD  dwDevSpecificOffset;
} LINEADDRESSSTATUS, FAR *LPLINEADDRESSSTATUS;
```

Members

dwTotalSize

The total size in bytes 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.

dwNumInUse

Specifies the number of stations that are currently using the address.

dwNumActiveCalls

The number of calls on the address that are in call states other than *idle*, *onhold*, *onholdpendingtransfer*, and *onholdpendingconference*.

dwNumOnHoldCalls

The number of calls on the address in the *onhold* state.

dwNumOnHoldPendCalls

The number of calls on the address in the *onholdpendingtransfer* or *onholdpendingconference* state.

dwAddressFeatures

This field specifies the address-related API functions that can be invoked on the address in its current state. It uses the following LINEADDRFEATURE_ constants:

LINEADDRFEATURE_FORWARD

The address can be forwarded.

LINEADDRFEATURE_MAKECALL

An outbound call can be placed on the address.

LINEADDRFEATURE_PICKUP

A call can be picked up at the address.

LINEADDRFEATURE_SETMEDIACONTROL

Media control can be set on this address.

LINEADDRFEATURE_SETTERMINAL

The terminal modes for this address can be set.

LINEADDRFEATURE_SETUPCONF

A conference call with a NULL initial call can be set up at this address.

LINEADDRFEATURE_UNCOMPLETECALL

Call completion requests can be canceled at this address.

LINEADDRFEATURE_UNPARK

Calls can be unparked using this address.

dwNumRingsNoAnswer

The number of rings set for this address before an unanswered call is considered as no answer.

dwForwardNumEntries

The number of entries in the array referred to by **dwForwardSize** and **dwForwardOffset**.

dwForwardSize

dwForwardOffset

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's forwarding information. This information is an array of **dwForwardNumEntries** elements, of type **LINEFORWARD**. The offsets of the addresses in the array are relative to the beginning of the **LINEADDRESSSTATUS** structure. The offsets **dwCallerAddressOffset** and **dwDestAddressOffset** in the variably sized field of type LINEFORWARD pointed to by **dwForwardSize** and **dwForwardOffset** are relative to the beginning of the **LINEADDRESSSTATUS** data structure (the "root" container).

dwTerminalModesSize

dwTerminalModesOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing an array with DWORD-sized entries, that use the LINETERMMODE_ constants. This array is indexed by terminal IDs, in the range from zero to one less than **dwNumTerminals**. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the **lineSetTerminal** function for this address. Values are:

LINETERMMODE_LAMPS

These are lamp events sent from the line to the terminal.

LINETERMMODE_BUTTONS

These are button-press events sent from the terminal to the line.

LINETERMMODE_DISPLAY

This is display information sent from the line to the terminal.

LINETERMMODE_RINGER

This is ringer-control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH

These are hookswitch events sent between the terminal and the line.

LINETERMMODE_MEDIATOLINE

This is the unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE

This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT

This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when the routing of both unidirectional channels of a call's media stream cannot be controlled independently.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

This data structure is returned by **lineGetAddressStatus**. When items in this data structure change as a consequence of activities on the address, a LINE_ADDRESSSTATE message is sent to the application. A parameter to this message is the address state, the constant LINEADDRESSSTATE_, which is an indication that the status item in this record changed.

See Also

LINE_ADDRESSSTATE, **LINEFORWARD**, **lineGetAddressStatus**, **lineSetTerminal**

LINEAGENTACTIVITYENTRY Overview

```
typedef struct lineagentactivityentry_tag {
    DWORD  dwID;
    DWORD  dwNameSize;
    DWORD  dwNameOffset;
} LINEAGENTACTIVITYENTRY, FAR *LPLINEAGENTACTIVITYENTRY;
```

Members

dwID

A unique identifier for a activity. It is the responsibility of the agent handler to generate and maintain uniqueness of these IDs.

dwNameSize

dwNameOffset

Size in bytes and offset from the beginning of the containing structure of a null-terminated string specifying the name and other identifying information of an activity which can be selected using **lineSetAgentActivity**.

LINEAGENTACTIVITYLIST Overview

```
typedef struct lineagentactivitylist_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;
    DWORD  dwNumEntries;
    DWORD  dwListSize;
    DWORD  dwListOffset;
} LINEAGENTACTIVITYLIST, FAR *LPLINEAGENTACTIVITYLIST;
```

Members

dwNumEntries

The number of **LINEAGENTACTIVITYENTRY** structures that appear in the *List* array. The value is 0 if no agent activity codes are available.

dwListSize

dwListOffset

Total size in bytes and offset from the beginning of **LINEAGENTACTIVITYLIST** of an array of **LINEAGENTACTIVITYENTRY** elements indicating information about activity which could be specified for the current logged-in agent. This will be **dwNumEntries** times SIZEOF (**LINEAGENTACTIVITYENTRY**).

LINEAGENTCAPS Overview

```
typedef struct lineagentcaps_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwAgentHandlerInfoSize;
    DWORD  dwAgentHandlerInfoOffset;
    DWORD  dwCapsVersion;

    DWORD  dwFeatures;
    DWORD  dwStates;
    DWORD  dwNextStates;
    DWORD  dwMaxNumGroupEntries;
    DWORD  dwAgentStatusMessages;
    DWORD  dwNumAgentExtensionIDs;
    DWORD  dwAgentExtensionIDListSize;
    DWORD  dwAgentExtensionIDListOffset;
} LINEAGENTCAPS, FAR *LPLINEAGENTCAPS;
```

Members

dwAgentHandlerInfoSize

dwAgentHandlerInfoOffset

The size in bytes and offset from the beginning of **LINEAGENTCAPS** of a null-terminated string specifying the name, version, or other identifying information of the server application that is handling agent requests.

dwCapsVersion

The TAPI version that the agent handler application used in preparing the contents of this structure. This will be no greater than the API version that the calling application passed in to **lineGetAgentCaps**.

dwFeatures

The agent-related features available for this line using the LINEAGENTFEATURE_ constants. Invoking a supported feature requires the line and address to be in the proper state. A zero in a bit position indicates that the corresponding feature is never available. A one indicates that the corresponding feature may be available if the line is in the appropriate state for the operation to be meaningful. This field allows an application to discover which agent features can be (and which can never be) supported by the device.

dwStates

The LINEAGENTSTATE_ values which may be used in the *dwAgentState* parameter of **lineSetAgentState**. Setting a supported state requires the line and address to be in the proper state. A zero in a bit position indicates that the corresponding state is never available. A one indicates that the corresponding state may be available if the line is in the appropriate state for the state to be meaningful. This field allows an application to discover which agent states can be (and which can never be) supported on the device.

dwNextStates

The LINEAGENTSTATE_ values which may be used in the *dwNextAgentState* parameter of **lineSetAgentState**. Setting a supported state requires the line and address to be in the proper state.

A zero in a bit position indicates that the corresponding state is never available. A one indicates that the corresponding state may be available if the line is in the appropriate state for the state to be meaningful. This field allows an application to discover which agent states can be (and which can never be) supported on the device.

dwMaxNumGroupEntries

The maximum number of agent IDs that can be logged in on the address simultaneously. Determines the highest value that can be passed in as the **dwNumEntries** field in the **LINEAGENTGROUPLIST** structure to **lineSetAgentGroup**.

dwAgentStatusMessages

Indicates the LINEAGENTSTATUS_ constants that can be received by the application in *dwParam2* of a LINE_AGENTSTATUS message.

dwNumExtensionIDs

The number of **LINEEXTENSIONID** structures that appear in the *ExtensionIDList* array. The value is 0 if agent-handler-specific extensions are supported on the address.

dwExtensionIDListSize

dwExtensionIDListOffset

Total size in bytes and offset from the beginning of **LINEAGENTCAPS** of an array of **LINEEXTENSIONID** elements. The size will be **dwNumExtensionIDs** times **SIZEOF** (**LINEEXTENSIONID**). The array lists the 128-bit universally unique identifiers for all agent-handler-specific extensions supported by the agent handle for the address. The extension being used is referenced in the **lineAgentSpecific** function and LINE_AGENTSPECIFIC message by its position in this table, the first entry being entry 0, so it is important that the agent handler always present extension IDs in this array in the same order.

See Also

LINE_AGENTSPECIFIC, LINE_AGENTSTATUS, **LINEAGENTGROUPLIST**, **lineAgentSpecific**, **LINEEXTENSIONID**, **lineGetAgentCaps**, **lineSetAgentGroup**, **lineSetAgentState**

LINEAGENTGROUPENTRY Overview

```
typedef struct lineagentgroupentry_tag {
  struct {
    DWORD  dwGroupID1;
    DWORD  dwGroupID2;
    DWORD  dwGroupID3;
    DWORD  dwGroupID4;
  } GroupID;
  DWORD  dwNameSize;
  DWORD  dwNameOffset;
} LINEAGENTGROUPENTRY, FAR *LPLINEAGENTGROUPENTRY;
```

Members

GroupID

This set of four DWORDs is a universally unique identifier for a group. It is the responsibility of the agent handler to generate and maintain uniqueness of these IDs.

dwNameSize

dwNameOffset

Size in bytes and offset from the beginning of the containing structure of a null-terminated string specifying the name and other identifying information of an ACD group or queue into which the agent can log in. This string can contain such information as supervisor and skill level, to assist the agent in selecting the correct group from a list displayed on their workstation screen.

LINEAGENTGROUPLIST Overview

```
typedef struct lineagentgrouplist_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;
    DWORD  dwNumEntries;
    DWORD  dwListSize;
    DWORD  dwListOffset;
} LINEAGENTGROUPLIST, FAR *LPLINEAGENTGROUPLIST;
```

Members

dwNumEntries

The number of **LINEAGENTGROUPENTRY** structures that appear in the *List* array. The value is 0 if no agent is to be logged in on the address.

dwListSize

dwListOffset

Total size in bytes and offset from the beginning of **LINEAGENTGROUPLIST** of an array of **LINEAGENTGROUPENTRY** elements specifying information about each group into which the current agent is to be logged in at the address. This will be **dwNumEntries** times **SIZEOF** (**LINEAGENTGROUPENTRY**).

LINEAGENTSTATUS Overview

```
typedef struct lineagentstatus_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwNumEntries;
    DWORD  dwGroupListSize;
    DWORD  dwGroupListOffset;

    DWORD  dwState;
    DWORD  dwNextState;
    DWORD  dwActivityID;
    DWORD  dwActivitySize;
    DWORD  dwActivityOffset;

    DWORD  dwAgentFeatures;
    DWORD  dwValidStates;
    DWORD  dwValidNextStates;
} LINEAGENTSTATUS, FAR *LPLINEAGENTSTATUS;
```

Members

dwNumEntries

The number of **LINEAGENTGROUPENTRY** structures that appear in the *GroupList* array. The value is 0 if no agent is logged in on the address.

dwGroupListSize

dwGroupListOffset

Total size in bytes and offset from the beginning of **LINEAGENTSTATUS** of an array of **LINEAGENTGROUPENTRY** elements. The size will be **dwNumEntries** times **SIZEOF** (**LINEAGENTGROUPENTRY**). The array contains groups into which the agent is currently logged in on the address.

dwState

The current state of the agent. One of the LINEAGENTSTATE_ constants.

dwNextState

The state into which the agent will automatically be placed when the current call completes. One of the LINEAGENTSTATE_ constants.

dwActivityID

The ID of the current agent activity.

dwActivitySize

dwActivityOffset

Size in bytes and offset from the beginning of **LINEAGENTSTATUS** of a null-terminated string specifying the current agent activity.

dwAgentFeatures

The agent-related features available at the time the status was obtained, using the

LINEAGENTFEATURE_ constants.

dwValidStates

The agent states which could be selected, at this point in time, using **lineSetAgentState**. Consists of one or more of the LINEAGENTSTATE_ constants.

dwValidNextStates

The next agent states which could be selected, at this point in time, using **lineSetAgentState**. Consists of one or more of the LINEAGENTSTATE_ constants.

See Also

LINEAGENTGROUPENTRY, **lineSetAgentState**

LINEAPPINFO Overview

```
typedef struct lineappinfo_tag {
} LINEAPPINFO, FAR *LPLINEAPPINFO;
    DWORD  dwMachineNameSize;
    DWORD  dwMachineNameOffset;
    DWORD  dwUserNameSize;
    DWORD  dwUserNameOffset;
    DWORD  dwModuleFilenameSize;
    DWORD  dwModuleFilenameOffset;
    DWORD  dwFriendlyNameSize;
    DWORD  dwFriendlyNameOffset;
    DWORD  dwMediaModes;
    DWORD  dwAddressID;
```

Members

dwMachineNameSize

dwMachineNameOffset

Size in bytes and offset from the beginning of **LINEDEVSTATUS** of a string specifying the name of the computer on which the application is executing.

dwUserNameSize

dwUserNameOffset

Size in bytes and offset from the beginning of **LINEDEVSTATUS** of a string specifying the username under whose account the application is running.

dwModuleFilenameSize

dwModuleFilenameOffset

Size in bytes and offset from the beginning of **LINEDEVSTATUS** of a string specifying the module filename of the application. This string may be used in a call to **lineHandoff** to perform a directed handoff to the application.

dwFriendlyNameSize

dwFriendlyNameOffset

Size in bytes and offset from the beginning of **LINEDEVSTATUS** of the string provided by the application to **lineInitialize** or **lineInitializeEx**, which should be used in any display of applications to the user.

dwMediaModes

The media modes for which the application has requested ownership of new calls; 0 if when it opened the line **dwPrivileges** did not include LINECALLPRIVILEGE_OWNER.

dwAddressID

If the line handle was opened using LINEOPENOPTION_SINGLEADDRESS, contains the address ID specified; set to 0xFFFFFFFF if the single address option was not used.

LINECALLINFO Overview

The **LINECALLINFO** structure contains information about a call. This information remains relatively fixed for the duration of the call and is obtained with **lineGetCallInfo**. If a part of the structure does change, then a LINE_CALLINFO message is sent to the application indicating which information item has changed. Dynamically changing information about a call, such as call progress status, is available in the **LINECALLSTATUS** structure, returned with the function **lineGetCallStatus**.

```
typedef struct linecallinfo_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

  HLINE  hLine;
    DWORD  dwLineDeviceID;
    DWORD  dwAddressID;

    DWORD  dwBearerMode;
    DWORD  dwRate;
    DWORD  dwMediaMode;

    DWORD  dwAppSpecific;
    DWORD  dwCallID;
    DWORD  dwRelatedCallID;
    DWORD  dwCallParamFlags;
    DWORD  dwCallStates;

    DWORD  dwMonitorDigitModes;
    DWORD  dwMonitorMediaModes;
  LINEDIALPARAMS  DialParams;

    DWORD  dwOrigin;
    DWORD  dwReason;
    DWORD  dwCompletionID;
    DWORD  dwNumOwners;
    DWORD  dwNumMonitors;

    DWORD  dwCountryCode;
    DWORD  dwTrunk;

    DWORD  dwCallerIDFlags;
    DWORD  dwCallerIDSize;
    DWORD  dwCallerIDOffset;
    DWORD  dwCallerIDNameSize;
    DWORD  dwCallerIDNameOffset;

    DWORD  dwCalledIDFlags;
    DWORD  dwCalledIDSize;
    DWORD  dwCalledIDOffset;
    DWORD  dwCalledIDNameSize;
    DWORD  dwCalledIDNameOffset;

    DWORD  dwConnectedIDFlags;
    DWORD  dwConnectedIDSize;
```

```
DWORD  dwConnectedIDOffset;
DWORD  dwConnectedIDNameSize;
DWORD  dwConnectedIDNameOffset;

DWORD  dwRedirectionIDFlags;
DWORD  dwRedirectionIDSize;
DWORD  dwRedirectionIDOffset;
DWORD  dwRedirectionIDNameSize;
DWORD  dwRedirectionIDNameOffset;

DWORD  dwRedirectingIDFlags;
DWORD  dwRedirectingIDSize;
DWORD  dwRedirectingIDOffset;
DWORD  dwRedirectingIDNameSize;
DWORD  dwRedirectingIDNameOffset;

DWORD  dwAppNameSize;
DWORD  dwAppNameOffset;
DWORD  dwDisplayableAddressSize;
DWORD  dwDisplayableAddressOffset;

DWORD  dwCalledPartySize;
DWORD  dwCalledPartyOffset;

DWORD  dwCommentSize;
DWORD  dwCommentOffset;

DWORD  dwDisplaySize;
DWORD  dwDisplayOffset;

DWORD  dwUserUserInfoSize;
DWORD  dwUserUserInfoOffset;

DWORD  dwHighLevelCompSize;
DWORD  dwHighLevelCompOffset;

DWORD  dwLowLevelCompSize;
DWORD  dwLowLevelCompOffset;

DWORD  dwChargingInfoSize;
DWORD  dwChargingInfoOffset;

DWORD  dwTerminalModesSize;
DWORD  dwTerminalModesOffset;

DWORD  dwDevSpecificSize;
DWORD  dwDevSpecificOffset;

DWORD  dwCallTreatment;
DWORD  dwCallDataSize;
DWORD  dwCallDataOffset;
DWORD  dwSendingFlowspecSize;
DWORD  dwSendingFlowspecOffset;
DWORD  dwReceivingFlowspecSize;
DWORD  dwReceivingFlowspecOffset;
```

```
} LINECALLINFO, FAR *LPLINECALLINFO;
```

Members

dwTotalSize

The total size in bytes 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.

hLine

The handle for the line device with which this call is associated.

dwLineDeviceID

The device ID of the line device with which this call is associated.

dwAddressID

The address ID of the address on the line on which this call exists.

dwBearerMode

The current bearer mode of the call. This field uses the following LINEBEARERMODE_ constants:

LINEBEARERMODE_VOICE

This is a regular 3.1 kHz analog voice grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH

This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation, and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE

The multiuse mode defined by ISDN.

LINEBEARERMODE_DATA

The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA

The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING

A non-call-associated signaling connection from the application to the service provider or switch (treated as a "media stream" by TAPI).

LINEBEARERMODE_PASSTHROUGH

When a call is active in LINEBEARERMODE_PASSTHROUGH, the service provider gives direct access to the attached hardware for control by the application. This mode is used primarily by applications desiring temporary direct control over asynchronous modems, accessed through the Win32 comm functions, for the purpose of configuring or using special features not otherwise supported by the service provider.

dwRate

The rate of the call's data stream in bps (bits per second).

dwMediaMode

Specifies the media mode of the information stream currently on the call. This is the media mode as determined by the owner of the call, which is not necessarily the same as that of the last LINE_MONITORMEDIA message. This field is not directly affected by the LINE_MONITORMEDIA messages. It uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN

A media stream exists but its mode is not known. This corresponds to a call with an unclassified media type. In typical analog telephony environments, an inbound call's media mode may be unknown until after the call has been answered and the media stream has been filtered to make a determination.

LINEMEDIAMODE_INTERACTIVEVOICE

The presence of voice energy is detected on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE

The presence of voice energy is detected on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

A data modem session is detected on the call.

LINEMEDIAMODE_G3FAX

A group 3 fax is being sent or received on the call.

LINEMEDIAMODE_G4FAX

A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_TDD

A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_DIGITALDATA

Digital data being sent or received over the call.

LINEMEDIAMODE_TELETEX

A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX

A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX

A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED

A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI

An ADSI (Analog Display Services Interface) session on the call.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

dwAppSpecific

This field is uninterpreted by the API implementation and service provider. It can be set by any owner application of this call with the operation **lineSetAppSpecific**.

dwCallID

In some telephony environments, the switch or service provider may 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** field makes this unique identifier available to the applications.

dwRelatedCallID

Telephony environments that use the call ID often may find it necessary to relate one call to another. The **dwRelatedCallID** field may be used by the service provider for this purpose.

dwCallParamFlags

A collection of call-related parameters when the call is outbound. These are same call parameters specified in **lineMakeCall**, of type LINECALLPARAMFLAGS_. Values are:

LINECALLPARAMFLAGS_SECURE

The call is currently secure. This flag is also updated if the call is later secured through **lineSecureCall**.

LINECALLPARAMFLAGS_IDLE

The call started out using an idle call.

LINECALLPARAMFLAGS_BLOCKID

The originator identity was concealed (block caller ID presentation to the remote party).

LINECALLPARAMFLAGS_ORIGOFFHOOK

The originator's phone was automatically taken offhook.

LINECALLPARAMFLAGS_DESTOFFHOOK

The called party's phone was automatically taken offhook.

dwCallStates

The call states for which the application may be notified on this call, of type LINECALLSTATE_. The **dwCallStates** member is constant in **LINECALLINFO** and does not change depending on the call state. Values are:

LINECALLSTATE_IDLE

The call is idle−no call exists.

LINECALLSTATE_OFFERING

The call is being offered to the station signaling the arrival of a new call. In some environments, a call in the offering state does not automatically alert the user; alerting is done by the switch instructing the line to ring. It does not affect any call states.

LINECALLSTATE_ACCEPTED

The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.

LINECALLSTATE_DIALTONE

The call is receiving 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 being sent to the switch over the call. Note that the operation **lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK

The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY

The call is receiving a busy tone. Busy tone indicates that the call cannot be completed—either a circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO

Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED

The call has been established, the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING

Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD

The call is on hold by the switch.

LINECALLSTATE_CONFERENCED

The call is currently a member of a multiparty conference call.

LINECALLSTATE_ONHOLDPENDCONF

The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENDTRANSF

The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED

The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN

The state of the call is not known. This may be due to limitations of the call progress detection implementation.

dwMonitorDigitsModes

The various digit modes for which monitoring is currently enabled, of type LINEDIGITMODE_. Values are:

LINEDIGITMODE_PULSE

Uses pulse/rotary for digit signaling.

LINEDIGITMODE_DTMF

Uses DTMF tones for digit signaling.

LINEDIGITMODE_DTMFEND

Uses DTMF tones for digit detection, and also detects the down edges.

dwMonitorMediaModes

The various media modes for which monitoring is currently enabled, of type LINEMEDIAMODE_. Values are:

LINEMEDIAMODE_INTERACTIVEVOICE

The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE

The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

A data modem session on the call.

LINEMEDIAMODE_G3FAX

A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_G4FAX

A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_TDD

A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_DIGITALDATA

Digital data is being sent or received over the call.

LINEMEDIAMODE_TELETEX

A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX

A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX

A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED

A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI

An ADSI (Analog Display Services Interface) session on the call.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

DialParams

The dialing parameters currently in effect on the call, of type **LINEDIALPARAMS**. Unless these parameters are set by either **lineMakeCall** or **lineSetCallParams**, their values will be the same as the defaults used in the **LINEDEVCAPS**.

dwOrigin

Identifies where the call originated from. This field uses the following LINECALLORIGIN_ constants:

LINECALLORIGIN_OUTBOUND

The call is an outbound call.

LINECALLORIGIN_INTERNAL

The call is inbound and originated internally (on the same PBX, for example).

LINECALLORIGIN_EXTERNAL

The call is inbound and originated externally.

LINECALLORIGIN_UNKNOWN

The call is an inbound call and its origin is currently unknown but may become known later.

LINECALLORIGIN_UNAVAIL

The call is an inbound call. Its origin is not available and will never become known for this call.

LINECALLORIGIN_CONFERENCE

The call handle is for a conference call, that is, the application's connection to the conference bridge in the switch.

dwReason

The reason why the call occurred. This field uses the following LINECALLREASON_ constants:

LINECALLREASON_DIRECT

This is a direct call.

LINECALLREASON_FWDBUSY

This call was forwarded from another extension that was busy at the time of the call.

LINECALLREASON_FWDNOANSWER

The call was forwarded from another extension that didn't answer the call after some number of rings.

LINECALLREASON_FWDUNCOND

The call was forwarded unconditionally from another number.

LINECALLREASON_PICKUP

The call was picked up from another extension.

LINECALLREASON_UNPARK

The call was retrieved as a parked call.

LINECALLREASON_REDIRECT

The call was redirected to this station.

LINECALLREASON_CALLCOMPLETION

The call was the result of a call completion request.

LINECALLREASON_TRANSFER

The call has been transferred from another number. Party ID information may indicate who the caller is and where the call was transferred from.

LINECALLREASON_REMINDER

The call is a reminder (or "recall") that the user has a call parked or on hold for potentially a long time.

LINECALLREASON_UNKNOWN

The reason for the call is currently unknown but may become known later.

LINECALLREASON_UNAVAIL

The reason for the call is unavailable and will not become known later.

dwCompletionID

The completion ID for the incoming call if it is the result of a completion request that terminates. This ID is meaningful only if **dwReason** is LINECALLREASON_CALLCOMPLETION.

dwNumOwners

The number of application modules with different call handles with owner privilege for the call.

dwNumMonitors

The number of application modules with different call handles with monitor privilege for the call.

dwCountryCode

The country code of the destination party. Zero if unknown.

dwTrunk

The number of the trunk over which the call is routed. This field is used for both inbound and outgoing calls. The **dwTrunk** field should be set to 0xFFFFFFFF if it is unknown.

dwCallerIDFlags

Determines the validity and content of the caller party ID information. The caller is the originator of the call. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED

Caller ID information for the call has been blocked by the caller but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA

Caller ID information for the call is not available because it is not propagated all the way by the network.

LINECALLPARTYID_NAME

The caller ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the caller ID name variably sized field.

LINECALLPARTYID_ADDRESS

The caller ID information for the call is the caller's number and is provided in the caller ID variably sized field.

LINECALLPARTYID_PARTIAL

Caller ID information for the call is valid but is limited to partial number information.

LINECALLPARTYID_UNKNOWN

Caller ID information is currently unknown but it may become known later.

LINECALLPARTYID_UNAVAIL

Caller ID information is unavailable and will not become known later.

dwCallerIDSize

dwCallerIDOffset

The size in bytes of the variably sized field containing the caller party ID number information, and the offset in bytes from the beginning of this data structure.

dwCallerIDNameSize

dwCallerIDNameOffset

The size in bytes of the variably sized field containing the caller party ID name information, and the offset in bytes from the beginning of this data structure.

dwCalledIDFlags

Determines the validity and content of the called-party ID information. The called party corresponds to the originally addressed party. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED

Called ID information for the call has been blocked by the caller but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA

Caller ID information for the call is not available because it is not propagated all the way by the network.

LINECALLPARTYID_NAME

The called ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the called ID name variably sized field.

LINECALLPARTYID_ADDRESS

The called ID information for the call is the caller's number and is provided in the called ID variably sized field.

LINECALLPARTYID_PARTIAL

Called ID information for the call is valid but is limited to partial number information.

LINECALLPARTYID_UNKNOWN

Called ID information is currently unknown but it may become known later.

LINECALLPARTYID_UNAVAIL

Called ID information is unavailable and will not become known later.

dwCalledIDSize

dwCalledIDOffset

The size in bytes of the variably sized field containing the called-party ID number information, and the offset in bytes from the beginning of this data structure.

dwCalledIDNameSize

dwCalledIDNameOffset

The size in bytes of the variably sized field containing the called-party ID name information, and the offset in bytes from the beginning of this data structure.

dwConnectedFlags

Determines the validity and content of the connected party ID information. The connected party is the

party that was actually connected to. This may be different from the called-party ID if the call was diverted. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED

Connected party ID information for the call has been blocked by the caller but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA

Connected ID information for the call is not available as it is not propagated all the way by the network.

LINECALLPARTYID_NAME

The connected party ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the connected ID name variably sized field.

LINECALLPARTYID_ADDRESS

The connected party ID information for the call is the caller's number and is provided in the connected ID variably sized field.

LINECALLPARTYID_PARTIAL

Connected party ID information for the call is valid but is limited to partial number information.

LINECALLPARTYID_UNKNOWN

Connected party ID information is currently unknown but it may become known later.

LINECALLPARTYID_UNAVAIL

Connected party ID information is unavailable and will not become known later.

dwConnectedIDSize

dwConnectedIDOffset

The size in bytes of the variably sized field containing the connected party ID number information, and the offset in bytes from the beginning of this data structure.

dwConnectedIDNameSize

dwConnectedIDNameOffset

The size in bytes of the variably sized field containing the connected party ID name information, and the offset in bytes from the beginning of this data structure.

dwRedirectionIDFlags

Determines the validity and content of the redirection party ID information. The redirection party identifies to the calling user the number towards which diversion was invoked. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED

Redirection party ID information for the call has been blocked by the caller but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA

Redirection ID information for the call is not available because it is not propagated all the way by the network.

LINECALLPARTYID_NAME

The redirection party ID information for the call is the caller's name (from a table maintained inside

the switch). It is provided in the redirection ID name variably sized field.

LINECALLPARTYID_ADDRESS

The redirection party ID information for the call is the caller's number and is provided in the redirection ID variably sized field.

LINECALLPARTYID_PARTIAL

Redirection party ID information for the call is valid but is limited to partial number information.

LINECALLPARTYID_UNKNOWN

Redirection ID information is currently unknown but it may become known later.

LINECALLPARTYID_UNAVAIL

Redirection ID information is unavailable and will not become known later.

dwRedirectionIDSize

dwRedirectionIDOffset

The size in bytes of the variably sized field containing the redirection party ID number information, and the offset in bytes from the beginning of this data structure.

dwRedirectionIDNameSize

dwRedirectionIDNameOffset

The size in bytes of the variably sized field containing the redirection party ID name information, and the offset in bytes from the beginning of this data structure.

dwRedirectingIDFlags

Determines the validity and content of the redirecting party ID information. The redirecting party identifies to the diverted-to user the party from which diversion was invoked. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED

Redirecting party ID information for the call has been blocked by the caller but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA

Redirecting ID information for the call is not available because it is not propagated all the way by the network.

LINECALLPARTYID_NAME

The redirecting party ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the redirecting ID name variably sized field.

LINECALLPARTYID_ADDRESS

The redirecting party ID information for the call is the caller's number and is provided in the redirecting ID variably sized field.

LINECALLPARTYID_PARTIAL

Redirecting party ID information for the call is valid but is limited to partial number information.

LINECALLPARTYID_UNKNOWN

Redirecting ID information is currently unknown but it may become known later.

LINECALLPARTYID_UNAVAIL

Redirecting ID information is unavailable and will not become known later.

dwRedirectingIDSize

dwRedirectingIDOffset

The size in bytes of the variably sized field containing the redirecting party ID number information, and the offset in bytes from the beginning of this data structure.

dwRedirectingIDNameSize

dwRedirectingIDNameOffset

The size in bytes of the variably sized field containing the redirecting party ID name information, and the offset in bytes from the beginning of this data structure.

dwAppNameSize

dwAppNameOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field holding the user-friendly application name of the application that first originated, accepted, or answered the call. This is the name that an application can specify in **lineInitializeEx**. If the application specifies no such name, then the application's module file name is used instead.

dwDisplayableAddressSize

dwDisplayableAddressOffset

The displayable string is used for logging purposes. The information is obtained from **LINECALLPARAMS** for functions that initiate calls. The function **lineTranslateAddress** returns appropriate information to be placed in this field in the **dwDisplayableAddressSize** and **dwDisplayableAddressOffset** fields of the **LINETRANSLATEOUTPUT** structure.

dwCalledPartySize

dwCalledPartyOffset

The size in bytes of the variably sized field holding a user-friendly description of the called party, and the offset in bytes from the beginning of this data structure. This information can be specified on **lineMakeCall** and can be optionally specified in the *lpCallParams* whenever a new call is established. It is useful for call logging purposes.

dwCommentSize

dwCommentOffset

The size in bytes of the variably sized field holding a comment about the call provided by the application that originated the call using **lineMakeCall**, and the offset in bytes from the beginning of this data structure. This information can be optionally specified in the *lpCallParams* whenever a new call is established.

dwDisplaySize

dwDisplayOffset

The size in bytes of the variably sized field holding raw display information, and the offset in bytes from the beginning of this data structure. Depending on the telephony environment, a service provider may extract functional information from this for presentation in a more functional way.

dwUserUserInfoSize

dwUserUserInfoOffset

The size in bytes of the variably sized field holding user-to-user information, and the offset in bytes from the beginning of this data structure The protocol discriminator field for the user-to-user information, if used, appears as the first byte of the data pointed to by **dwUserUserInfoOffset**, and is accounted for in **dwUserUserInfoSize**.

dwHighLevelCompSize

dwHighLevelCompOffset

The size in bytes of the variably sized field holding high-level compatibility information, and the offset in bytes from the beginning of this data structure. The format of this information is specified by other standards (ISDN Q.931).

dwLowLevelCompSize

dwLowLevelCompOffset

The size in bytes of the variably sized field holding low-level compatibility information, and the offset in bytes from the beginning of this data structure The format of this information is specified by other standards (ISDN Q.931).

dwChargingInfoSize

dwChargingInfoOffset

The size in bytes of the variably sized field holding charging information, and the offset in bytes from the beginning of this data structure The format of this information is specified by other standards (ISDN Q.931).

dwTerminalModesSize

dwTerminalModesOffset

The size in bytes of the variably sized device field containing an array with DWORD-sized entries, and the offset in bytes from the beginning of this data structure. The set of LINETERMMODE_ constants is indexed by terminal IDs, in the range from zero to one less than **dwNumTerminals**. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the **lineSetTerminal** operation for this call's media stream. Values are:

LINETERMMODE_LAMPS

Lamp events sent from the line to the terminal.

LINETERMMODE_BUTTONS

Button-press events sent from the terminal to the line.

LINETERMMODE_DISPLAY

Display information sent from the line to the terminal.

LINETERMMODE_RINGER

Ringer-control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH

Hookswitch event sent between the terminal and the line.

LINETERMMODE_MEDIATOLINE

The unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE

The unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT

The bidirectional media stream associated with a call on the line and the terminal. Use this value when the routing of both the unidirectional channels of a call's media stream cannot be controlled independently.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized field holding device-specific information., and the offset in bytes from the beginning of this data structure.

dwCallTreatment

The call treatment currently being applied on the call or that will be applied when the call enters the next applicable state. May be 0 if call treatments are not supported.

dwCallDataSize

dwCallDataOffset

The size in bytes and offset from the beginning of **LINECALLINFO** of the application-settable call data.

dwSendingFlowspecSize

dwSendingFlowspecOffset

The total size in bytes and offset from the beginning of **LINECALLINFO** of a WinSock2 **FLOWSPEC** structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in SendingFlowspec.len in a WinSock2 **QOS** structure. Specifies the quality of service current in effect in the sending direction on the call. The provider-specific portion following the **FLOWSPEC** structure must not contain pointers to other blocks of memory, because TAPI will not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.

dwReceivingFlowspecSize

dwReceivingFlowspecOffset

The total size in bytes and offset from the beginning of **LINECALLINFO** of a WinSock2 **FLOWSPEC** structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in ReceivingFlowspec.len in a WinSock2 **QOS** structure. Specifies the quality of service current in effect in the receiving direction on the call. The provider-specific portion following the **FLOWSPEC** structure must not contain pointers to other blocks of memory, because TAPI will not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

The **LINECALLINFO** data structure contains relatively fixed information about a call. This structure is returned with **lineGetCallInfo**. When information items in this data structure have changed, a LINE_CALLINFO message is sent to the application. A parameter to this message is the information item or field that changed.

The **dwAppSpecific** field can be used by applications to tag calls by using **lineSetAppSpecific**. This field is uninterpreted by TAPI or service providers. It is initially set to zero.

The members **dwCallTreatment** through **dwReceivingFlowspecOffset** are available only to applications that open the line device with an API version of 0x00020000 or greater.

Note The preferred format for specification of the contents of the *callerID* field and the other five similar fields is the TAPI *canonical* number format. For example, a ICLID of "2068828080" received from the switch should be converted to "+1 (206) 8828080" before being placed in the **LINECALLINFO** structure. This standardized format facilitates searching of databases and callback functions implemented in applications.

See Also

LINE_CALLINFO, LINE_MONITORMEDIA, **LINECALLSTATUS**, **LINEDEVCAPS**, **LINEDIALPARAMS**, **lineGenerateDigits**, **lineGetCallInfo**, **lineGetCallStatus**, **lineInitializeEx**, **lineMakeCall**, **lineSecureCall**, **lineSetAppSpecific**, **lineSetCallParams**, **lineSetTerminal**, **lineTranslateAddress**, **LINETRANSLATEOUTPUT**

LINECALLLIST Overview

The **LINECALLLIST** structure describes a list of call handles. A structure of this type is returned by the functions **lineGetNewCalls** and **lineGetConfRelatedCalls**.

```
typedef struct linecalllist_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwCallsNumEntries;
    DWORD  dwCallsSize;
    DWORD  dwCallsOffset;
} LINECALLLIST, FAR *LPLINECALLLIST;
```

Members

dwTotalSize

The total size in bytes 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).

Remarks

No extensions.

See Also

lineGetConfRelatedCalls, **lineGetNewCalls**

LINECALLPARAMS Overview

The **LINECALLPARAMS** structure describes parameters supplied when making calls using **lineMakeCall**. The **LINECALLPARAMS** structure is also used as a parameter in other operations. The comments on the right indicate the default values used when this structure is not provided to **lineMakeCall**.

```
typedef struct linecallparams_tag {  // Defaults:
    DWORD  dwTotalSize;          // ---------

    DWORD  dwBearerMode;             // voice
    DWORD  dwMinRate;                // (3.1kHz)
    DWORD  dwMaxRate;                // (3.1kHz)
    DWORD  dwMediaMode;              // interactiveVoice

    DWORD  dwCallParamFlags;         // 0
    DWORD  dwAddressMode;            // addressID
    DWORD  dwAddressID;              // (any available)

  LINEDIALPARAMS DialParams;        // (0, 0, 0, 0)

    DWORD  dwOrigAddressSize;        // 0
    DWORD  dwOrigAddressOffset;

    DWORD  dwDisplayableAddressSize;  // 0
    DWORD  dwDisplayableAddressOffset;

    DWORD  dwCalledPartySize;        // 0
    DWORD  dwCalledPartyOffset;

    DWORD  dwCommentSize;            // 0
    DWORD  dwCommentOffset;

    DWORD  dwUserUserInfoSize;       // 0
    DWORD  dwUserUserInfoOffset;

    DWORD  dwHighLevelCompSize;      // 0
    DWORD  dwHighLevelCompOffset;

    DWORD  dwLowLevelCompSize;       // 0
    DWORD  dwLowLevelCompOffset;

    DWORD  dwDevSpecificSize;        // 0
    DWORD  dwDevSpecificOffset;

    DWORD  dwPredictiveAutoTransferStates;
    DWORD  dwTargetAddressSize;
    DWORD  dwTargetAddressOffset;
    DWORD  dwSendingFlowspecSize;
    DWORD  dwSendingFlowspecOffset;
    DWORD  dwReceivingFlowspecSize;
    DWORD  dwReceivingFlowspecOffset;
    DWORD  dwDeviceClassSize;
    DWORD  dwDeviceClassOffset;
```

```
    DWORD  dwDeviceConfigSize;
    DWORD  dwDeviceConfigOffset;
    DWORD  dwCallDataSize;
    DWORD  dwCallDataOffset;
    DWORD  dwNoAnswerTimeout;
    DWORD  dwCallingPartyIDSize;
    DWORD  dwCallingPartyIDOffset;
} LINECALLPARAMS, FAR *LPLINECALLPARAMS;
```

Members

dwTotalSize

The total size in bytes allocated to this data structure. This size should be big enough to hold all the fixed and variably sized portions of this data structure.

dwBearerMode

The bearer mode for the call. This field uses the following LINEBEARERMODE_ constants:

LINEBEARERMODE_VOICE

This is a regular 3.1 kHz analog voice grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH

This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation, and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE

The multiuse mode defined by ISDN.

LINEBEARERMODE_DATA

The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA

The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING

This corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a "media stream" by the Telephony API).

LINEBEARERMODE_PASSTHROUGH

When a call is active in LINEBEARERMODE_PASSTHROUGH, the service provider gives direct access to the attached hardware for control by the application. This mode is used primarily by applications desiring temporary direct control over asynchronous modems, accessed via the Win32 comm functions, for the purpose of configuring or using special features not otherwise supported by the service provider.

If **dwBearerMode** is 0, default value is LINEBEARERMODE_VOICE.

dwMinRate

dwMaxRate

The data rate range requested for the call's data stream in bps (bits per second). When making a call, the service provider attempts to provide the highest available rate in the requested range. If a specific data rate is required, both min and max should be set to that value. If an application works best with

one rate but is able to degrade to lower rates, the application should specify these as the max and min rates respectively. If **dwMaxRate** is 0, the default value is as specified by the **dwMaxRate** member of the **LINEDEVCAPS** structure. This is the maximum rate supported by device.

dwMediaMode

The expected media mode of the call. This field uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN

A media stream exists but its mode is not known. This would correspond to a call with an unclassified media type. In typical analog telephony environments, an inbound call's media mode may be unknown until after the call has been answered and the media stream has been filtered to make a determination.

LINEMEDIAMODE_INTERACTIVEVOICE

The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE

The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

A data modem session on the call.

LINEMEDIAMODE_G3FAX

A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_G4FAX

A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_TDD

A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_DIGITALDATA

Digital data is being sent or received over the call.

LINEMEDIAMODE_TELETEX

A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX

A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX

A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED

A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI

An ADSI (Analog Display Services Interface) session on the call.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

If **dwMediaMode** is 0, the default value is LINEMEDIAMODE_INTERACTIVEVOICE.

dwCallParamFlags

These flags specify a collection of Boolean call-setup parameters. This field uses the following LINECALLPARAMFLAGS_ constants:

LINECALLPARAMFLAGS_SECURE

The call should be set up as secure.

LINECALLPARAMFLAGS_IDLE

The call should get an idle call appearance.

LINECALLPARAMFLAGS_BLOCKID

The originator identity should be concealed (block caller ID).

LINECALLPARAMFLAGS_ORIGOFFHOOK

The originator's phone should be automatically taken offhook.

LINECALLPARAMFLAGS_DESTOFFHOOK

The called party's phone should be automatically taken offhook.

dwAddressMode

The mode by which the originating address is specified. The **dwAddressMode** field cannot be LINEADDRESSMODE_ADDRESSID for the function call **lineOpen**. This field uses the following LINEADDRESSMODE_ constants:

LINEADDRESSMODE_ADDRESSID

The address is specified with a small integer in the range 0 to **dwNumAddresses** minus one, where **dwNumAddresses** is the value in the line's **LINEDEVCAPS** structure. The selected address is specified in the **dwAddressID** field.

LINEADDRESSMODE_DIALABLEADDR

The address is specified with its dialable address. The address is contained in the **dwOrigAddressSize dwOrigAddressOffset** variably sized field. If **dwAddressMode** is 0, the default value is LINEADDRESSMODE_ADDRESSID.

dwAddressID

The address ID of the originating address if **dwAddressMode** is set to LINEADDRESSMODE_ADDRESSID.

DialParams

Dial parameters to be used on this call, of type **LINEDIALPARAMS**. When a value of zero is specified for this field, the default value for the field is used as indicated in the **DefaultDialParams** member of the **LINEDEVCAPS** structure. If a non-zero value is specified for a field which is outside the range specified by the corresponding fields in **MinDialParams** and **MaxDialParams** in the **LINEDEVCAPS** structure, the nearest value within the valid range is used instead.

dwOrigAddressSize

dwOrigAddressOffset

The size in bytes of the variably sized field holding the originating address, and the offset in bytes from the beginning of this data structure. The format of this address is dependent on the **dwAddressMode** field.

dwDisplayableAddressSize

dwDisplayableAddressOffset

The displayable string is used for logging purposes. The content of these fields is recorded in the **dwDisplayableAddressOffset** and **dwDisplayableAddressSize** fields of the call's LINECALLINFO message. The **lineTranslateAddress** function returns appropriate information to be placed in this field in the **dwDisplayableAddressSize** and **dwDisplayableAddressOffset** fields of the **LINETRANSLATEOUTPUT** structure.

dwCalledPartySize

dwCalledPartyOffset

The size in bytes of the variably sized field holding called-party information, and the offset in bytes from the beginning of this data structure. This information can be specified by the application that makes the call and is made available in the call's information structure for logging purposes. The format of this field is that of **dwStringFormat**, as specified in **LINEDEVCAPS**.

dwCommentSize

dwCommentOffset

The size in bytes of the variably sized field holding comments about the call, and the offset in bytes from the beginning of this data structure. This information can be specified by the application that makes the call and is made available in the call's information structure for logging purposes. The format of this field is that of **dwStringFormat**, as specified in **LINEDEVCAPS**.

dwUserUserInfoSize

dwUserUserInfoOffset

The size in bytes of the variably sized field holding user-to-user information, and the offset in bytes from the beginning of this data structure. The protocol discriminator field for the user-user information, if required, should appear as the first byte of the data pointed to by **dwUserUserInfoOffset**, and must be accounted for in **dwUserUserInfoSize**.

dwHighLevelCompSize

dwHighLevelCompOffset

The size in bytes of the variably sized field holding high-level compatibility information, and the offset in bytes from the beginning of this data structure

dwLowLevelCompSize

dwLowLevelCompOffset

The size in bytes of the variably sized field holding low-level compatibility information, and the offset in bytes from the beginning of this data structure.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized field holding device-specific information, and the offset in bytes from the beginning of this data structure

dwPredictiveAutoTransferStates

The LINECALLSTATE_ values, entry into which cause the call to be blind-transferred to the specified target address. Set to 0 if automatic transfer is not desired.

dwTargetAddressSize

dwTargetAddressOffset

The size in bytes and offset from the beginning of **LINECALLPARAMS** of a string specifying the

target dialable address (*not* **dwAddressID**); used in the case of certain automatic actions. In the case of predictive dialing, specifies the address to which the call should be automatically transferred. This is essentially the same string that would be passed to **lineBlindTransfer** if automatic transfer were not being used. Set to 0 if automatic transfer is not desired. In the case of a No Hold Conference, specifies the address that should be conferenced to the call. In the case of a One Step Transfer, specifies the address to dial on the consultation call.

dwSendingFlowspecSize

dwSendingFlowspecOffset

The total size in bytes and offset from the beginning of **LINECALLPARAMS** of a WinSock2 **FLOWSPEC** structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in SendingFlowspec.len in a WinSock2 **QOS** structure. Specifies the quality of service desired in the sending direction on the call. The provider-specific portion following the **FLOWSPEC** structure must not contain pointers to other blocks of memory, because TAPI will not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.

dwReceivingFlowspecSize

dwReceivingFlowspecOffset

The total size in bytes and offset from the beginning of **LINECALLPARAMS** of a WinSock2 **FLOWSPEC** structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in ReceivingFlowspec.len in a WinSock2 **QOS** structure. Specifies the quality of service desired in the receiving direction on the call. The provider-specific portion following the **FLOWSPEC** structure must not contain pointers to other blocks of memory, because TAPI will not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.

dwDeviceClassSize

dwDeviceClassOffset

The size in bytes and offset from the beginning of **LINECALLPARAMS** of a NULL-terminated ASCII string (the size includes the null) that indicates the device class of the device whose configuration is specified in *DeviceConfig*. Valid device class strings are the same as those specified for the **lineGetID** function.

dwDeviceConfigSize

dwDeviceConfigOffset

The number of bytes and offset from the beginning of **LINECALLPARAMS** of the opaque configuration data structure pointed to by **dwDevConfigOffset**. This value will have been returned in the **dwStringSize** member in the **VARSTRING** structure returned by **lineGetDevConfig**. If the size is 0, the default device configuration is used. This allows the application to set the device configuration before the call is initiated.

dwCallDataSize

dwCallDataOffset

The size in bytes and offset from the beginning of **LINECALLPARAMS** of the application-settable call data to be initially attached to the call.

dwNoAnswerTimeout

The number of seconds, after the completion of dialing, that the call should be allowed to wait in the PROCEEDING or RINGBACK states, before it is automatically abandoned by the service provider with a LINECALLSTATE_DISCONNECTED and LINEDISCONNECTMODE_NOANSWER. A value of

0 indicates that the application does not desire automatic call abandonment.

dwCallingPartyIDSize

dwCallingPartyIDOffset

The size in bytes and offset from the beginning of **LINECALLPARAMS** of a NULL-terminated ASCII string (the size includes the null) that specifies the identity of the party placing the call. The service provider will, if the content of the ID is acceptable and a path is available, pass the ID along to the called party to indicate the identity of the calling party.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

This structure is used as a parameter to **lineMakeCall** when setting up a call. Its fields allow the application to specify the quality of service requested from the network as well as a variety of ISDN call-setup parameters. If no **LINECALLPARAMS** structure is supplied to **lineMakeCall**, a default POTS voice-grade call is requested with the default values listed above.

Note The fields **DialParams** through **dwDevSpecificOffset** are ignored when an *lpCallParams* parameter is specified with the function **lineOpen**.

The members **dwPredictiveAutoTransferStates** though **dwCallingPartyIDOffset** are available only to applications that open the line device with an API version of 0x00020000 or greater.

See Also

lineBlindTransfer, **LINEDEVCAPS**, **LINEDIALPARAMS**, **lineGetDevConfig**, **lineMakeCall**, **lineOpen**, **lineTranslateAddress**, **LINETRANSLATEOUTPUT**, **VARSTRING**

LINECALLSTATUS Overview

The **LINECALLSTATUS** structure describes the current status of a call. The information in this structure, as returned with **lineGetCallStatus**, depends on the device capabilities of the address, the ownership of the call by the invoking application, and the current state of the call being queried.

```
typedef struct linecallstatus_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwCallState;
    DWORD  dwCallStateMode;
    DWORD  dwCallPrivilege;
    DWORD  dwCallFeatures;

    DWORD  dwDevSpecificSize;
    DWORD  dwDevSpecificOffset;

    DWORD  dwCallFeatures2;
    SYSTEMTIME  tStateEntryTime;
} LINECALLSTATUS, FAR *LPLINECALLSTATUS;
```

Members

dwTotalSize

The total size in bytes 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.

dwCallState

dwCallStateMode

The **dwCallState** field specifies the current call state of the call. The interpretation of the **dwCallStateMode** field is call-state-dependent. It specifies the current mode of the call while in its current state (if that state defines a mode). This field uses the following LINECALLSTATE_ constants:

LINECALLSTATE_IDLE

The call state mode is unused.

LINECALLSTATE_OFFERING

The call state mode is of type LINEOFFERINGMODE_. Values are:

LINEOFFERINGMODE_ACTIVE

Indicates that the call is alerting at the current station (will be accompanied by LINEDEVSTATE_RINGING messages), and if any application is set up to automatically answer, it may do so.

LINEOFFERINGMODE_INACTIVE

Indicates that the call is being offered at more than one station, but the current station is not

alerting (for example, it may be an attendant station where the offering status is advisory, such as blinking a light).

LINECALLSTATE_ACCEPTED

The call state mode is unused.

LINECALLSTATE_DIALTONE

The call state mode, of type LINEDIALTONEMODE_. Values are:

LINEDIALTONEMODE_NORMAL

This is a "normal" dial tone, which typically is a continuous tone.

LINEDIALTONEMODE_SPECIAL

This is a special dial tone indicating a certain condition is currently in effect.

LINEDIALTONEMODE_INTERNAL

This an internal dial tone, as within a PBX.

LINEDIALTONEMODE_EXTERNAL

This is an external (public network) dial tone.

LINEDIALTONEMODE_UNKNOWN

The dial tone mode is not currently known but may become known later.

LINEDIALTONEMODE_UNAVAIL

The dial tone mode is unavailable and will not become known.

LINECALLSTATE_DIALING

Call state mode is unused.

LINECALLSTATE_RINGBACK

Call state mode is unused.

LINECALLSTATE_BUSY

The call state mode is of type LINEBUSYMODE_. Values are:

LINEBUSYMODE_STATION

The busy signal indicates that the called party's station is busy. This is usually signaled with a "normal" busy tone.

LINEBUSYMODE_TRUNK

The busy signal indicates that a trunk or circuit is busy. This is usually signaled with a "long" busy tone.

LINEBUSYMODE_UNKNOWN

The busy signal's specific mode is currently unknown but may become known later.

LINEBUSYMODE_UNAVAIL

The busy signal's specific mode is unavailable and will not become known.

LINECALLSTATE_SPECIALINFO

The call state mode is of type LINESPECIALINFO_. Values are:

LINESPECIALINFO_NOCIRCUIT

This special information tone precedes a no circuit or emergency announcement (trunk blockage category).

LINESPECIALINFO_CUSTIRREG

This special information tone precedes a vacant number, AIS, Centrex number change and nonworking station, access code not dialed or dialed in error, or manual intercept operator message (customer irregularity category).

LINESPECIALINFO_REORDER

This special information tone precedes a reorder announcement (equipment irregularity category).

LINESPECIALINFO_UNKNOWN

Specifics about the special information tone are currently unknown but may become known later.

LINESPECIALINFO_UNAVAIL

Specifics about the special information tone are unavailable and will not become known.

LINECALLSTATE_CONNECTED

Call state mode is of type LINECONNECTEDMODE_. Values are:

LINECONNECTEDMODE_ACTIVE

Indicates that the call is connected at the current station (the current station is a participant in the call).

LINECONNECTEDMODE_INACTIVE

Indicates that the call is active at one or more other stations, but the current station is not a participant in the call.

LINECALLSTATE_PROCEEDING

Call state mode is unused.

LINECALLSTATE_ONHOLD

Call state mode is unused.

LINECALLSTATE_CONFERENCED

Call state mode is unused.

LINECALLSTATE_ONHOLDPENDCONF

Call state mode is unused.

LINECALLSTATE_ONHOLDPENDTRANSF

Call state mode is unused.

LINECALLSTATE_DISCONNECTED

Call state mode is of type LINEDISCONNECTMODE_. Values are:

LINEDISCONNECTMODE_NORMAL

This is a "normal" disconnect request by the remote party. The call was terminated normally.

LINEDISCONNECTMODE_UNKNOWN

The reason for the disconnect request is unknown.

LINEDISCONNECTMODE_REJECT

The remote user has rejected the call.

LINEDISCONNECTMODE_PICKUP

The call was picked up from elsewhere.

LINEDISCONNECTMODE_FORWARDED

The call was forwarded by the switch.

LINEDISCONNECTMODE_BUSY

The remote user's station is busy.

LINEDISCONNECTMODE_NOANSWER

The remote user's station does not answer.

LINEDISCONNECTMODE_NODIALTONE

A dial tone was not detected within a service-provider defined timeout, at a point during dialing when one was expected (such as at a "W" in the dialable string). This can also occur without a service-provider-defined timeout period or without a value specified in the **dwWaitForDialTone** member of the **LINEDIALPARAMS** structure.

LINEDISCONNECTMODE_BADADDRESS

The destination address in invalid.

LINEDISCONNECTMODE_UNREACHABLE

The remote user could not be reached.

LINEDISCONNECTMODE_CONGESTION

The network is congested.

LINEDISCONNECTMODE_INCOMPATIBLE

The remote user's station equipment is incompatible with the type of call requested.

LINEDISCONNECTMODE_UNAVAIL

The reason for the disconnect is unavailable and will not become known later.

LINECALLSTATE_UNKNOWN

Call state mode is unused.

dwCallPrivilege

The application's privilege for this call. This field uses the following LINECALLPRIVILEGE_ constants. Values are:

LINECALLPRIVILEGE_MONITOR

The application has monitor privilege.

LINECALLPRIVILEGE_OWNER

The application has owner privilege.

dwCallFeatures

These flags indicate which Telephony API functions can be invoked on the call, given the availability of the feature in the device capabilities, the current call state, and call ownership of the invoking application. A zero indicates the corresponding feature cannot be invoked by the application on the call in its current state; a one indicates the feature can be invoked. This field uses LINECALLFEATURE_ constants.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized device-specific field, and the offset in bytes from the beginning of this data structure.

dwCallFeatures2

Indicates additional functions can be invoked on the call, given the availability of the feature in the device capabilities, the current call state, and call ownership of the invoking application. An extension of the **dwCallFeatures** field. This field uses LINECALLFEATURE2_ constants.

tStateEntryTime

The Coordinated Universal Time at which the current call state was entered.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

The application is sent a LINE_CALLSTATE message whenever the call state of a call changes. This message only provides the new call state of the call. Additional status about a call is available with **lineGetCallStatus**.

The members **dwCallFeatures2** and **tStateEntryTime** are available only to applications that open the line device with an API version of 0x00020000 or greater.

See Also

LINE_CALLSTATE, **LINEDIALPARAMS**, **lineGetCallStatus**

LINECALLTREATMENTENTRY Overview

```
typedef struct linecalltreatmententry_tag {
    DWORD  dwCallTreatmentID;
    DWORD  dwCallTreatmentNameSize;
    DWORD  dwCallTreatmentNameOffset;
} LINECALLTREATMENTENTRY, FAR *LPLINECALLTREATMENTENTRY;
```

Members

dwCallTreatmentID

One of the LINECALLTREATMENT_ constants (if the treatment is of a predefined type) or a service provider-specific value.

dwCallTreatmentNameSize

dwCallTreatmentNameOffset

Size in bytes (including the terminating null) and offset from the beginning of **LINEADDRESSCAPS** of a null-terminated string identifying the treatment. This would ordinarily describe the content of the music or recorded announcement. If the treatment is of a predefined type, a meaningful name should still be specified, for example, "Silence\0", "Busy Signal\0", "Ringback\0", or "Music\0".

LINECARDENTRY Overview

The **LINECARDENTRY** structure describes a calling card.

```
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

dwPermanentCardID

The permanent ID that identifies the card.

dwCardNameSize

dwCardNameOffset

Contains a NULL-terminated ASCII 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 (it is stored in scrambled form by TAPI). The application can use this 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 defined for calls to numbers in the same area code. The rule is a null-terminated ASCII string.

dwLongDistanceRuleSize

dwLongDistanceRuleOffset

The offset in bytes from the beginning of the **LINETRANSLATECAPS** structure and the total number of bytes in the dialing rule defined for calls to numbers in the other areas in the same country. The rule is a null-terminated ASCII string.

dwInternationalRuleSize

dwInternationalRuleOffset

The offset in bytes from the beginning of the **LINETRANSLATECAPS** structure and the total number of bytes in the dialing rule defined for calls to numbers in other countries. The rule is a null-terminated ASCII string.

dwOptions

Indicates other settings associated with this calling card, using the LINECARDOPTION_ set of constants.

Remarks

Older applications will have been compiled without knowledge of these new fields, and using a SIZEOF LINECARDENTRY smaller than the new size. Because this is an array in the variable portion of a **LINETRANSLATECAPS** structure, it is imperative that older applications receive **LINECARDENTRY** structures in the format they previously expected, or they will not be able to index properly through the array. The application passes in a *dwAPIVersion* parameter with the **lineGetTranslateCaps** function, which can be used for guidance by TAPI in handling this situation. **lineGetTranslateCaps** should use the **LINECARDENTRY** fields and size that match the indicated API version, when building the **LINETRANSLATECAPS** structure to be returned to the application.

No extensions.

See Also

lineGetTranslateCaps, **LINETRANSLATECAPS**

LINECOUNTRYENTRY Overview

The **LINECOUNTRYENTRY** structure provides the information for a single country entry. An array of 1 or more of these structures is returned as part of the **LINECOUNTRYLIST** structure returned by the function **lineGetCountry**.

```
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

dwCountryID

The country ID of the entry. The country ID is an internal identifier which allows multiple entries to exist in the country 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 represented by the entry (that is, the digits that would be dialed in an international call). Only this value should ever be displayed to users (country IDs should never be displayed, as they would be confusing).

dwNextCountryID

The country ID of the next entry in the country list. Because country codes and IDs are not assigned in any regular numeric sequence, the country list is a single linked list, with each entry pointing to the next. The last country in the list has a **dwNextCountryID** value of 0. When the LINECOUNTRYLIST structure is used to obtain the entire list, the entries in the list will be in sequence as linked by their **dwNextCountryID** fields.

dwCountryNameSize

dwCountryNameOffset

The size in bytes and the offset in bytes from the beginning of the **LINECOUNTRYLIST** structure of a null-terminated string giving the name of the country.

dwSameAreaRuleSize

dwSameAreaRuleOffset

The size in bytes and the offset in bytes from the beginning of the **LINECOUNTRYLIST** structure of a null-terminated ASCII string containing the dialing rule for direct-dialed calls to the same area code.

dwLongDistanceRuleSize

dwLongDistanceRuleOffset

The size in bytes and the offset in bytes from the beginning of the LINECOUNTRYLIST structure of a

null-terminated ASCII string containing the dialing rule for direct-dialed calls to other areas in the same country.

dwInternationalRuleSize

dwInternationalRuleOffset

The size in bytes and the offset in bytes from the beginning of the LINECOUNTRYLIST structure of a null-terminated ASCII string containing the dialing rule for direct-dialed calls to other countries.

Remarks

Not extensible.

Because this structure is returned by a new function, backward compatibility is not an issue at this time.

See Also

LINECOUNTRYLIST, **lineGetCountry**

LINECOUNTRYLIST Overview

The **LINECOUNTRYLIST** structure describes a list of countries. A structure of this type is returned by the function **lineGetCountry**.

```
typedef struct linecountrylist_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwNumCountries;
    DWORD  dwCountryListSize;
    DWORD  dwCountryListOffset;
} LINECOUNTRYLIST, FAR *LPLINECOUNTRYLIST;
```

Members

dwTotalSize

The total size in bytes 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 present in the array denominated by **dwCountryListSize** and **dwCountryListOffset**.

dwCountryListSize

dwCountryListOffset

The size in bytes and the offset in bytes from the beginning of this data structure of an array of **LINECOUNTRYENTRY** elements which provide the information on each country.

Remarks

Not extensible.

Because this structure is returned by a new function, backward compatibility is not an issue at this time.

See Also

LINECOUNTRYENTRY, **lineGetCountry**

LINEDEVCAPS Overview

The **LINEDEVCAPS** structure describes the capabilities of a line device.

```
typedef struct linedevcaps_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwProviderInfoSize;
    DWORD  dwProviderInfoOffset;

    DWORD  dwSwitchInfoSize;
    DWORD  dwSwitchInfoOffset;

    DWORD  dwPermanentLineID;
    DWORD  dwLineNameSize;
    DWORD  dwLineNameOffset;
    DWORD  dwStringFormat;
    DWORD  dwAddressModes;
    DWORD  dwNumAddresses;
    DWORD  dwBearerModes;
    DWORD  dwMaxRate;
    DWORD  dwMediaModes;

    DWORD  dwGenerateToneModes;
    DWORD  dwGenerateToneMaxNumFreq;
    DWORD  dwGenerateDigitModes;
    DWORD  dwMonitorToneMaxNumFreq;
    DWORD  dwMonitorToneMaxNumEntries;
    DWORD  dwMonitorDigitModes;
    DWORD  dwGatherDigitsMinTimeout;
    DWORD  dwGatherDigitsMaxTimeout;

    DWORD  dwMedCtlDigitMaxListSize;
    DWORD  dwMedCtlMediaMaxListSize;
    DWORD  dwMedCtlToneMaxListSize;
    DWORD  dwMedCtlCallStateMaxListSize;

    DWORD  dwDevCapFlags;
    DWORD  dwMaxNumActiveCalls;
    DWORD  dwAnswerMode;
    DWORD  dwRingModes;
    DWORD  dwLineStates;

    DWORD  dwUUIAcceptSize;
    DWORD  dwUUIAnswerSize;
    DWORD  dwUUIMakeCallSize;
    DWORD  dwUUIDropSize;
    DWORD  dwUUISendUserUserInfoSize;
    DWORD  dwUUICallInfoSize;

  LINEDIALPARAMS  MinDialParams;
  LINEDIALPARAMS  MaxDialParams;
```

```
  LINEDIALPARAMS  DefaultDialParams;

    DWORD  dwNumTerminals;
    DWORD  dwTerminalCapsSize;
    DWORD  dwTerminalCapsOffset;
    DWORD  dwTerminalTextEntrySize;
    DWORD  dwTerminalTextSize;
    DWORD  dwTerminalTextOffset;

    DWORD  dwDevSpecificSize;
    DWORD  dwDevSpecificOffset;

    DWORD  dwLineFeatures;

    DWORD  dwSettableDevStatus;
    DWORD  dwDeviceClassesSize;
    DWORD  dwDeviceClassesOffset;
} LINEDEVCAPS, FAR *LPLINEDEVCAPS;
```

Members

dwTotalSize

The total size in bytes 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.

dwProviderInfoSize

dwProviderInfoOffset

The size in bytes of the variably sized field containing service provider information, and the offset in bytes from the beginning of this data structure. The **dwProviderInfoSize/Offset** field is intended to provide information about the provider hardware and/or software, such as the vendor name and version numbers of hardware and software. This information can be useful when a user needs to call customer service with problems regarding the provider.

dwSwitchInfoSize

dwSwitchInfoOffset

The size in bytes of the variably sized device field containing switch information, and the offset in bytes from the beginning of this data structure. The **dwSwitchInfoSize/Offset** field is intended to provide information about the switch to which the line device is connected, such as the switch manufacturer, the model name, the software version, and so on. This information can be useful when a user needs to call customer service with problems regarding the switch.

dwPermanentLineID

The permanent DWORD identifier by which the line device is known in the system's configuration. It is a permanent name for the line device. This permanent name (as opposed to *dwDevice ID*) does not change as lines are added or removed from the system. It can therefore be used to link line-specific information in INI files (or other files) in a way that is not affected by adding or removing other lines.

dwLineNameSize

dwLineNameOffset

The size in bytes of the variably sized device field containing a user configurable name for this line device, and the offset in bytes from the beginning of this data structure. This name can be configured by the user when configuring the line device's service provider, and is provided for the user's convenience.

dwStringFormat

The string format used with this line device. This field uses the following STRINGFORMAT_ constants:

STRINGFORMAT_ASCII

The ASCII string format using one byte per character.

STRINGFORMAT_DBCS

The DBCS string format using two bytes per character.

STRINGFORMAT_UNICODE

The Unicode string format using two bytes per character.

dwAddressModes

The mode by which the originating address is specified. This field uses the LINEADDRESSMODE_ constants.

dwNumAddresses

The number of addresses associated with this line device. Individual addresses are referred to by address IDs. Address IDs range from zero to one less than the value indicated by **dwNumAddresses**.

dwBearerModes

This flag array indicates the different bearer modes that the address is able to support. It uses the following LINEBEARERMODE_ constants:

LINEBEARERMODE_VOICE

This is a regular 3.1 kHz analog voice-grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH

This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE

The multiuse mode defined by ISDN.

LINEBEARERMODE_DATA

The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA

The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING

This corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a "media stream" by the Telephony API).

LINEBEARERMODE_PASSTHROUGH

When a call is active in LINEBEARERMODE_PASSTHROUGH, the service provider gives direct access to the attached hardware for control by the application. This mode is used primarily by applications desiring temporary direct control over asynchronous modems, accessed through the Win32 comm functions, for the purpose of configuring or using special features not otherwise supported by the service provider.

dwMaxRate

This field contains the maximum data rate in bits per second for information exchange over the call.

dwMediaModes

This flag array indicates the different media modes the address is able to support. It uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN

A media stream exists but its mode is not known. This corresponds to a call with an unclassified media type. In typical analog telephony environments, an inbound call's media mode may be unknown until after the call has been answered and the media stream filtered to make a determination.

LINEMEDIAMODE_INTERACTIVEVOICE

The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE

The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

A data modem session on the call.

LINEMEDIAMODE_G3FAX

A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_G4FAX

A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_TDD

A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_DIGITALDATA

Digital data being transmitted over the call.

LINEMEDIAMODE_TELETEX

A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX

A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX

A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED

A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI

An ADSI (Analog Display Services Interface) session on the call.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

dwGenerateToneModes

The different kinds of tones that can be generated on this line. This field uses the following LINETONEMODE_ constants:

LINETONEMODE_CUSTOM

The tone is a custom tone defined by the specified frequencies.

LINETONEMODE_RINGBACK

The tone to be generated is a ringback tone.

LINETONEMODE_BUSY

The tone is a standard (station) busy tone.

LINETONEMODE_BEEP

The tone is a beep, as used to announce the beginning of a recording.

LINETONEMODE_BILLING

The tone is billing information tone such as a credit card prompt tone.

dwGenerateToneMaxNumFreq

This field contains the maximum number of frequencies that can be specified in describing a general tone using the **LINEGENERATETONE** data structure when generating a tone using **lineGenerateTone**. A value of zero indicates that tone generation is not available.

dwGenerateDigitModes

This field specifies the digit modes than can be generated on this line. It uses the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE

Generate digits as pulse/rotary pulse sequences.

LINEDIGITMODE_DTMF

Generate digits as DTMF tones.

dwMonitorToneMaxNumFreq

This field contains the maximum number of frequencies that can be specified in describing a general tone using the **LINEMONITORTONE** data structure when monitoring a general tone using **lineMonitorTones**. A value of zero indicates that tone monitor is not available.

dwMonitorToneMaxNumEntries

This field contains the maximum number of entries that can be specified in a tone list to **lineMonitorTones**.

dwMonitorDigitModes

This field specifies the digit modes than can be detected on this line. It uses the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE

Detect digits as audible clicks that are the result of rotary pulse sequences.

LINEDIGITMODE_DTMF

Detect digits as DTMF tones.

LINEDIGITMODE_DTMFEND

Detect the down edges of digits detected as DTMF tones.

dwGatherDigitsMinTimeout

dwGatherDigitsMaxTimeout

These fields contain the minimum and maximum values in milliseconds that can be specified for both the first digit and inter-digit timeout values used by **lineGatherDigits**. If both these field are zero, timeouts are not supported.

dwMedCtlDigitMaxListSize

dwMedCtlMediaMaxListSize

dwMedCtlToneMaxListSize

dwMedCtlCallStateMaxListSize

These fields contain the maximum number of entries that can be specified in the digit list, the media list, the tone list, and the call state list parameters of **lineSetMediaControl** respectively.

dwDevCapFlags

This field specifies various Boolean device capabilities. It uses the following LINEDEVCAPFLAGS_ constants:

LINEDEVCAPFLAGS_CROSSADDRCONF

Specifies whether calls on different addresses on this line can be conferenced.

LINEDEVCAPFLAGS_HIGHLEVCOMP

Specifies whether high-level compatibility information elements are supported on this line.

LINEDEVCAPFLAGS_LOWLEVCOMP

Specifies whether low-level compatibility information elements are supported on this line.

LINEDEVCAPFLAGS_MEDIACONTROL

Specifies whether media-control operations are available for calls at this line.

LINEDEVCAPFLAGS_MULTIPLEADDR

Specifies whether **lineMakeCall** or **lineDial** are able to deal with multiple addresses at once (such as for inverse multiplexing).

LINEDEVCAPFLAGS_CLOSEDROP

Specifies what happens when an open line is closed while the application has calls active on the line. If TRUE, the service provider drops (clears) all active calls on the line when the last application that has opened the line closes it with **lineClose**. If FALSE, the service provider does not drop active calls in such cases. Instead, the calls remain active and under control of external device(s). A service provider typically sets this bit to FALSE if there is some other device that can keep the call alive. For example, if an analog line has the computer and phoneset both connect directly to them in a party-line configuration, the offhook phone will automatically keep the call active even after the computer powers down.

Applications should check this flag to determine whether to warn the user (with an OK/Cancel dialog box) that active calls will be lost.

LINEDEVCAPFLAGS_DIALBILLING

LINEDEVCAPFLAGS_DIALQUIET

LINEDEVCAPFLAGS_DIALDIALTONE

These flags indicate whether the "$", "@", or "W" dialable string modifier is supported for a given line device. It is TRUE if the modifier is supported; otherwise, FALSE. The "?" (prompt user to continue dialing) is never supported by a line device. These flags allow an application to determine "up front" which modifiers would result in the generation of a LINEERR. The application has the choice of pre-scanning dialable strings for unsupported characters, or passing the "raw" string from **lineTranslateAddress** directly to the provider as part of **lineMakeCall** (**lineDial**, and so on) and let the function generate an error to tell it which unsupported modifier occurs first in the string.

dwMaxNumActiveCalls

This field provides the maximum number of (minimum bandwidth) calls that can be active (connected) on the line at any one time. The actual number of active calls may be lower if higher bandwidth calls have been established on the line.

dwAnswerMode

This field specifies the effect on the active call when answering another offering call on a line device. This field uses the following LINEANSWERMODE_ constants:

LINEANSWERMODE_NONE

Answering another call on the same line has no effect on the existing active call(s) on the line.

LINEANSWERMODE_DROP

The currently active call will be automatically dropped.

LINEANSWERMODE_HOLD

The currently active call will automatically be placed on hold.

dwRingModes

This field contains the number of different ring modes that can be reported in the LINE_LINEDEVSTATE message with the *ringing* indication. Different ring modes range from one to **dwRingModes**. Zero indicates no ring.

dwLineStates

This field specifies the different line status components for which the application may be notified in a LINE_LINEDEVSTATE message on this line. It uses the following LINEDEVSTATE_ constants:

LINEDEVSTATE_OTHER

Device-status items other than those listed below have changed. The application should check the current device status to determine which items have 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 containing this constant. For example, in the United States, service providers send a message with this constant every six seconds.

LINEDEVSTATE_CONNECTED

The line was previously disconnected and is now connected to TAPI.

LINEDEVSTATE_DISCONNECTED

This line was previously connected and is now disconnected from TAPI.

LINEDEVSTATE_MSGWAITON

The "message waiting" indicator is turned on.

LINEDEVSTATE_MSGWAITOFF

The "message waiting" indicator is turned off.

LINEDEVSTATE_NUMCOMPLETIONS

The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_INSERVICE

The line is connected to TAPI. This happens when TAPI is first activated or when the line wire is physically plugged in and in service at the switch while TAPI is active.

LINEDEVSTATE_OUTOFSERVICE

The line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE

Maintenance is being performed on the line at the switch. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_OPEN

The line has been opened.

LINEDEVSTATE_CLOSE

The line has been closed.

LINEDEVSTATE_NUMCALLS

The number of calls on the line device has changed.

LINEDEVSTATE_TERMINALS

The terminal settings have changed.

LINEDEVSTATE_ROAMMODE

The roam mode of the line device has changed.

LINEDEVSTATE_BATTERY

The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL

The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC

The line's device-specific information has changed.

LINEDEVSTATE_REINIT

Items have changed in the configuration of line devices. To become aware of these changes (such as for the appearance of new line devices), the application should reinitialize its use of TAPI. The *hDevice* parameter is left NULL for this state change as it applies to any of the lines in the system.

LINEDEVSTATE_LOCK

The locked status of the line device has changed.

LINEDEVSTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINEDEVCAPS** structure for the address have changed. The application should use **lineGetDevCaps** to read the updated structure. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINEDEVSTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI in order to obtain the updated information.

LINEDEVSTATE_CONFIGCHANGE

Indicates that configuration changes have been made to one or more of the media devices associated with the line device. The application, if it desires, may use **lineGetDevConfig** to read the updated information. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

LINEDEVSTATE_TRANSLATECHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINETRANSLATECAPS** structure have changed. The application should use **lineGetTranslateCaps** to read the updated structure. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINEDEVSTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI in order to obtain the updated information.

LINEDEVSTATE_COMPLCANCEL

Indicates that the call completion identified by the completion ID contained in parameter *dwParam2* of the LINE_LINEDEVSTATE message has been externally canceled and is no longer considered valid (if that value were to be passed in a subsequent call to **lineUncompleteCall**, the function would fail with LINEERR_INVALCOMPLETIONID). If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

LINEDEVSTATE_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). A LINE_LINEDEVSTATE message with this value will normally be immediately followed by a LINE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in LINEERR_NODEVICE being returned to the application. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

dwUUIAcceptSize

This field specifies the maximum size of user-to-user information that can be sent during a call accept.

dwUUIAnswerSize

This field specifies the maximum size of user-to-user information that can be sent during a call answer.

dwUUIMakeCallSize

This field specifies the maximum size of user-to-user information that can be sent during a make call.

dwUUIDropSize

This field specifies the maximum size of user-to-user information that can be sent during a call drop.

dwUUISendUserUserInfoSize

This field specifies the maximum size of user-to-user information that can be sent separately any time during a call with **lineSendUserUserInfo**.

dwUUICallInfoSize

This field specifies the maximum size of user-to-user information that can be received in the **LINECALLINFO** structure.

MinDialParams

MaxDialParams

These fields contain the minimum and maximum values for the dial parameters in milliseconds that can be set for calls on this line. Dialing parameters can be set to values in this range. The granularity of the actual settings is service provider-specific.

DefaultDialParams

This field contains the default dial parameters used for calls on this line. These parameter values can be overridden on a per-call basis.

dwNumTerminals

The number of terminals that can be set for this line device, its addresses, or its calls. Individual terminals are referred to by terminal IDs and range from zero to one less than the value indicated by **dwNumTerminals**.

dwTerminalCapsSize

dwTerminalCapsOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing an array with entries of type **LINETERMCAPS**. This array is indexed by terminal IDs, in the range from zero to **dwNumTerminals** minus one. Each entry in the array specifies the terminal device capabilities of the corresponding terminal.

dwTerminalTextEntrySize

The size in bytes of each of the terminal text descriptions pointed at by **dwTerminalTextSize/Offset**.

dwTerminalTextSize

dwTerminalTextOffset

The size in bytes of the variably sized field containing descriptive text about each of the line's available terminals, and the offset in bytes from the beginning of this data structure. Each message is **dwTerminalTextEntrySize** bytes long. The string format of these textual descriptions is indicated by **dwStringFormat** in the line's device capabilities.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized device-specific field, and the offset in bytes from the beginning of this data structure

dwLineFeatures

This field specifies the features available for this line using the LINEFEATURE_ constants. Invoking a supported feature requires the line to be in the proper state and the underlying line device to be opened in a compatible mode. A zero in a bit position indicates that the corresponding feature is never available. A one indicates that the corresponding feature may be available if the line is in the appropriate state for the operation to be meaningful. This field allows an application to discover which line features can be (and which can never be) supported by the device.

dwSettableDevStatus

The LINEDEVSTATUS_ values which can be modified using **lineSetLineDevStatus**.

dwDeviceClassesSize

dwDeviceClassesOffset

Length in bytes and offset from the beginning of **LINEDEVCAPS** of a string consisting of the device class identifiers supported on one or more addresses on this line for use with **lineGetID**, separated by nulls; the last identifier in the list is followed by two nulls.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

Older applications will have been compiled without new fields in the **LINEDEVCAPS** structure, and using a SIZEOF LINEDEVCAPS smaller than the new size. The application passes in a *dwAPIVersion* parameter with the **lineGetDevCaps** function, which can be used for guidance by TAPI in handling this situation. If the application passes in a **dwTotalSize** less than the size of the fixed portion of the structure as defined in the **dwAPIVersion** specified, LINEERR_STRUCTURETOOSMALL will be returned. If sufficient memory has been allocated by the application, before calling **TSPI_lineGetDevCaps**, TAPI will set the **dwNeededSize** and **dwUsedSize** fields to the fixed size of the structure as it existed in the specified API version.

New applications must be cognizant of the API version negotiated, and not examine the contents of fields in the fixed portion beyond the original end of the fixed portion of the structure for the negotiated API version.

If the LINEBEARERMODE_DATA bit is set in **dwBearerModes** member, the **dwMaxRate** member indicates the maximum rate of digital transmission on the bearer channel. The **dwMaxRate** member of the **LINEDEVCAPS** structure can contain valid values even if the **dwBearerModes** member of the **LINEDEVCAPS** structure is *not* set to LINEBEARERMODE_DATA.

If LINEBEARERMODE_DATA is not set in **dwBearerModes**, but the LINEBEARERMODE_VOICE value is set and the LINEMEDIAMODE_DATAMODEM value is set in the **dwMediaModes** member, the **dwMaxRate** member indicates the maximum SYNCHRONOUS (DCE) bit rate on the phone line for the attached modem or functional equivalent. For example, if the modem's fastest modulation speed is V.32bis at 14,400bps, **dwMaxRate** will equal 14400. This is *not* the fastest DTE port rate (which would most likely be 38400, 57600, or 115200), but the fastest bit rate the modem supports on the phone line.

The application must be careful to check to see that LINEBEARERMODE_DATA is *not* set, to avoid misinterpreting the **dwMaxRate** member. It is likely to be 64000 or higher if LINEBEARERMODE_DATA is set.

It should also be noted that if the modem has not been specifically identified (for example, it is a "generic" modem), the figure indicated is a "best guess" based on examination of the modem.

The members **dwSettableDevStatus** through **dwDeviceClassesOffset** are available only to applications that open the line device with an API version of 0x00020000 or greater.

See Also

LINE_LINEDEVSTATE, **LINECALLINFO**, **lineClose**, **lineDial**, **lineGatherDigits**, **LINEGENERATETONE**, **lineGenerateTone**, **lineGetDevCaps**, **lineGetID**, **lineGetTranslateCaps**, **lineSendUserUserInfo**, **lineMakeCall**, **LINEMONITORTONE**, **lineMonitorTones**, **lineSetMediaControl**, **LINETERMCAPS**, **lineTranslateAddress**, **LINETRANSLATECAPS**, **lineUncompleteCall**

LINEDEVSTATUS Overview

The **LINEDEVSTATUS** structure describes the current status of a line device.

```
typedef struct linedevstatus_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwNumOpens;
    DWORD  dwOpenMediaModes;
    DWORD  dwNumActiveCalls;
    DWORD  dwNumOnHoldCalls;
    DWORD  dwNumOnHoldPendCalls;
    DWORD  dwLineFeatures;
    DWORD  dwNumCallCompletions;
    DWORD  dwRingMode;
    DWORD  dwSignalLevel;
    DWORD  dwBatteryLevel;
    DWORD  dwRoamMode;

    DWORD  dwDevStatusFlags;

    DWORD  dwTerminalModesSize;
    DWORD  dwTerminalModesOffset;

    DWORD  dwDevSpecificSize;
    DWORD  dwDevSpecificOffset;

    DWORD dwAvailableMediaModes;
    DWORD dwAppInfoSize;
    DWORD dwAppInfoOffset;
} LINEDEVSTATUS, FAR *LPLINEDEVSTATUS;
```

Members

dwTotalSize

The total size in bytes 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.

dwNumOpens

The number of active opens on the line device.

dwOpenMediaModes

This bit array indicates for which media modes the line device is currently open.

dwNumActiveCalls

The number of calls on the line in call states other than *idle*, *onhold*, *onholdpendingtransfer*, and *onholdpendingconference*.

dwNumOnHoldCalls

The number of calls on the line in the *onhold* state.

dwNumOnHoldPendingCalls

The number of calls on the line in the *onholdpendingtransfer* or *onholdpendingconference* states.

dwLineFeatures

This field specifies the line-related API functions that are currently available on this line. It uses the following LINEFEATURE_ constants:

LINEFEATURE_DEVSPECIFIC

Device-specific operations can be used on the line.

LINEFEATURE_DEVSPECIFICFEAT

Device-specific features can be used on the line.

LINEFEATURE_FORWARD

Forwarding of all addresses can be used on the line.

LINEFEATURE_MAKECALL

An outbound call can be placed on this line using an unspecified address.

LINEFEATURE_SETMEDIACONTROL

Media control can be set on this line.

LINEFEATURE_SETTERMINAL

Terminal modes for this line can be set.

dwNumCallCompletions

The number of outstanding call completion requests on the line.

dwRingMode

The current ring mode on the line device.

dwBatteryLevel

The current battery level of the line device hardware. This is a value in the range 0x00000000 (battery empty) to 0x0000FFFF (battery full).

dwSignalLevel

The current signal level of the connection on the line. This is a value in the range 0x00000000 (weakest signal) to 0x0000FFFF (strongest signal).

dwRoamMode

The current roam mode of the line device. It uses the following LINEROAMMODE_ constants:

LINEROAMMODE_UNKNOWN

The roam mode is currently unknown but may become known later.

LINEROAMMODE_UNAVAIL

The roam mode is unavailable and will not be known.

LINEROAMMODE_HOME

The line is connected to the home network node.

LINEROAMMODE_ROAMA

The line is connected to the Roam-A carrier and calls are charged accordingly.

LINEROAMMODE_ROAMB

The line is connected to the Roam-B carrier and calls are charged accordingly.

dwDevStatusFlags

The size in bytes of this data structure that contains useful information, of type LINEDEVSTATUSFLAGS_.

LINEDEVSTATUSFLAGS_CONNECTED

Specifies whether the line is connected to TAPI. If TRUE, the line is connected and API is able to operate on the line device. If FALSE, the line is disconnected and the application is unable to control the line device using TAPI.

LINEDEVSTATUSFLAGS_LOCKED

This bit is most often used with line devices associated with cellular phones. Many cellular phones have a security mechanism that requires the entry of a password to enable the phone to place calls. This bit may be used to indicate to applications that the phone is locked and cannot place calls until the password is entered on the user interface of the phone, so that the application can present an appropriate alert to the user.

LINEDEVSTATUSFLAGS_MSGWAIT

This field indicates whether the line has a message waiting. If TRUE, a message is waiting; if FALSE, no message is waiting.

LINEDEVSTATUSFLAGS_INSERVICE

This field indicates whether the line is in service. If TRUE, the line is in service; if FALSE, the line is out of service.

dwTerminalModesSize

dwTerminalModesOffset

The size in bytes of the variably sized device field containing an array with DWORD-sized entries, and the offset in bytes from the beginning of this data structure. This array is indexed by terminal IDs, in the range from zero to **dwNumTerminals** minus one. Each entry in the array specifies the current terminal modes for the corresponding terminal set using the **lineSetTerminal** operation for this line. It uses the following LINETERMMODE_ constants:

LINETERMMODE_LAMPS

Lamp events sent from the line to the terminal.

LINETERMMODE_BUTTONS

Button-press events sent from the terminal to the line.

LINETERMMODE_DISPLAY

Display information sent from the line to the terminal.

LINETERMMODE_RINGER

Ringer-control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH

Hookswitch events sent between the terminal and the line.

LINETERMMODE_MEDIATOLINE

The unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMTERM

This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT

This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when the routing of both unidirectional channels of a call's media stream cannot be controlled independently.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized device-specific field, and the offset in bytes from the beginning of this data structure.

dwAvailableMediaModes

Indicates the media modes that can be invoked on new calls created on this line device, when the **dwLineFeatures** field indicates that new calls are possible. If this field is zero, it indicates that the service provider either does not know or cannot indicate which media modes are available, in which case any or all of the media modes indicated in the **dwMediaModes** field in **LINEDEVCAPS** may be available.

dwAppInfoSize

dwAppInfoOffset

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.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

These members **dwAvailableMediaModes** through **dwAppInfoOffset** are available only to applications that open the line device with an API version of 0x00020000 or greater.

See Also

LINEAPPINFO, **LINEDEVCAPS**, **lineSetTerminal**

LINEDIALPARAMS Overview

The **LINEDIALPARAMS** structure specifies a collection of dialing-related fields.

```
typedef struct linedialparams_tag {
    DWORD  dwDialPause;
    DWORD  dwDialSpeed;
    DWORD  dwDigitDuration;
    DWORD  dwWaitForDialtone;
} LINEDIALPARAMS, FAR *LPLINEDIALPARAMS;
```

Members

dwDialPause

The duration in milliseconds of a comma in the dialable address.

dwDialSpeed

The inter-digit time period in milliseconds between successive digits.

dwDigitDuration

The duration in milliseconds of a digit.

dwWaitForDialtone

The maximum amount of time that should be waited for dial tone when a 'W' is used in the dialable address.

Remarks

No extensions.

When a value of zero is specified for a field, the default value for that field is used. If a non-zero value is specified for a field which is outside the range specified by the corresponding fields in **MinDialParams** and **MaxDialParams** in the **LINEDEVCAPS** structure, the nearest value within the valid range is used instead.

The **lineMakeCall** function allows an application to adjust the dialing parameters to be used for the call. The **lineSetCallParams** function can be used to adjust the dialing parameters of an existing call. The **LINECALLINFO** structure lists the call's current dialing parameters.

See Also

LINECALLINFO, LINEDEVCAPS, lineMakeCall, lineSetCallParams

LINEEXTENSIONID Overview

The **LINEEXTENSIONID** structure describes an extension ID. Extension IDs are used to identify service provider-specific extensions for line devices.

```
typedef struct lineextensionid_tag {
    DWORD  dwExtensionID0;
    DWORD  dwExtensionID1;
    DWORD  dwExtensionID2;
    DWORD  dwExtensionID3;
} LINEEXTENSIONID, FAR *LPLINEEXTENSIONID;
```

Members

dwExtensionID0

dwExtensionID1

dwExtensionID2

dwExtensionID3

These four DWORD-sized fields together specify a universally unique extension ID that identifies a line device class extension.

Remarks

No extensions.

Extension IDs are generated using an SDK-provided generation utility.

LINEFORWARD Overview

The **LINEFORWARD** structure describes an entry of the forwarding instructions.

```
typedef struct lineforward_tag {
    DWORD  dwForwardMode;

    DWORD  dwCallerAddressSize;
    DWORD  dwCallerAddressOffset;

    DWORD  dwDestCountryCode;
    DWORD  dwDestAddressSize;
    DWORD  dwDestAddressOffset;
} LINEFORWARD, FAR *LPLINEFORWARD;
```

Members

dwForwardMode

The types of forwarding. The **dwForwardMode** field can have only a single bit set. This field 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.

LINEFORWARDMODE_UNCONDINTERNAL

Forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

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.

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/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/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/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/no answer all calls that originated at a specified address (selective call forwarding).

LINEFORWARDMODE_UNKNOWN

Calls are forwarded, but the conditions under which forwarding will occur are not known at this time. It is possible that the conditions may become known at a future time.

LINEFORWARDMODE_UNAVAIL

Calls are forwarded, but the conditions under which forwarding will occur are not known, and will never be known by the service provider.

dwCallerAddressSize

dwCallerAddressOffset

The size in bytes of the variably sized address field containing the address of a caller to be forwarded, and the offset in bytes from the beginning of the containing data structure. The **dwCallerAddressSize/Offset** field is set to zero if **dwForwardMode** is *not* one of the following: 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 containing the address of the address where calls are to be forwarded, and the offset in bytes from the beginning of the containing data structure.

Remarks

No extensions.

Each entry in the **LINEFORWARD** structure specifies a forwarding request.

LINEFORWARDLIST Overview

The **LINEFORWARDLIST** structure describes a list of forwarding instructions.

```
typedef struct lineforwardlist_tag {
    DWORD  dwTotalSize;

    DWORD  dwNumEntries;
    LINEFORWARD  ForwardList[1];
} LINEFORWARDLIST, FAR *LPLINEFORWARDLIST;
```

Members

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's entries are of type **LINEFORWARD**.

Remarks

No extensions.

The **LINEFORWARDLIST** structure defines the forwarding parameters requested for forwarding calls on an address or on all addresses on a line.

LINEGENERATETONE Overview

The **LINEGENERATETONE** structure contains information about a tone to be generated.

```
typedef struct linegeneratetone_tag {
    DWORD  dwFrequency;
    DWORD  dwCadenceOn;
    DWORD  dwCadenceOff;
    DWORD  dwVolume;
} LINEGENERATETONE, FAR *LPLINEGENERATETONE;
```

Members

dwFrequency

The frequency in Hertz of this tone component. A service provider may adjust (round up or down) the frequency specified by the application to fit its resolution.

dwCadenceOn

The "on" duration in milliseconds of the cadence of the custom tone to be generated. Zero means no tone is 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 is to be generated. A value of 0x0000FFFF represents full volume, and a value of 0x00000000 is silence.

Remarks

No extensions.

This structure is only used for the generation of tones. It is not used for tone monitoring.

LINEINITIALIZEEXPARAMS Overview

```
typedef struct lineinitializeexparams_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;
    DWORD  dwOptions;

  union
  {
    HANDLE  hEvent;
    HANDLE  hCompletionPort;
  } Handles;

    DWORD  dwCompletionKey;

} LINEINITIALIZEEXPARAMS, FAR *LPLINEINITIALIZEEXPARAMS;
```

Members

dwOptions

One of the LINEINITIALIZEEXOPTION_ constants. Specifies the event notification mechanism the applications desires 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 opened using **CreateIoCompletionPort**.

dwCompletionKey

If **dwOptions** specifies LINEINITIALIZEEXOPTION_USECOMPLETIONPORT, the application must specify in this field a value that will be returned through the *lpCompletionKey* parameter of **GetQueuedCompletionStatus** to identify the completion message as a telephony message.

Remarks

See **lineInitializeEx** for further information on these options.

LINELOCATIONENTRY Overview

The **LINELOCATIONENTRY** structure describes a location used to provide an address translation context.

```
typedef struct linelocationentry_tag {
    DWORD  dwPermanentLocationID;
    DWORD  dwLocationNameSize;
    DWORD  dwLocationNameOffset;
    DWORD  dwCountryCode;
    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

dwPermanentLocationID

The permanent ID that identifies the location.

dwLocationNameSize

dwLocationNameOffset

Contains a NULL-terminated ASCII 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 ASCII string specifying the city/area code associated with the location (the size includes the NULL). This information, along with the country code, can be used by applications to "default" entry fields for the user when entering 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 ASCII string containing 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 ASCII string containing 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 ASCII string containing the toll prefix list for the location. The string will contain only prefixes consisting of the digits "0" through "9", separated from each other by a single "," (comma) character.

dwCountryID

The country ID of the country selected for the location. This can be used with the **lineGetCountry** function to obtain additional information about the specific country, such as the country name (the **dwCountryCode** field cannot be used for this purpose because country codes are not unique).

dwOptions

Indicates options in effect for this location, with values taken from the LINELOCATIONOPTION_ set of constants.

dwCancelCallWaitingSize

dwCancelCallWaitingOffset

The size in bytes and the offset in bytes from the beginning of the **LINETRANSLATECAPS** structure of a null-terminated ASCII string containing 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, this may be indicated by **dwCancelCallWaitingSize** being set to 0, or by it being set to 1 and **dwCancelCallWaitingOffset** pointing to an empty string (single null byte).

Remarks

No extensions.

Older applications will have been compiled without knowledge of these new fields, and using a SIZEOF LINELOCATIONENTRY smaller than the new size. Because this is an array in the variable portion of a **LINETRANSLATECAPS** structure, it is imperative that older applications receive **LINELOCATIONENTRY** structures in the format they previously expected, or they will not be able to index through the array properly. The application passes in a *dwAPIVersion* parameter with the **lineGetTranslateCaps** function, which can be used for guidance by TAPI in handling this situation. **lineGetTranslateCaps** should use the **LINELOCATIONENTRY** fields and size that match the indicated API version, when building the **LINETRANSLATECAPS** structure to be returned to the application.

See Also

lineGetCountry, **lineGetTranslateCaps**, **lineTranslateAddress**, **LINETRANSLATECAPS**

LINEMEDIACONTROLCALLSTATE Overview

The **LINEMEDIACONTROLCALLSTATE** structure describes a media action to be executed when detecting transitions into one or more call states.

```
typedef struct linemediacontrolcallstate_tag {
    DWORD  dwCallStates;
    DWORD  dwMediaControl;
} LINEMEDIACONTROLCALLSTATE, FAR *LPLINEMEDIACONTROLCALLSTATE;
```

Members

dwCallStates

Specifies one or more call states. This field uses the following LINECALLSTATE_ constants:

LINECALLSTATE_IDLE

The call is idle−no call exists.

LINECALLSTATE_OFFERING

The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the offering state does not automatically alert the user. Alerting is done by the switch instructing the line to ring, and it does not affect any call states.

LINECALLSTATE_ACCEPTED

The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.

LINECALLSTATE_DIALTONE

The call is receiving 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 being sent to the switch over the call. Note that the operation **lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK

The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY

The call is receiving a busy tone. Busy tone indicates that the call cannot be completed because either a circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO

Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED

The call has been established, and the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING

Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD

The call is on hold by the switch.

LINECALLSTATE_CONFERENCED

The call is currently a member of a multiparty conference call.

LINECALLSTATE_ONHOLDPENDCONF

The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENDTRANSF

The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED

The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN

The state of the call is not known. This may be due to limitations of the call-progress detection implementation.

dwMediaControl

The media-control action. This field uses the following LINEMEDIACONTROL_ constants:

LINEMEDIACONTROL_NONE

No change is to be made to the media stream.

LINEMEDIACONTROL_RESET

Reset the media stream. Equivalent to an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE

Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME

Start or resume a paused media stream.

LINEMEDIACONTROL_RATEUP

The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN

The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL

The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP

The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN

The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL

The amplitude of the media stream is returned to normal.

Remarks

No extensions.

The **LINEMEDIACONTROLCALLSTATE** structure defines a triple <call state(s), media-control action>. An array of these triples is passed to the **lineSetMediaControl** function to set the media-control actions triggered by the transition to the call state of the given call. When a transition to a listed call state is detected, the corresponding action on the media stream is invoked.

See Also

lineGenerateDigits, **lineSetMediaControl**

LINEMEDIACONTROLDIGIT Overview

The **LINEMEDIACONTROLDIGIT** structure describes a media action to be executed when detecting a digit. It is used as an entry in an array.

```
typedef struct linemediacontroldigit_tag {
    DWORD  dwDigit;
    DWORD  dwDigitModes;
    DWORD  dwMediaControl;
} LINEMEDIACONTROLDIGIT, FAR *LPLINEMEDIACONTROLDIGIT;
```

Members

dwDigit

The low-order byte of this DWORD specifies the digit in ASCII whose detection is to trigger a media action. Valid digits depend on the media mode.

dwDigitModes

The digit mode(s) that are to be monitored. This field uses the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE

Detect digits as audible clicks that are the result of rotary pulse sequences. Valid digits for pulse are '0' through '9'.

LINEDIGITMODE_DTMF

Detect digits as DTMF tones. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

LINEDIGITMODE_DTMFEND

Detect and provide application notification of DTMF down edges. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

dwMediaControl

The media-control action. This field uses the following LINEMEDIACONTROL_ constants:

LINEMEDIACONTROL_NONE

No change is to be made to the media stream.

LINEMEDIACONTROL_RESET

Reset the media stream. Equivalent to an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE

Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME

Start or resume a paused media stream.

LINEMEDIACONTROL_RATEUP

The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN

The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL

The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP

The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN

The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL

The amplitude of the media stream is returned to normal.

Remarks

No extensions.

The **LINEMEDIACONTROLMEDIA** structure defines a triple <digit, digit mode(s), media-control action>. An array of these triples is passed to the **lineSetMediaControl** function to set the media-control actions triggered by digits detected on a given call. When a listed digit is detected, then the corresponding action on the media stream is invoked.

See Also

LINEMEDIACONTROLMEDIA, **lineSetMediaControl**

LINEMEDIACONTROLMEDIA Overview

The **LINEMEDIACONTROLMEDIA** structure describes a media action to be executed when detecting a media-mode change. It is used as an entry in an array.

```
typedef struct linemediacontrolmedia_tag {
    DWORD  dwMediaModes;
    DWORD  dwDuration;
    DWORD  dwMediaControl;
} LINEMEDIACONTROLMEDIA, FAR *LPLINEMEDIACONTROLMEDIA;
```

Members

dwMediaModes

One or more media modes. This field uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN

A media stream exists but its mode is not known. This would correspond to a call with an unclassified media type. In typical analog telephony environments, an inbound call's media mode may be unknown until after the call has been answered and the media stream filtered to make a determination.

LINEMEDIAMODE_INTERACTIVEVOICE

The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE

The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM

A data modem session on the call.

LINEMEDIAMODE_G3FAX

A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_G4FAX

A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_TDD

A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_DIGITALDATA

Digital data is being sent or received over the call.

LINEMEDIAMODE_TELETEX

A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX

A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX

A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED

A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI

An ADSI (Analog Display Services Interface) session on the call.

LINEMEDIAMODE_VOICEVIEW

The media mode of the call is VoiceView.

dwDuration

The duration in milliseconds during which the media mode should be present before the application should be notified or media-control action should be taken.

dwMediaControl

The media-control action. This field uses the following LINEMEDIACONTROL_ constants:

LINEMEDIACONTROL_NONE

No change is to be made to the media stream.

LINEMEDIACONTROL_RESET

Reset the media stream. Equivalent to an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE

Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME

Start or resume a paused media stream.

LINEMEDIACONTROL_RATEUP

The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN

The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL

The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP

The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN

The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL

The amplitude of the media stream is returned to normal.

Remarks

No extensions.

The **LINEMEDIACONTROLMEDIA** structure defines a triple <media mode(s), duration, media-control action>. An array of these triples is passed to the **lineSetMediaControl** function to set the media-control actions triggered by media-mode changes for a given call. When a change to a listed media mode is detected, then the corresponding action on the media stream is invoked.

See Also

lineSetMediaControl

LINEMEDIACONTROLTONE Overview

The **LINEMEDIACONTROLTONE** structure describes a media action to be executed when a tone has been detected. It is used as an entry in an array.

```
typedef struct linemediacontroltone_tag {
    DWORD  dwAppSpecific;
    DWORD  dwDuration;
    DWORD  dwFrequency1;
    DWORD  dwFrequency2;
    DWORD  dwFrequency3;
    DWORD  dwMediaControl;
} LINEMEDIACONTROLTONE, FAR *LPLINEMEDIACONTROLTONE;
```

Members

dwAppSpecific

This field is used by the application for tagging the tone. When this tone is detected, the value of the **dwAppSpecific** field is 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 zero should be used for the unused frequencies. A tone with all three frequencies set to zero is interpreted as silence and can be use for silence detection.

dwMediaControl

The media-control action. This field uses the following LINEMEDIACONTROL_ constants:

LINEMEDIACONTROL_NONE

No change is to be made to the media stream.

LINEMEDIACONTROL_RESET

Reset the media stream. Equivalent to an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE

Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME

Start or resume a paused media stream.

LINEMEDIACONTROL_RATEUP

The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN

The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL

The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP

The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN

The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL

The amplitude of the media stream is returned to normal.

Remarks

No extensions.

The **LINEMEDIACONTROLTONE** structure defines a tuple <tone, media-control action>. An array of these tuples is passed to the **lineSetMediaControl** function to set media-control actions triggered by media-mode changes for a given call. When a change to a listed media mode is detected, the corresponding action on the media stream is invoked.

A tone with all frequencies set to zero corresponds to silence. An application can thus monitor the call's information stream for silence.

See Also

lineSetMediaControl

LINEMESSAGE Overview

```
typedef struct linemessage_tag {
    DWORD  hDevice;
    DWORD  dwMessageID;
    DWORD  dwCallbackInstance;
    DWORD  dwParam1;
   DWORD  dwParam2;
    DWORD  dwParam3;
} LINEMESSAGE, FAR *LPLINEMESSAGE;
```

Members

hDevice

A handle to either a line device or a call. The nature of this handle (line handle or call handle) can be determined by the context provided by **dwMessageID**.

dwMessageID

A line or call device message.

dwCallbackInstance

Instance data passed back to the application, which was specified by the application in **lineInitializeEx**. This DWORD is not interpreted by TAPI.

dwParam1

A parameter for the message.

dwParam2

A parameter for the message.

dwParam3

A parameter for the message.

Remarks

For information about parameter values passed in this structure, see Line Device Messages.

LINEMONITORTONE Overview

The **LINEMONITORTONE** structure describes a tone to be monitored. This is used as an entry in an array.

```
typedef struct linemonitortone_tag {
    DWORD  dwAppSpecific;
    DWORD  dwDuration;
    DWORD  dwFrequency1;
    DWORD  dwFrequency2;
    DWORD  dwFrequency3;
} LINEMONITORTONE, FAR *LPLINEMONITORTONE;
```

Members

dwAppSpecific

This field is used by the application for tagging the tone. When this tone is detected, the value of the **dwAppSpecific** field is 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 zero should be used for the unused frequencies. A tone with all three frequencies set to zero is interpreted as silence and can be used for silence detection.

Remarks

No extensions.

The **LINEMONITORTONE** structure defines a tone for the purpose of detection. An array of tones is passed to the **lineMonitorTones** function which 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's information stream for silence.

See Also

LINE_MONITORTONE, **lineMonitorTones**

LINEPROVIDERENTRY Overview

The **LINEPROVIDERENTRY** structure provides the information for a single service provider entry. An array of these structures is returned as part of the **LINEPROVIDERLIST** structure returned by the function **lineGetProviderList**.

```
typedef struct lineproviderentry_tag {
    DWORD  dwPermanentProviderID;
    DWORD  dwProviderFilenameSize;
    DWORD  dwProviderFilenameOffset;
} LINEPROVIDERENTRY, FAR *LPLINEPROVIDERENTRY;
```

Members

dwPermanentProviderID

The permanent provider ID 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 ASCII string containing the filename (path) of the service provider DLL (.TSP) file.

Remarks

Not extensible.

See Also

lineGetProviderList, **LINEPROVIDERLIST**

LINEPROVIDERLIST Overview

The **LINEPROVIDERLIST** structure describes a list of service providers. A structure of this type is returned by the function **lineGetProviderList**.

```
typedef struct lineproviderlist_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;

    DWORD  dwNumProviders;
    DWORD  dwProviderListSize;
    DWORD  dwProviderListOffset;
} LINEPROVIDERLIST, FAR *LPLINEPROVIDERLIST;
```

Members

dwTotalSize

The total size in bytes 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 present in the array 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.

Remarks

Not extensible.

LINEPROXYREQUEST Overview

```
typedef struct lineproxyrequest_tag {
    DWORD dwSize;
    DWORD dwClientMachineNameSize;
    DWORD dwClientMachineNameOffset;
    DWORD dwClientUserNameSize;
    DWORD dwClientUserNameOffset;
    DWORD dwClientAppAPIVersion;
    DWORD dwRequestType;
  union {
   struct {
    DWORD dwAddressID;
      LINEAGENTGROUPLIST GroupList;
      } SetAgentGroup;
   struct {
    DWORD dwAddressID;
    DWORD dwAgentState;
    DWORD dwNextAgentState;
      } SetAgentState;
   struct {
    DWORD dwAddressID;
    DWORD dwActivityID;
      } SetAgentActivity;
   struct {
    DWORD dwAddressID;
      LINEAGENTCAPS AgentCaps;
      } GetAgentCaps;
   struct {
    DWORD dwAddressID;
      LINEAGENTSTATUS AgentStatus;
      } GetAgentStatus;
   struct {
    DWORD dwAddressID;
    DWORD dwAgentExtensionIDIndex;
    DWORD dwSize;
      BYTE Params[1];
      } AgentSpecific;
   struct {
    DWORD dwAddressID;
      LINEAGENTACTIVITYLIST ActivityList;
      } GetAgentActivityList;
   struct {
    DWORD dwAddressID;
      LINEAGENTGROUPLIST GroupList;
      } GetAgentGroupList;
  };
} LINEPROXYREQUEST, FAR *LPLINEPROXYREQUEST;
```

Members

dwSize

The total number of bytes allocated by TAPI to contain the **LINEPROXYREQUEST** structure. Note that the **dwTotalSize** field of any structure contained within **LINEPROXYREQUEST** (for example,

LINEAGENTCAPS) reflects only the number of bytes allocated for that specific structure.

dwClientMachineNameSize

dwClientMachineNameOffset

Size in bytes (including the terminating null) and offset from the beginning of LINEPROXYREQUEST of a null-terminated string identifying the client machine that made this request.

dwClientUserNameSize

dwClientUserNameOffset

Size in bytes (including the terminating null) and offset from the beginning of **LINEPROXYREQUEST** of a null-terminated string identifying the user under whose account the application is running on the client machine.

dwClientAppAPIVersion

The global (highest) API version supported by the application that made the request. The proxy handler should restrict the contents of any data returned to the application to those fields and values that were defined in this, or earlier, versions of TAPI.

dwRequestType

One of the LINEPROXYREQUEST_ constants. Identifies the type of function and the union component that defines the remaining data in the structure.

SetAgentGroup

The union component used when **dwRequestType** is LINEPROXYREQUEST_SETAGENT.

dwAddressID

The identifier of the address for which the agent is to be set.

GroupList

A structure of type **LINEAGENTGROUPLIST**. Offsets within this structure are relative to the beginning of **SetAgentGroup.GroupList** rather than the beginning of the **LINEPROXYREQUEST** structure.

SetAgentState

The union component used when **dwRequestType** is LINEPROXYREQUEST_SETAGENTSTATE.

dwAddressID

The identifier of the address for which the agent state is to be set.

dwAgentState

The new agent state, or 0 to leave the agent state unchanged.

dwNextAgentState

The new next agent state, or 0 to use the default next state associated with the specified agent state.

SetAgentActivity

The union component used when **dwRequestType** is LINEPROXYREQUEST_SETAGENTACTIVITY.

dwAddressID

The identifier of the address for which the agent activity is to be set.

dwActivityID

The identifier for the activity being selected.

GetAgentCaps

The union component used when **dwRequestType** is LINEPROXYREQUEST_GETAGENTCAPS.

dwAddressID

The identifier of the address for which the agent capabilities are to be retrieved.

AgentCaps

A structure of type **LINEAGENTCAPS**. Offsets within this structure are relative to the beginning of **GetAgentCaps.AgentCaps** rather than the beginning of the **LINEPROXYREQUEST** structure. The **dwTotalSize** field is set by TAPI and the remaining bytes set to 0. The proxy handler must fill in **dwNeededSize**, **dwUsedSize**, and the remaining fields as appropriate, before calling **lineProxyResponse**.

GetAgentStatus

The union component used when **dwRequestType** is LINEPROXYREQUEST_SETAGENTGROUP.

dwAddressID

The identifier of the address for which the agent status is to be retrieved.

AgentStatus

A structure of type **LINEAGENTSTATUS**. Offsets within this structure are relative to the beginning of **GetAgentStatus.AgentStatus** rather than the beginning of the **LINEPROXYREQUEST** structure. The **dwTotalSize** field is set by TAPI and the remaining bytes set to 0. The proxy handler must fill in **dwNeededSize**, **dwUsedSize**, and the remaining fields as appropriate, before calling **lineProxyResponse**.

AgentSpecific

The union component used when **dwRequestType** is LINEPROXYREQUEST_AGENTSPECIFIC.

dwAddressID

The identifier of the address for which the agent status is to be retrieved.

dwAgentExtensionIDIndex

The index of the handler extension being invoked; the ID's position within the array of extension IDs returned in **LINEAGENTCAPS**.

dwSize

The total size in bytes of the **Params** parameter block.

Params

A block of memory which includes the contents passed to the handler from the application. If the handler is to return data to the application, it must be written into this parameter block before calling **lineProxyResponse**.

GetAgentActivityList

The union component used when **dwRequestType** is LINEPROXYREQUEST_GETAGENTACTIVITYLIST.

dwAddressID

The identifier of the address for which the agent activity list is to be retrieved.

ActivityList

A structure of type LINEAGENTACTIVITYLIST. Offsets within this structure are relative to the beginning of **GetAgentActivityList.ActivityList** rather than the beginning of the **LINEPROXYREQUEST** structure. The **dwTotalSize** field is set by TAPI and the remaining bytes set to 0. The proxy handler must fill in **dwNeededSize**, **dwUsedSize**, and the remaining fields as appropriate, before calling **lineProxyResponse**.

GetAgentGroupList

The union component used when **dwRequestType** is LINEPROXYREQUEST_GETAGENTGROUPLIST.

dwAddressID

The identifier of the address for which the agent group list is to be retrieved.

GroupList

A structure of type **LINEAGENTGROUPLIST**. Offsets within this structure are relative to the beginning of **GetAgentGroupList.GroupList** rather than the beginning of the **LINEPROXYREQUEST** structure. The **dwTotalSize** field is set by TAPI and the remaining bytes set to 0. The proxy handler must fill in **dwNeededSize**, **dwUsedSize**, and the remaining fields as appropriate, before calling **lineProxyResponse**.

See Also

lineProxyResponse

LINEREQMAKECALL Overview

The **LINEREQMAKECALL** structure describes a **tapiRequestMakeCall** request.

```
typedef struct linereqmakecall_tag {
    char  szDestAddress[TAPIMAXDESTADDRESSSIZE];
    char  szAppName[TAPIMAXAPPNAMESIZE];
    char  szCalledParty[TAPIMAXCALLEDPARTYSIZE];
    char  szComment[TAPIMAXCOMMENTSIZE];
} LINEREQMAKECALL, FAR *LPLINEREQMAKECALL;
```

Members

szDestAddress[TAPIMAXADDRESSSIZE**]**

The NULL-terminated destination address of the make-call request. The address can use the canonical address format or the dialable address format. The maximum length of the address is TAPIMAXDESTADDRESSSIZE characters, which includes the NULL terminator. Longer strings are truncated.

szAppName[TAPIMAXAPPNAMESIZE**]**

The ASCII NULL-terminated user-friendly application name or filename of the application that originated the request. The maximum length of the address is TAPIMAXAPPNAMESIZE characters, which includes the NULL terminator.

szCalledParty[TAPIMAXCALLEDPARTYSIZE**]**

The ASCII NULL-terminated user-friendly called-party name. The maximum length of the called-party information is TAPIMAXCALLEDPARTYSIZE characters, which includes the NULL terminator.

szComment[TAPIMAXCOMMENTSIZE**]**

The ASCII NULL-terminated comment about the call request. The maximum length of the comment string is TAPIMAXCOMMENTSIZE characters, which includes the NULL terminator.

Remarks

No extensions.

The **szDestAddress** field contains the address of the remote party; the other fields are useful for logging purposes. An application must use this structure to interpret the request buffer it received from **lineGetRequest** with the LINEREQUESTMODE_MAKECALL request mode.

See Also

lineGetRequest, **tapiRequestMakeCall**

LINETERMCAPS Overview

The **LINETERMCAPS** structure describes the capabilities of a line's terminal device.

```
typedef struct linetermcaps_tag {
    DWORD  dwTermDev;
    DWORD  dwTermModes;
    DWORD  dwTermSharing;
} LINETERMCAPS, FAR *LPLINETERMCAPS;
```

Members

dwTermDev

The device type of the terminal. This field uses the following LINETERMDEV_ constants:

LINETERMDEV_PHONE

The terminal is a phone set.

LINETERMDEV_HEADSET

The terminal is a headset

LINETERMDEV_SPEAKER

The terminal is an external speaker and microphone.

dwTermModes

The terminal mode(s) the terminal device is able to deal with. This field uses the following LINETERMMODE_ constants:

LINETERMMODE_BUTTONS

Button-press events sent from the terminal to the line.

LINETERMMODE_LAMPS

Lamp events sent from the line to the terminal.

LINETERMMODE_DISPLAY

Display information sent from the line to the terminal.

LINETERMMODE_RINGER

Ringer-control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH

Hookswitch events sent from the terminal to the line.

LINETERMMODE_MEDIATOLINE

The unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE

The unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT

This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when the routing of both unidirectional channels of a call's media stream cannot be controlled independently.

dwTermSharing

Specifies how the terminal device is shared between line devices. This field uses the following LINETERMSHARING_ constants:

LINETERMSHARING_PRIVATE

The terminal device is private to a single line device.

LINETERMSHARING_SHAREDEXCL

The terminal device can be used by multiple lines. The last line device to do a **lineSetTerminal** to the terminal for a given terminal mode will have exclusive connection to the terminal for that mode.

LINETERMSHARING_SHAREDCONF

The terminal device can be used by multiple lines. The **lineSetTerminal** requests of the various terminals end up being "merged" at the terminal.

Remarks

No extensions.

See Also

lineSetTerminal

LINETRANSLATECAPS Overview

The **LINETRANSLATECAPS** structure describes the address translation capabilities.

```
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

dwTotalSize

The total size in bytes 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 **LocationList**. It includes all locations defined, including 0 (default).

dwLocationListSize

dwLocationListOffset

List of locations known to address translation. The list consists of a sequence of **LINELOCATIONENTRY** structures. The **dwLocationListOffset** field points to the first byte of the first **LINELOCATIONENTRY** structure, and the **dwLocationListSize** field indicates the total number of bytes in the entire list.

dwCurrentLocationID

This is the **dwPermanentLocationID** from the **LINELOCATIONENTRY** for the Current Location.

dwNumCards

The number of entries in the **CardList**.

dwCardListSize

dwCardListOffset

List of calling cards known to address translation. It includes only non-hidden card entries and always includes card 0 (direct dial). The list consists of a sequence of **LINECARDENTRY** structures. The

dwCardListOffset field points to the first byte of the first **LINECARDENTRY** structure, and the **dwCardListSize** field indicates the total number of bytes in the entire list.

dwCurrentPreferredCardID

This is the **dwPreferredCardID** from the LINELOCATIONENTRY for the Current Location.

Remarks

No extensions.

See Also

LINECARDENTRY, **LINELOCATIONENTRY**

LINETRANSLATEOUTPUT Overview

The **LINETRANSLATEOUTPUT** structure describes the result of an address translation.

```
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

dwTotalSize

The total size in bytes 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 which can be passed to the **lineMakeCall**, **lineDial**, or other function requiring a dialable string. The output is always a null-terminated ASCII string (null is accounted for in Size). Ancillary fields such as name and subaddress are included in this output string if they were in the input string. This string may contain private information such as calling card numbers. It should not be displayed to the user, to prevent inadvertent visibility to unauthorized persons.

dwDisplayableStringSize

dwDisplayableStringOffset

Contains the translated output which can be displayed to the user for confirmation. It will be identical to **DialableString**, except that calling card digits will be replaced with the "friendly name" of the card enclosed within bracket characters (for example, "[AT&T Card]"), and ancillary fields such as name and subaddress will be removed. It should normally be safe to display this string in call-status dialog boxes without exposing private information to unauthorized persons. This information is also appropriate to include in call logs.

dwCurrentCountry

Contains the country code configured in **CurrentLocation**. This value may be used to control the display by the application of certain user interface elements, local call progress tone detection, and for other purposes.

dwDestCountry

Contains the destination country code of the translated address. This value may be passed to the *dwCountryCode* parameter of **lineMakeCall** and other dialing functions (so that the call progress tones of the destination country such as a busy signal will be properly detected). This field is set to 0 if the destination address passed to **lineTranslateAddress** is not in canonical format.

dwTranslateResults

Indicates the information derived from the translation process, which may assist the application in presenting user-interface elements. This field uses the following LINETRANSLATERESULT_ constants:

LINETRANSLATERESULT_CANONICAL

Indicates that the input string was in valid canonical format.

LINETRANSLATERESULT_INTERNATIONAL

If this bit is on, the call is being treated as an international call (country code specified in the destination address is the different from the country code specified for the **CurrentLocation**).

LINETRANSLATERESULT_LONGDISTANCE

If this bit is on, the call is being treated as a long distance call (country code specified in the destination address is the same but area code is different from those specified for the **CurrentLocation**).

LINETRANSLATERESULT_LOCAL

If this bit is on, the call is being treated as a local call (country code and area code specified in the destination address are the same as those specified for the **CurrentLocation**).

LINETRANSLATERESULT_INTOLLLIST

If this bit is on, the local call is being dialed as long distance because the country has toll calling and the prefix appears in the **TollPrefixList** of the **CurrentLocation**.

LINETRANSLATERESULT_NOTINTOLLLIST

If this bit is on, the country supports toll calling but the prefix does not appear in the **TollPrefixList**, so the call is dialed as a local call. Note that if both INTOLLIST and NOTINTOLLIST are off, the current country does *not* support toll prefixes, and user-interface elements related to toll prefixes should not be presented to the user; if either such bit is on, the country *does* support toll lists, and the related user-interface elements should be enabled.

LINETRANSLATERESULT_DIALBILLING

Indicates that the returned address contains a "$".

LINETRANSLATERESULT_DIALQUIET

Indicates that the returned address contains a "@".

LINETRANSLATERESULT_DIALDIALTONE

Indicates that the returned address contains a "W".

LINETRANSLATERESULT_DIALPROMPT

Indicates that the returned address contains a "?".

Remarks

No extensions.

See Also

lineDial, **lineMakeCall**, **lineTranslateAddress**

Phone Device Structures

The following is the reference for phone device structures.

PHONEBUTTONINFO Overview

The **PHONEBUTTONINFO** structure contains information about a button on a phone device.

```
typedef struct phonebuttoninfo_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;
    DWORD  dwButtonMode;
    DWORD  dwButtonFunction;
    DWORD  dwButtonTextSize;
    DWORD  dwButtonTextOffset;
    DWORD  dwDevSpecificSize;
    DWORD  dwDevSpecificOffset;
    DWORD  dwButtonState;
} PHONEBUTTONINFO, FAR *LPPHONEBUTTONINFO;
```

Members

dwTotalSize

The total size in bytes 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.

dwButtonMode

The mode or general usage class of the button. This parameter uses the following PHONEBUTTONMODE_ constants:

PHONEBUTTONMODE_DUMMY

This value is used to describe a button/lamp position that has no corresponding button but has only a lamp.

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 twelve keypad buttons, '0' through '9', '*', and '#'.

PHONEBUTTONMODE_LOCAL

The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY

The button is a "soft" button associated with the phone's display. A phone set can have zero or more display buttons.

dwButtonFunction

The function assigned to the button. This field uses the PHONEBUTTONFUNCTION_ constants.

dwButtonTextSize

dwButtonTextOffset

The size in bytes of the variably sized field containing descriptive text for this button., and the offset in bytes from the beginning of this data structure. The format of this information is as specified in the **dwStringFormat** field of the phone's device capabilities.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized device-specific field, and the offset in bytes from the beginning of this data structure.

dwButtonState

For the **phoneGetButtonInfo** function, this field indicates the current state of the button, using the **PHONEBUTTONSTATE_** constants. This field is ignored by the **phoneSetButtonInfo** function.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

Older applications will have been compiled without this field in the **PHONEBUTTONINFO** structure, and using a **SIZEOF PHONEBUTTONINFO** smaller than the new size. The application passes in a *dwAPIVersion* parameter with the **phoneOpen** function, which can be used for guidance by TAPI in handling this situation. If the application passes in a **dwTotalSize** less than the size of the fixed portion of the structure as defined in the **dwAPIVersion** specified, PHONEERR_STRUCTURETOOSMALL will be returned. If sufficient memory has been allocated by the application, before calling **TSPI_phoneGetButtonInfo**, TAPI will set the **dwNeededSize** and **dwUsedSize** fields to the fixed size of the structure as it existed in the specified API version.

New service providers (which support the new API version) must examine the API version passed in. If the API version is less than the highest version supported by the provider, the service provider must not fill in fields not supported in older API versions, as these would fall in the variable portion of the older structure.

New applications must be cognizant of the API version negotiated, and not examine the contents of fields in the fixed portion beyond the original end of the fixed portion of the structure for the negotiated API version.

See Also

phoneGetButtonInfo, **phoneOpen**

PHONECAPS Overview

The **PHONECAPS** structure describes the capabilities of a phone device.

```
typedef struct phonecaps_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;
    DWORD  dwProviderInfoSize;
    DWORD  dwProviderInfoOffset;
    DWORD  dwPhoneInfoSize;
    DWORD  dwPhoneInfoOffset;
    DWORD  dwPermanentPhoneID;
    DWORD  dwPhoneNameSize;
    DWORD  dwPhoneNameOffset;
    DWORD  dwStringFormat;
    DWORD  dwPhoneStates;
    DWORD  dwHookSwitchDevs;
    DWORD  dwHandsetHookSwitchModes;
    DWORD  dwSpeakerHookSwitchModes;
    DWORD  dwHeadsetHookSwitchModes;
    DWORD  dwVolumeFlags;
    DWORD  dwGainFlags;
    DWORD  dwDisplayNumRows;
    DWORD  dwDisplayNumColumns;
    DWORD  dwNumRingModes;
    DWORD  dwNumButtonLamps;
    DWORD  dwButtonModesSize;
    DWORD  dwButtonModesOffset;
    DWORD  dwButtonFunctionsSize;
    DWORD  dwButtonFunctionsOffset;
    DWORD  dwLampModesSize;
    DWORD  dwLampModesOffset;
    DWORD  dwNumSetData;
    DWORD  dwSetDataSize;
    DWORD  dwSetDataOffset;
    DWORD  dwNumGetData
    DWORD  dwGetDataSize;
    DWORD  dwGetDataOffset;
    DWORD  dwDevSpecificSize;
    DWORD  dwDevSpecificOffset;

    DWORD dwDeviceClassesSize;
    DWORD dwDeviceClassesOffset;
    DWORD dwPhoneFeatures;
    DWORD dwSettableHandsetHookSwitchModes;
    DWORD dwSettableSpeakerHookSwitchModes;
    DWORD dwSettableHeadsetHookSwitchModes;
    DWORD dwMonitoredHandsetHookSwitchModes;
    DWORD dwMonitoredSpeakerHookSwitchModes;
    DWORD dwMonitoredHeadsetHookSwitchModes;
} PHONECAPS, FAR *LPPHONECAPS;
```

Members

dwTotalSize

The total size in bytes 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.

dwProviderInfoSize

dwProviderInfoOffset

The size in bytes of the variably sized field containing service provider-specific information, and the offset in bytes from the beginning of this data structure.

The **dwProviderInfoSize/Offset** field is intended to provide information about the provider hardware and/or software, such as the vendor name and version numbers of hardware and software. This information can be useful when a user needs to call customer service with problems regarding the provider.

dwPhoneInfoSize

dwPhoneInfoOffset

The size in bytes of the variably sized device field containing phone-specific information, and the offset in bytes from the beginning of this data structure.

The **dwPhoneInfoSize/Offset** field is intended to provide information about the attached phone device, such as the phone device manufacturer, the model name, the software version, and so on. This information can be useful when a user needs to call customer service with problems regarding the phone.

dwPermanentPhoneID

The permanent DWORD identifier by which the phone device is known in the system's configuration.

dwPhoneNameSize

dwPhoneNameOffset

The size in bytes of the variably sized device field containing a user configurable name for this phone device, and the offset in bytes from the beginning of this data structure This name can be configured by the user when configuring the phone device's service provider and is provided for the user's convenience.

dwStringFormat

The string format to be used with this phone device. This parameter uses the following STRINGFORMAT_ constants:

STRINGFORMAT_ASCII

ASCII string format using one byte per character.

STRINGFORMAT_DBCS

DBCS string format using two bytes per character.

STRINGFORMAT_UNICODE

Unicode string format using two bytes per character.

dwPhoneStates

The state changes for this phone device for which the application can be notified in a PHONE_STATE message. This parameter uses the following PHONESTATE_ constants:

PHONESTATE_OTHER

Phone status items other than those listed below have changed. The application should check the current phone status to determine which items have 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 connecting the phone to the PC is plugged in with TAPI active.

PHONESTATE_DISCONNECTED

The connection between the phone device and TAPI was just broken. This happens when the wire connecting the phone set to the PC is unplugged while TAPI is active.

PHONESTATE_OWNER

The number of owners for the phone device has changed.

PHONESTATE_MONITORS

The number of monitors for the phone device has changed.

PHONESTATE_DISPLAY

The display of the phone has changed.

PHONESTATE_LAMP

A lamp of the phone has changed.

PHONESTATE_RINGMODE

The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME

The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH

The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME

The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN

The handset's microphone gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH

The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME

The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN

The speakerphone's microphone gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH

The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME

The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN

The headset's microphone gain setting has changed.

PHONESTATE_SUSPEND

The application's use of the phone is temporarily suspended.

PHONESTATE_RESUME

The application's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC

The phone's device-specific information has changed.

PHONESTATE_REINIT

Items have changed in the configuration of phone devices. To become aware of these changes (as for the appearance of new phone devices) the application should reinitialize its use of TAPI.

PHONESTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **PHONECAPS** structure have changed. The application should use **phoneGetDevCaps** to read the updated structure.

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). A PHONE_STATE message with this value will normally be immediately followed by a PHONE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in PHONEERR_NODEVICE being returned to the application. If a service provider sends a PHONE_STATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

dwHookSwitchDevs

This field specifies the phone's hookswitch devices. This parameter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET

This is the ubiquitous, handheld, ear- and mouthpiece.

PHONEHOOKSWITCHDEV_SPEAKER

A built-in loudspeaker and microphone. This could also be an externally connected adjunct to the telephone set.

PHONEHOOKSWITCHDEV_HEADSET

This is a headset connected to the phone set.

dwHandsetHookSwitchModes

dwSpeakerHookSwitchModes

dwHeadsetHookSwitchModes

This field specifies the phone's hookswitch mode capabilities of the handset, speaker, or headset, respectively. The field is only meaningful if the hookswitch device is listed in **dwHookSwitchDevs**. This parameter uses the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK

The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC

The device's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER

The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER

The device's microphone and speaker are both active.

dwVolumeFlags

This field specifies the volume setting capabilities of the phone device's speaker components. If the bit in position PHONEHOOKSWITCHDEV_ is TRUE, the volume of the corresponding hookswitch device's speaker component can be adjusted with **phoneSetVolume**; otherwise FALSE.

dwGainFlags

This field specifies the gain setting capabilities of the phone device's microphone components. If the bit position PHONEHOOKSWITCHDEV_ is TRUE, the volume of the corresponding hookswitch device's microphone component can be adjusted with **phoneSetGain**; otherwise FALSE.

dwDisplayNumRows

This field specifies the display capabilities of the phone device by describing the number of rows in the phone display. The **dwDisplayNumRows** and **dwDisplayNumColumns** fields are both zero for a phone device without a display.

dwDisplayNumColumns

This field specifies the display capabilities of the phone device by describing the number of columns in the phone display. **dwDisplayNumRows** and **dwDisplayNumColumns** are both zero for a phone device without a display.

dwNumRingModes

The ring capabilities of the phone device. The phone is able to ring with **dwNumRingModes** different ring patterns, identified as 1, 2, through **dwNumRingModes** minus one. If the value of this field is zero, applications have no control over the ring mode of the phone. If the value of this field is greater than zero, it indicates the number of ring modes in addition to silence that are supported by the service provider. A value of 0 in the *lpdwRingMode* parameter of **phoneGetRing** or the *dwRingMode* parameter of **phoneSetRing** indicates silence (the phone is not ringing or should not be rung), and *dwRingMode* values of 1 to **dwNumRingModes** are valid ring modes for the phone device.

dwNumButtonLamps

This field specifies the number of button/lamps on the phone device that are detectable in TAPI. Button/lamps are identified by their ID. Valid button/lamp IDs range from zero to **dwNumButtonLamps** minus one. The keypad buttons '0', through '9', '*', and '#' are assigned the IDs 0 through 12.

dwButtonModesSize

dwButtonModesOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field containing the button modes of the phone's buttons. The array is indexed by button/lamp ID. This parameter uses the following PHONEBUTTONMODE_ constants:

PHONEBUTTONMODE_DUMMY

This value is used to describe a button/lamp position that has no corresponding button, but has only a lamp. If the phone provides any non-DUMMY buttons, the PHONE_BUTTON message will be sent to the application if a button is pressed at the phone device.

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 twelve keypad buttons, '0' through '9', '*', and '#'.

PHONEBUTTONMODE_LOCAL

The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY

The button is a "soft" button associated with the phone's display. A phone set can have zero or more display buttons.

dwButtonFunctionsSize

dwButtonFunctionsOffset

The size in bytes of the variably sized field containing the button modes of the phone's buttons, and the offset in bytes from the beginning of this data structure. This field uses the values specified by the PHONEBUTTONFUNCTION_ constants. The array is indexed by button/lamp ID.

dwLampModesSize

dwLampModesOffset

The size in bytes of the variably sized field containing the lamp modes of the phone's lamps, and the offset in bytes from the beginning of this data structure. The array is indexed by button/lamp ID. This parameter uses the following PHONELAMPMODE_ constants:

PHONELAMPMODE_BROKENFLUTTER

Broken flutter is the superposition of flash and flutter.

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 is winking.

PHONELAMPMODE_DUMMY

This value is used to describe a button/lamp position that has no corresponding lamp.

dwNumSetData

The number of different download areas in the phone device. The different areas are referred to using the data IDs 0, 1, , **dwNumSetData** minus one. If this field is zero, the phone does not support the download capability.

dwSetDataSize

dwSetDataOffset

The size in bytes of the variably sized field containing the sizes (in bytes) of the phone's download data areas, and the offset in bytes from the beginning of this data structure. This is an array with DWORD-sized elements indexed by data ID.

dwNumGetData

The number of different upload areas in the phone device. The different areas are referred to using the data IDs 0, 1, , **dwNumGetData** minus one. If this field is zero, the phone does not support the upload capability.

dwGetDataSize

dwGetDataOffset

The size in bytes of the variably sized field containing the sizes (in bytes) of the phone's upload data areas, and the offset in bytes from the beginning of this data structure. This is an array with DWORD-sized elements indexed by data ID.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized device-specific field, and the offset in bytes from the beginning of this data structure

dwDeviceClassesSize

dwDeviceClassesOffset

Length in bytes and offset from the beginning of PHONECAPS of a string consisting of the device class identifiers supported on this device for use with **phoneGetID**, separated by nulls; the last identifier in the list is followed by two nulls.

dwPhoneFeatures

These flags indicate which Telephony API functions can be invoked on the phone. A zero indicates the corresponding feature is not implemented and can never be invoked by the application on the phone; a one indicates the feature may be invoked depending on the device state and other factors. This field uses PHONEFEATURE_ constants.

dwSettableHandsetHookSwitchModes

The PHONEHOOKSWITCHMODE_ values which can be *set* on the handset using **phoneSetHookSwitch**.

dwSettableSpeakerHookSwitchModes

The PHONEHOOKSWITCHMODE_ values which can be *set* on the speakerphone using **phoneSetHookSwitch**.

dwSettableHeadsetHookSwitchModes

The PHONEHOOKSWITCHMODE_ values which can be *set* on the headset using **phoneSetHookSwitch**.

dwMonitoredHandsetHookSwitchModes

The PHONEHOOKSWITCHMODE_ values which can be *detected* and *reported* for the handset in a PHONE_STATE message and by **phoneGetHookSwitch**.

dwMonitoredSpeakerHookSwitchModes

The PHONEHOOKSWITCHMODE_ values which can be *detected* and *reported* for the speakerphone in a PHONE_STATE message and by **phoneGetHookSwitch**.

dwMonitoredHeadsetHookSwitchModes

The PHONEHOOKSWITCHMODE_ values which can be *detected* and *reported* for the headset in a PHONE_STATE message and by **phoneGetHookSwitch**.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

The members **dwDeviceClassesSize** through **dwMonitoredHeadsetHookSwitchModes** are available only to applications that open the phone device with an API version of 0x00020000 or greater.

See Also

PHONE_BUTTON, PHONE_CLOSE, PHONE_STATE, **phoneGetDevCaps**, **phoneGetHookSwitch**, **phoneGetRing**, **phoneSetGain**, **phoneSetHookSwitch**, **phoneSetRing**, **phoneSetVolume**

PHONEEXTENSIONID Overview

The **PHONEEXTENSIONID** structure describes an extension ID. Extension IDs are used to identify service provider-specific extensions for phone device classes.

```
typedef struct phoneextensionid_tag {
    DWORD  dwExtensionID0;
    DWORD  dwExtensionID1;
    DWORD  dwExtensionID2;
    DWORD  dwExtensionID3;
} PHONEEXTENSIONID, FAR *LPPHONEEXTENSIONID;
```

Members

dwExtensionID0

dwExtensionID1

dwExtensionID2

dwExtensionID3

These four DWORD-sized fields together specify a universally unique extension ID that identifies a phone device class extension.

Remarks

No extensibility.

Extension IDs are generated using an SDK-provided generation utility.

PHONEINITIALIZEEXPARAMS Overview

```
typedef struct phoneinitializeexparams_tag {
    DWORD  dwTotalSize;
    DWORD  dwNeededSize;
    DWORD  dwUsedSize;
    DWORD  dwOptions;

  union
  {
    HANDLE hEvent;
    HANDLE hCompletionPort;
  } Handles;

    DWORD dwCompletionKey;

} PHONEINITIALIZEEXPARAMS, FAR *LPPHONEINITIALIZEEXPARAMS;
```

Members

dwOptions

One of the PHONEINITIALIZEEXOPTION_ constants. Specifies the event notification mechanism the application desires to use.

hEvent

If **dwOptions** specifies PHONEINITIALIZEEXOPTION_USEEVENT, TAPI returns the event handle in this field.

hCompletionPort

If **dwOptions** specifies PHONEINITIALIZEEXOPTION_USECOMPLETIONPORT, the application must specify in this field the handle of an existing completion port opened using **CreateIoCompletionPort**.

dwCompletionKey

If **dwOptions** specifies PHONEINITIALIZEEXOPTION_USECOMPLETIONPORT, the application must specify in this field a value that will be returned through the *lpCompletionKey* parameter of **GetQueuedCompletionStatus** to identify the completion message as a telephony message.

Remarks

See **phoneInitializeEx** for further information on these options.

PHONEMESSAGE Overview

```
typedef struct phonemessage_tag {
    DWORD  hDevice;
    DWORD  dwMessageID;
    DWORD  dwCallbackInstance;
    DWORD  dwParam1;
    DWORD  dwParam2;
    DWORD  dwParam3;
} PHONEMESSAGE, FAR *LPPHONEMESSAGE;
```

Members

hDevice

A handle to a phone device.

dwMessageID

A phone message.

dwCallbackInstance

Instance data passed back to the application, which was specified by the application in **phoneInitializeEx**. This DWORD is not interpreted by TAPI.

dwParam1

A parameter for the message.

dwParam2

A parameter for the message.

dwParam3

A parameter for the message.

Remarks

For information about parameters values passed in this structure, see Phone Device Messages.

See Also

phoneInitializeEx

PHONESTATUS Overview

The **PHONESTATUS** structure describes the current status of a phone device.

```
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 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

This field provides a set of status flags for this phone device. This parameter uses the following PHONESTATUSFLAGS_ constants:

PHONESTATUSFLAGS_CONNECTED

Specifies whether the phone is currently connected to TAPI. TRUE if connected; FALSE otherwise.

PHONESTATUSFLAGS_SUSPENDED

Specifies whether TAPI's manipulation of the phone device is suspended. TRUE if suspended;

FALSE otherwise. An application's use of a phone device may be temporarily suspended when the switch wants to manipulate the phone in a way that cannot tolerate interference from the application.

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

The current ring volume of a phone device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum volume).

dwHandsetHookSwitchMode

The current hookswitch mode of the phone's handset. This parameter uses the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK

The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC

The device's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER

The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER

The device's microphone and speaker are both active.

dwHandsetVolume

The current speaker volume of the phone's handset device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum volume).

dwHandsetGain

The current microphone gain of the phone's handset device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum gain).

dwSpeakerHookSwitchMode

The current hookswitch mode of the phone's speakerphone. This parameter uses the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK

The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC

The device's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER

The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER

The device's microphone and speaker are both active.

dwSpeakerVolume

The current speaker volume of the phone's speaker device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum volume).

dwSpeakerGain

The current microphone gain of the phone's speaker device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum gain).

dwHeadsetHookSwitchMode

The current hookswitch mode of the phone's headset. This parameter uses the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK

The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC

The device's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER

The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER

The device's microphone and speaker are both active.

dwHeadsetVolume

The current speaker volume of the phone's headset device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum volume).

dwHeadsetGain

The current microphone gain of the phone's headset device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum gain).

dwDisplaySize

dwDisplayOffset

The size in bytes of the variably sized field containing the phone's current display information, and the offset in bytes, from the beginning of this data structure.

dwLampModesSize

dwLampModesOffset

The size in bytes of the variably sized field containing the phone's current lamp modes, and the offset in bytes from the beginning of this data structure.

dwOwnerNameSize

dwOwnerNameOffset

The size in bytes of the variably sized field containing 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 provided by the application when it invoked with **phoneInitialize** or **phoneInitializeEx** . If no application name was supplied, the application's file name is used instead. If the phone currently has no owner, **dwOwnerNameSize** is zero.

dwDevSpecificSize

dwDevSpecificOffset

The size in bytes of the variably sized device-specific field, and the offset in bytes from the beginning of this data structure

dwPhoneFeatures

These flags indicate which Telephony API functions can be invoked on the phone, considering the availability of the feature in the device capabilities, the current device state, and device ownership of the invoking application. A zero indicates the corresponding feature cannot be invoked by the application on the phone in its current state; a one indicates the feature can be invoked. This field uses PHONEFEATURE_ constants.

Remarks

Device-specific extensions should use the **DevSpecific** (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

The **dwPhoneFeatures** member is available only to applications that open the phone device with an API version of 0x00020000 or greater.

See Also

phoneInitialize, **phoneInitializeEx**

VARSTRING Overview

The **VARSTRING** structure is used for returning variably sized strings. It is used both by the line device class and the phone device class.

```
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 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 parameter uses the following STRINGFORMAT_ constants:

STRINGFORMAT_ASCII

ASCII string format using one byte per character. The actual string is a NULL-terminated ASCII string with the terminating NULL accounted for in the string size.

STRINGFORMAT_DBCS

DBCS string format using one or two bytes per character.

STRINGFORMAT_UNICODE

Unicode string format using two bytes per character.

STRINGFORMAT_BINARY

An array of unsigned characters that could be used for numeric values.

dwStringSize

dwStringOffset

The size in bytes of the variably sized device field containing the string information, and the offset in bytes from the beginning of this data structure.

Remarks

No extensibility.

If a string cannot be returned in a variable structure, the **dwStringSize** and **dwStringOffset** members are set in one of these ways:

- **dwStringSize** and **dwStringOffset** members are both set to zero.
- **dwStringOffset** is nonzero and **dwStringSize** is zero.
- **dwStringOffset** is nonzero, **dwStringSize** is 1, and the byte at the given offset is 0.

Constants

This section contains the reference for line device, phone device, and assisted telephony constants.

Line Device Constants

This section describes the constants used by the Telephony API. They are listed in alphabetical order.

LINEADDRCAPFLAGS_ Constants

The LINEADDRCAPFLAGS_ bit-flag constants are used in the **dwAddrCapFlags** field of the **LINEADDRESSCAPS** data structure to describe various Boolean address capabilities.

LINEADDRCAPFLAGS_FWDNUMRINGS
: Specifies whether the number of rings for a no-answer can be specified when forwarding calls on no answer. If TRUE, the valid range is provided in **LINEADDRESSCAPS**.

LINEADDRCAPFLAGS_PICKUPGROUPID
: Specifies whether a group ID is required for call pickup.

LINEADDRCAPFLAGS_SECURE
: Specifies whether calls on this address can be made secure at call-setup time.

LINEADDRCAPFLAGS_BLOCKIDDEFAULT
: Specifies whether the network by default sends or blocks caller ID information when making a call on this address. If TRUE, ID information is blocked by default; if FALSE, ID information is transmitted by default.

LINEADDRCAPFLAGS_BLOCKIDOVERRIDE
: Specifies whether the default setting for sending or blocking of caller ID information can be overridden per call. If TRUE, override is possible; if FALSE, override is not possible.

LINEADDRCAPFLAGS_DIALED
: Specifies whether a destination address can be dialed on this address for making a call. TRUE if a destination address must be dialed; FALSE if the destination address is fixed (as with a "hot phone").

LINEADDRCAPFLAGS_ORIGOFFHOOK
: Specifies whether the originating party's phone can automatically be taken offhook when making calls.

LINEADDRCAPFLAGS_DESTOFFHOOK
: Specifies whether the called party's phone can automatically be forced offhook when making calls.

LINEADDRCAPFLAGS_FWDCONSULT
: Specifies whether call forwarding involves the establishment of a consultation call.

LINEADDRCAPFLAGS_SETUPCONFNULL
: Specifies whether setting up a conference call starts out with an initial call (FALSE) or with no initial call (TRUE).

LINEADDRCAPFLAGS_AUTORECONNECT
: Specifies whether dropping a consultation call automatically reconnects to the call on consultation hold. TRUE if reconnect happens automatically; otherwise, FALSE.

LINEADDRCAPFLAGS_COMPLETIONID
: Specifies whether the completion IDs returned by **lineCompleteCall** are useful and unique. TRUE if useful; otherwise, FALSE.

LINEADDRCAPFLAGS_TRANSFERHELD
: Specifies whether a hard-held call can be transferred. Often, only calls on consultation hold can be transferred.

LINEADDRCAPFLAGS_TRANSFERMAKE
: Specifies whether an entirely new call can be established for use as a consultation call on transfer.

LINEADDRCAPFLAGS_CONFERENCEHELD
: Specifies whether a hard-held call can be conferenced to. Often, only calls on consultation hold can be added to as a conference call.

LINEADDRCAPFLAGS_CONFERENCEMAKE
: Specifies whether an entirely new call can be established for use as a consultation call (to add) on conference.

LINEADDRCAPFLAGS_PARTIALDIAL
: Specifies whether partial dialing is available.

LINEADDRCAPFLAGS_FWDSTATUSVALID

Specifies whether the forwarding status in the **LINEADDRESSSTATUS** structure for this address is valid or is at most a "best estimate," given absence of accurate confirmation by the switch or network.

LINEADDRCAPFLAGS_FWDINTEXTADDR
Specifies whether internal and external calls can be forwarded to different forwarding addresses. This flag is meaningful only if forwarding of internal and external calls can be controlled separately. This flag is TRUE if internal and external calls can be forwarded to different destination addresses; otherwise, it is FALSE.

LINEADDRCAPFLAGS_FWDBUSYNAADDR
Specifies whether call forwarding for busy and for no answer can use different forwarding addresses. This flag is meaningful only if forwarding for busy and for no answer can be controlled separately. This flag is TRUE if forwarding for busy and for no answer can use different destination addresses; otherwise, it is FALSE.

LINEADDRCAPFLAGS_ACCEPTTOALERT
TRUE if an offering call must be accepted using **lineAccept** to start alerting the users at both ends of the call; otherwise, FALSE. This is typically only used with ISDN.

LINEADDRCAPFLAGS_CONFDROP
TRUE if **lineDrop** on a conference call parent also has the side effect of dropping (that is, disconnecting) the other parties involved in the conference call; FALSE if dropping a conference call still allows the other parties to talk among themselves.

LINEADDRCAPFLAGS_PICKUPCALLWAIT
TRUE if **linePickup** can be used to pick up a call detected by the user as a call-waiting call; otherwise, FALSE.

LINEADDRCAPFLAGS_PREDICTIVEDIALER
This address has enhanced call progress monitoring capabilities which can be applied to outbound calls to determine call states such as *ringback*, *busy*, *specialinfo*, and *connected*, or the media mode of the device answering the call. It may also have the ability to automatically transfer outbound calls to another address when a call reaches any of a predefined set of states.

LINEADDRCAPFLAGS_QUEUE
This address is not associated with a particular station or physical device, but is a holding place where calls wait for further processing. The calls placed in the queue may receive a particular treatment. They may also be automatically transferred when a particular resource becomes available (for example, if the queue is an ACD queue and calls are waiting for an available agent).

LINEADDRCAPFLAGS_ROUTEPOINT
This address is not associated with a particular station or physical device, but is a holding place where calls wait for routing by the application (the application examines the called address, and can redirect the call to another address). The call may also be automatically transferred if a routing timeout expires (the switch usually assumes a default routing).

LINEADDRCAPFLAGS_HOLDMAKESNEW
When a call on this address is placed on hold (using **lineHold** or external action), a new call is automatically created (most likely in LINECALLSTATE_DIALTONE).

LINEADDRCAPFLAGS_NOINTERNALCALLS
The address is associated with a direct CO line (trunk), and cannot be used to make internal calls on a PBX. The application can use this indication to assist the user in selecting the correct call appearance to use for making a call. When this bit is off, it does not necessarily indicate that the address can be used to make internal calls, because the service provider may not be cognizant of the line type.

LINEADDRCAPFLAGS_NOEXTERNALCALLS
The address is associated with an internal line on a PBX that is restricted in such a way that it cannot be used to place calls to an address outside the switch (for example, it is an intercom). The application can use this indication to assist the user in selecting the correct call appearance to use for making a call. When this bit is off, it does not necessarily indicate that the address can be used to make external calls, because the service provider may not be cognizant of the line type.

LINEADDRCAPFLAGS_SETCALLINGID

The application may choose to set the **CallingPartyID** member in **LINECALLPARAMS** when calling **lineMakeCall** and other functions that accept a **LINECALLPARAMS** structure. The service provider will, if the content of the ID is acceptable and a path is available, pass the ID along to the called party to indicate the identity of the calling party.

No extensibility. All 32 bits are reserved.

LINEADDRESSMODE_ Constants

The LINEADDRESSMODE_ bit-flag constants describe various ways of identifying an address on a line device.

LINEADDRESSMODE_ADDRESSID
: The address is specified with a small integer in the range 0 to *dwNumAddresses* minus one, where *dwNumAddresses* is the value in the line's device capabilities.

LINEADDRESSMODE_DIALABLEADDR
: The address is specified through its phone number.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

This constant is used to select an address on a line on which to originate a call. The usual model would select the address by means of its address ID. Address IDs are the mechanism used to identify addresses throughout TAPI. However, in some environments, when making a call, it is often more practical to identify a call's originating address by phone number rather than by address ID. One example is in the possible modeling of large numbers of stations (third party) on the switch by means of one line device with many addresses. The line represents the set of all stations, and each station is mapped to an address with its own primary phone number and address ID.

LINEADDRESSSHARING_ Constants

The LINEADDRESSSHARING_ bit-flag constants describe various ways an address can be shared between lines.

LINEADDRESSSHARING_PRIVATE
: The address is private to the user's line; it is not assigned to any other station.

LINEADDRESSSHARING_BRIDGEDEXCL
: The address is bridged to one or more other stations. The first line to activate a call on the line will have exclusive access to the address and calls that may exist on it. Other lines will not be able to use the bridged address while it is in use.

LINEADDRESSSHARING_BRIDGEDNEW
: The address is bridged with one or more other stations. The first line to activate a call on the line will have exclusive access to only the corresponding call. Other applications that use the address will result in new and separate call appearances.

LINEADDRESSSHARING_BRIDGEDSHARED
: The address is bridged with one or more other lines. All bridged parties can share in calls on the address, which then functions as a conference.

LINEADDRESSSHARING_MONITORED
: This is an address whose idle/busy status is made available to this line.

No extensibility. All 32 bits are reserved.

The way in which an address is shared across lines can affect the behavior of that address. LINE_CALLSTATE and LINE_ADDRESSSTATE messages are sent to the application in response to activities by the bridging stations. It is through these messages that an application can track the status of the address.

LINEADDRESSSTATE_ Constants

The LINEADDRESSSTATE_ bit-flag constants describe various address status items.

LINEADDRESSSTATE_OTHER
: Address-status items other than those listed below have changed. The application should check the current address status to determine which items have changed.

LINEADDRESSSTATE_DEVSPECIFIC
: The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO
: The address has changed to idle (it is not in use by any stations).

LINEADDRESSSTATE_INUSEONE
: The address has changed from idle or in use by many bridged stations to being in use by just one station.

LINEADDRESSSTATE_INUSEMANY
: The monitored or bridged address has changed from being in use by one station to being in use by more than one station.

LINEADDRESSSTATE_NUMCALLS
: The number of calls on the address has changed. This is the result of events such as a new inbound call, an outbound call on the address, or a call changing its hold status. This flag covers changes in any of the fields **dwNumActiveCalls**, **dwNumOnHoldCalls** and **dwNumOnHoldPendingCalls** in the **LINEADDRESSSTATUS** structure. The application should check all three of these fields when it receives a LINE_ADDRESSSTATE (numCalls) message.

LINEADDRESSSTATE_FORWARD
: The forwarding status of the address has changed, including possibly the number of rings for determining a no-answer condition. The application should check the address status to determine details about the address's current forwarding status.

LINEADDRESSSTATE_TERMINALS
: The terminal settings for the address have changed.

LINEADDRESSSTATE_CAPSCHANGE
: Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINEADDRESSCAPS** structure for the address have changed. The application should use **lineGetAddressCaps** to read the updated structure. If a service provider sends a LINE_ADDRESSSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINEDEVSTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI to obtain the updated information.

No extensibility. All 32 bits are reserved.

An application is notified about changes to these status items in the LINE_ADDRESSSTATE message. The address's device capabilities indicate which address state changes can possibly be reported for this address.

LINEADDRFEATURE_ Constants

The LINEADDRFEATURE_ constants list the operations that can be invoked on an address.

LINEADDRFEATURE_FORWARD
The address can be forwarded.

LINEADDRFEATURE_MAKECALL
An outbound call can be placed on the address.

LINEADDRFEATURE_PICKUP
A call can be picked up at the address.

LINEADDRFEATURE_SETMEDIACONTROL
Media control can be set on this address.

LINEADDRFEATURE_SETTERMINAL
The terminal modes for this address can be set.

LINEADDRFEATURE_SETUPCONF
A conference call with a NULL initial call can be set up at this address.

LINEADDRFEATURE_UNCOMPLETECALL
Call completion requests can be canceled at this address.

LINEADDRFEATURE_UNPARK
Calls can be unparked using this address.

LINEADDRFEATURE_PICKUPHELD
The **linePickup** function (with a null destination address) can be used to pick up a call that is held on the address. This is normally used only in a bridged-exclusive arrangement.

LINEADDRFEATURE_PICKUPGROUP
The **linePickup** function can be used to pick up a call in the group.

LINEADDRFEATURE_PICKUPDIRECT
The **linePickup** function can be used to pick up a call on a specific address.

LINEADDRFEATURE_PICKUPWAITING
The **linePickup** function (with a null destination address) can be used to pick up a call waiting call. Note that this does not necessarily indicate that a waiting call is actually present, because it is often impossible for a telephony device to automatically detect such a call; it does, however, indicate that the hook-flash function will be invoked to attempt to switch to such a call.

Note If none of the new modified "PICKUP" bits is set in the **dwAddressFeatures** member in **LINEADDRESSSTATUS** but the LINEADDRFEATURE_PICKUP bit *is* set, then any of the pickup modes may work; the service provider has simply not specified which ones.

LINEADDRFEATURE_FORWARDFWD
The **lineForward** function can be used to forward calls on the address to other numbers. LINEADDRFEATURE_FORWARD will also be set.

LINEADDRFEATURE_FORWARDDND
The **lineForward** function (with an empty destination address) can be used to turn on the Do Not Disturb feature on the address. LINEADDRFEATURE_FORWARD will also be set.

Note If neither of the new modified "FORWARD" bits is set in the **dwAddressFeatures** member in **LINEADDRESSSTATUS** but the LINEADDRFEATURE_FORWARD bit *is* set, then any of the forward modes may work; the service provider has simply not specified which ones.

No extensibility. All 32 bits are reserved.

This constant is used both in **LINEADDRESSCAPS** (returned by **lineGetAddressCaps**) and in **LINEADDRESSSTATUS** (returned by **lineGetAddressStatus**). **LINEADDRESSCAPS** reports the

availability of the address features by the service provider (mainly the switch) for a given address. An application would make this determination when it initializes. The **LINEADDRESSSTATUS** structure reports, for a given address, which address features can actually be invoked while the address is in the current state. An application would make this determination dynamically after address-state changes, typically caused by call-related activities on the address.

LINEAGENTFEATURE_ Constants

The LINEAGENTFEATURE_ constants list features that are available for an agent on an address.

LINEAGENTFEATURE_SETAGENTGROUP
: The **lineSetAgentGroup** function may be invoked on this address.

LINEAGENTFEATURE_SETAGENTSTATE
: The **lineSetAgentState** function may be invoked on this address.

LINEAGENTFEATURE_SETAGENTACTIVITY
: The **lineSetAgentActivity** function may be invoked on this address.

LINEAGENTFEATURE_AGENTSPECIFIC
: The **lineAgentSpecific** function may be invoked on this address.

LINEAGENTFEATURE_GETAGENTACTIVITYLIST
: The **lineGetAgentActivityList** function may be invoked on this address.

LINEAGENTFEATURE_GETAGENTGROUPLIST
: The **lineGetAgentGroupList** function may be invoked on this address.

LINEAGENTSTATE_ Constants

The LINEAGENTSTATE_ constants list the state of an agent on an address.

LINEAGENTSTATE_LOGGEDOFF
: No agent is logged in on the address.

LINEAGENTSTATE_NOTREADY
: The agent is logged in, but occupied with a task other than serving a call (such as on a break). No additional calls should be routed to the agent.

LINEAGENTSTATE_READY
: The agent is ready to accept calls.

LINEAGENTSTATE_BUSYACD
: The agent is busy handling a call routed from an ACD queue.

LINEAGENTSTATE_BUSYINCOMING
: The agent is busy handling an inbound call that was not transferred to the agent from an ACD queue in which the agent is logged in.

LINEAGENTSTATE_BUSYOUTBOUND
: The agent is busy handling an outbound call, such as one routed from a predictive dialing queue.

LINEAGENTSTATE_BUSYOTHER
: The agent is busy handling another type of call, such as an outbound personal call not transferred to the agent by a predictive dialer. This value may also be used when the agent is known to be busy on a call but the type of call is unknown.

LINEAGENTSTATE_WORKINGAFTERCALL
: The agent has completed the preceding call, but is still occupied with work related to that call. The agent should not receive additional calls.

LINEAGENTSTATE_UNKNOWN
: The agent state is currently unknown, but may become known later. This may be a transitional state when a line or address is first opened.

LINEAGENTSTATE_UNAVAIL
: The agent state is unknown and will never become known. In **LINEADDRESSSTATUS**, this condition may also be represented by the **dwAgentState** field being set to 0.

The upper 16-bits of this set of constants are reserved for device-specific extensions.

LINEAGENTSTATUS_ Constants

The LINEAGENTSTATUS_ constants list the update status of the members of the **LINEAGENTSTATUS** for an agent.

LINEAGENTSTATE_GROUP
: The **Group List** in **LINEAGENTSTATUS** has been updated.

LINEAGENTSTATE_STATE
: The **dwState** member in **LINEAGENTSTATUS** has been updated.

LINEAGENTSTATE_NEXTSTATE
: The **dwNextState** member in **LINEAGENTSTATUS** has been updated.

LINEAGENTSTATE_ACTIVITY
: The **ActivityID**, **ActivitySize**, or **ActivityOffset** members in **LINEAGENTSTATUS** has been updated.

LINEAGENTSTATE_ACTIVITYLIST
: The **List** member in **LINEAGENTACTIVITYLIST** has been updated. The application can call **lineGetAgentActivityList** to get the updated list.

LINEAGENTSTATE_GROUPLIST
: The **List** member in **LINEAGENTGROUPLIST** has been updated. The application can call **lineGetAgentGroupList** to get the updated list.

LINEAGENTSTATE_CAPSCHANGE
: The capabilities in **LINEAGENTCAPS** have been updated. The application can call **lineGetAgentCaps** to get the updated list.

LINEAGENTSTATE_VALIDSTATES
: The **dwValidStates** member in **LINEAGENTSTATUS** has been updated.

LINEAGENTSTATE_VALIDNEXTSTATES
: The **dwValidNextStates** member in **LINEAGENTSTATUS** has been updated.

LINEANSWERMODE_ Constants

The LINEANSWERMODE_ bit-flag constants describe how an existing active call on a line device is affected by answering another *offering* call on the same line.

LINEANSWERMODE_NONE
: Answering another call on the same line has no effect on the existing active call on the line.

LINEANSWERMODE_DROP
: The currently active call will automatically be dropped.

LINEANSWERMODE_HOLD
: The currently active call will automatically be placed on hold.

No extensibility. All 32 bits are reserved.

If a call comes in (is offered) at the time another call is already active, the new call is connected to by invoking **lineAnswer**. The effect this has on the existing active call depends on the line's device capabilities. The first call may be unaffected, it may automatically be dropped, or it may automatically be placed on hold.

LINEBEARERMODE_ Constants

The LINEBEARERMODE_ bit-flag constants describe different bearer modes of a call. When an application makes a call, it can request a specific bearer mode. These modes are used to select a certain quality of service for the requested connection from the underlying telephone network. Bearer modes available on a given line are a device capability of the line.

LINEBEARERMODE_VOICE
: This is a regular 3.1 kHz analog voice-grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH
: This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation, and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE
: The multiuse mode defined by ISDN.

LINEBEARERMODE_DATA
: The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA
: The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING
: This corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a media stream by TAPI).

LINEBEARERMODE_PASSTHROUGH
: When a call is active in LINEBEARERMODE_PASSTHROUGH, the service provider gives direct access to the attached hardware for control by the application. This mode is used primarily by applications desiring temporary direct control over asynchronous modems, accessed through the Win32 comm functions, for the purpose of configuring or using special features not otherwise supported by the service provider.

LINEBEARERMODE_RESTRICTEDDATA
: Bearer service for digital data in which only the low-order seven bits of each octet may contain user data (for example, for Switched 56kbit/s service).

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Note that bearer mode and media mode are different notions. The bearer mode of a call is an indication of the quality of the telephone connection as provided primarily by the network. The media mode of a call is an indication of the type of information stream that is exchanged over that call. Group 3 fax or data modem are media modes that use a call with a 3.1 kHz voice bearer mode.

LINEBUSYMODE_ Constants

The LINEBUSYMODE_ bit-flag constants describe different busy signals that the switch or network may generate. These busy signals typically indicate that a different resource that is required to make a call is currently in use.

LINEBUSYMODE_STATION
: The busy signal indicates that the called party's station is busy. This is usually signaled with a *normal* busy tone.

LINEBUSYMODE_TRUNK
: The busy signal indicates that a trunk or circuit is busy. This is usually signaled with a *fast* busy tone.

LINEBUSYMODE_UNKNOWN
: The busy signal's specific mode is currently unknown but may become known later.

LINEBUSYMODE_UNAVAIL
: The busy signal's specific mode is unavailable and will not become known.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Note that busy signals may be sent as inband tones or out-of-band messages. TAPI makes no assumption about the specific signaling mechanism.

LINECALLCOMPLCOND_ Constants

The LINECALLCOMPLCOND_ bit-flag constants describe the conditions under which a call can be completed.

LINECALLCOMPLCOND_BUSY
: Completion of the call under busy conditions.

LINECALLCOMPLCOND_NOANSWER
: Completion of the call under ringback no answer conditions.

No extensibility. All 32 bits are reserved.

LINECALLCOMPLMODE_ Constants

The LINECALLCOMPLMODE_ bit-flag constants describe different ways in which a call can be completed.

LINECALLCOMPLMODE_CAMPON
: Queues the call until the call can be completed.

LINECALLCOMPLMODE_CALLBACK
: Requests the called station to return the call when it returns to idle.

LINECALLCOMPLMODE_INTRUDE
: Adds the application to the existing call at the called station (barge in).

LINECALLCOMPLMODE_MESSAGE
: Leaves a short predefined message for the called station (Leave Word Calling). The message to be sent is specified separately.

No extensibility. All 32 bits are reserved.

LINECALLFEATURE_ Constants

The LINECALLFEATURE_ constants list the operations that can be invoked on a call using this API.

Each of the LINECALLFEATURE_ values correspond to the TAPI operations with the same name. The list is not repeated here.

No extensibility. All 32 bits are reserved.

This constant is used both in **LINEADDRESSCAPS** (returned by **lineGetAddressCaps**) and in **LINECALLSTATUS** (returned by **lineGetCallStatus**). **LINEADDRESSCAPS** reports the availability of the call features on the specified address. An application would use this information when it initializes to determine what it may be able to do later when calls exist. For the specified call, **LINECALLSTATUS** reports which call features can be invoked while the call is in the current call state. The latter takes call privileges into account. An application would make this determination dynamically, after call state changes.

The LINECALLFEATURE_RELEASEUSERUSER value is new. No backward compatibility considerations. A service provider may elect to return this value in relevant fields (in **LINEADDRESSCAPS** and **LINECALLSTATUS**) even when older API versions have been negotiated on the line device.

The LINECALLFEATURE_SETTREATMENT, LINECALLFEATURE_SETQOS, and LINECALLFEATURE_SETCALLDATA constants are available for API versions 0x00020000 and greater.

LINECALLFEATURE2_ Constants

The LINECALLFEATURE2_ constants list the supplemental features available for conferencing, transferring, and parking calls.

LINECALLFEATURE2_NOHOLDCONFERENCE
: If this bit is on, a "no hold conference" can be created by using the LINECALLPARAMFLAGS_NOHOLDCONFERENCE option with **lineSetupConference**. The LINECALLFEATURE_SETUPCONF bit will also be on in the **dwCallFeatures** member.

LINECALLFEATURE2_ONESTEPTRANSFER
: If this bit is on, "one step transfer" can be created by using the LINECALLPARAMFLAGS_ONESTEPTRANSFER option with **lineSetupTransfer**. The LINECALLFEATURE_SETUPTRANSFER bit will also be on in the **dwCallFeatures** member.

LINECALLFEATURE2_COMPLCAMPON
: If this bit is on, the "camp on" feature can be invoked by using the LINECOMPLMODE_CAMPON option with **lineCompleteCall**. The LINECALLFEATURE_COMPLETECALL bit will also be on in the **dwCallFeatures** member.

LINECALLFEATURE2_COMPLCALLBACK
: If this bit is on, the "callback" feature can be invoked by using the LINECOMPLMODE_CALLBACK option with **lineCompleteCall**. The LINECALLFEATURE_COMPLETECALL bit will also be on in the **dwCallFeatures** member.

LINECALLFEATURE2_COMPLINTRUDE
: If this bit is on, the "intrude" feature can be invoked by using the LINECOMPLMODE_INTRUDE option with **lineCompleteCall**. The LINECALLFEATURE_COMPLETECALL bit will also be on in the **dwCallFeatures** member.

LINECALLFEATURE2_COMPLMESSAGE
: If this bit is on, the "leave message" feature can be invoked by using the LINECOMPLMODE_MESSAGE option with **lineCompleteCall**. The LINECALLFEATURE_COMPLETECALL bit will also be on in the **dwCallFeatures** member.

  Note If none of the "COMPL" bits is specified in the **dwCallFeature2** member in **LINECALLSTATUS** but LINECALLFEATURE_COMPLETECALL *is* specified, then it is possible that any of them will work, but the service provider has not specified which.

LINECALLFEATURE2_TRANSFERNORM
: If this bit is on, the **lineCompleteTransfer** function can be used to resolve the transfer as a normal transfer. The LINECALLFEATURE_COMPLETETRANSF bit will also be on in the **dwCallFeatures** member.

LINECALLFEATURE2_TRANSFERCONF
: If this bit is on, the **lineCompleteTransfer** function can be used to resolve the transfer as a three-way conference. The LINECALLFEATURE_COMPLETETRANSF bit will also be on in the **dwCallFeatures** member.

  Note If neither TRANSFERNORM nor TRANSFERCONF is specified in the **dwCallFeature2** member in **LINECALLSTATUS** but LINECALLFEATURE_COMPLETETRANSF *is* specified, then it is possible that either will work, but the service provider has not specified which.

LINECALLFEATURE2_PARKDIRECT
: If this bit is on, the "directed park" feature can be invoked by using the LINEPARKMODE_DIRECTED option with **linePark**. The LINECALLFEATURE_PARK bit will also be on in the **dwCallFeatures** member.

LINECALLFEATURE2_PARKNONDIRECT
: If this bit is on, the "non-directed park" feature can be invoked by using the

LINEPARKMODE_NONDIRECTED option with **linePark**. The LINECALLFEATURE_PARK bit will also be on in the **dwCallFeatures** field.

Note If neither PARKDIRECT nor PARKNONDIRECT is specified in the **dwCallFeature2** member in **LINECALLSTATUS** but LINECALLFEATURE_PARK *is* specified, then it is possible that either will work, but the service provider has not specified which.

LINECALLINFOSTATE_ Constants

The LINECALLINFOSTATE_ bit-flag constants describe various call information items about which an application may be notified in the LINE_CALLINFO message.

LINECALLINFOSTATE_OTHER
: Call information items other than those listed below have changed. The application should check the current call information to determine which items have changed.

LINECALLINFOSTATE_DEVSPECIFIC
: The device-specific field of the call-information record.

LINECALLINFOSTATE_BEARERMODE
: The bearer mode field of the call-information record.

LINECALLINFOSTATE_RATE
: The rate field of the call-information record.

LINECALLINFOSTATE_MEDIAMODE
: The media-mode field of the call-information record.

LINECALLINFOSTATE_APPSPECIFIC
: The application-specific field of the call-information record.

LINECALLINFOSTATE_CALLID
: The call ID field of the call-information record.

LINECALLINFOSTATE_RELATEDCALLID
: The related call ID field of the call-information record.

LINECALLINFOSTATE_ORIGIN
: The origin field of the call-information record.

LINECALLINFOSTATE_REASON
: The reason field of the call-information record.

LINECALLINFOSTATE_COMPLETIONID
: The completion ID field of the call-information record.

LINECALLINFOSTATE_NUMOWNERINCR
: The number of owner field in the call-information record was increased.

LINECALLINFOSTATE_NUMOWNERDECR
: The number of owner field in the call-information record was decreased.

LINECALLINFOSTATE_NUMMONITORS
: The number of monitors field in the call-information record has changed.

LINECALLINFOSTATE_TRUNK
: The trunk field of the call-information record.

LINECALLINFOSTATE_CALLERID
: One of the callerID-related fields of the call-information record.

LINECALLINFOSTATE_CALLEDID
: One of the calledID-related fields of the call-information record.

LINECALLINFOSTATE_CONNECTEDID
: One of the cconnectedID-related fields of the call-information record.

LINECALLINFOSTATE_REDIRECTIONID
: One of the redirectionID-related fields of the call-information record.

LINECALLINFOSTATE_REDIRECTINGID
: One of the redirectingID-related fields of the call-information record.

LINECALLINFOSTATE_DISPLAY
: The display field of the call-information record.

LINECALLINFOSTATE_USERUSERINFO
: The user-to-user information of the call-information record.

LINECALLINFOSTATE_HIGHLEVELCOMP
The high level compatibility field of the call-information record.

LINECALLINFOSTATE_LOWLEVELCOMP
The low level compatibility field of the call-information record.

LINECALLINFOSTATE_CHARGINGINFO
The charging information of the call-information record.

LINECALLINFOSTATE_TERMINAL
The terminal mode information of the call-information record.

LINECALLINFOSTATE_DIALPARAMS
The dial parameters of the call-information record.

LINECALLINFOSTATE_MONITORMODES
One or more of the digit, tone, or media monitoring fields in the call-information record.

LINECALLINFOSTATE_TREATMENT
The **CallTreatment** member in **LINECALLINFO** has been updated. This may occur in response to a **lineSetCallTreatment** function, a call state change, a call "vector" or other script controlling the call, or upon completion of playback of a recorded message (ordinarily, indicating a change to "silence" or "music").

LINECALLINFOSTATE_QOS
One or more of the **QOS** members in **LINECALLINFO** has been updated.

LINECALLINFOSTATE_CALLDATA
The **CallData** member in **LINECALLINFO** has been updated.

No extensibility. All 32 bits are reserved.

When changes occur in this data structure, a LINE_CALLINFO message is sent to the application. The parameters to this message are a handle to the call and an indication of the information item that has changed. The **LINEADDRESSCAPS** data structure also indicates which of these call information elements are valid for every call on the address.

LINECALLORIGIN_ Constants

The LINECALLORIGIN_ constants describe the origin of a call.

LINECALLORIGIN_OUTBOUND
: The call originated from this station as an outbound call.

LINECALLORIGIN_INTERNAL
: The call originated as an inbound call at a station internal to the same switching environment.

LINECALLORIGIN_EXTERNAL
: The call originated as an inbound call on an external line.

LINECALLORIGIN_UNKNOWN
: The call origin is currently unknown but may become known later.

LINECALLORIGIN_UNAVAIL
: The call origin is not available and will never become known for this call.

LINECALLORIGIN_CONFERENCE
: The call handle is for a conference call, that is, it is the application's connection to the conference bridge in the switch.

LINECALLORIGIN_INBOUND
: The call originated as an inbound call, but the service provider is unable to determine whether it came from another station on the same switch or from an external line.

No extensibility. All 32 bits are reserved.

The origin of a call is stored in the **dwOrigin** field of the call's **LINECALLINFO** structure.

For backward compatibility, it is the responsibility of the service provider to examine the negotiated API version on the line, and to not use the LINECALLORIGIN_INBOUND value if it is not supported on the negotiated version (LINECALLORIGIN_UNAVAIL may be substituted).

LINECALLPARAMFLAGS_ Constants

The LINECALLPARAMFLAGS_ constants describe various status flags about a call.

LINECALLPARAMFLAGS_SECURE
: The call should be set up as secure.

LINECALLPARAMFLAGS_IDLE
: The call should be originated on an idle call appearance and not join a call in progress. When using the **lineMakeCall** function, if the LINECALLPARAMFLAGS_IDLE value is *not* set and there is an existing call on the line, the function breaks into the existing call if necessary to make the new call. If there is no existing call, the function makes the new call as specified.

LINECALLPARAMFLAGS_BLOCKID
: The originator identity should be concealed (block caller ID).

LINECALLPARAMFLAGS_ORIGOFFHOOK
: The originator's phone should be automatically taken offhook.

LINECALLPARAMFLAGS_DESTOFFHOOK
: The called party's phone should be automatically taken offhook.

LINECALLPARAMFLAGS_NOHOLDCONFERENCE
: This bit is used only in conjunction with **lineSetupConference** and **linePrepareAddToConference**. The address to be conferenced with the current call is specified in the **TargetAddress** member in **LINECALLPARAMS**. The consultation call does not physically draw dial tone from the switch, but will progress through various call establishment states (for example, *dialing, proceeding*). When the consultation call reaches the *connected* state, the conference is automatically established; the original call, which had remained in the *connected* state, enters the *conferenced* state; the consultation call enters the *conferenced* state; the *hConfCall* enters the *connected* state. If the consultation call fails (enters the *disconnected* state followed by *idle*), the *hConfCall* also enters the *idle* state, and the original call (which may have been an existing conference, in the case of **linePrepareAddToConference**) remains in the *connected* state. The original party (or parties) never perceive the call has having gone *onhold*. This feature is often used to add a supervisor to an ACD agent call when necessary to monitor interactions with an irate caller.

LINECALLPARAMFLAGS_PREDICTIVEDIAL
: This bit is used only when placing a call on an address with predictive dialing capability (LINEADDRCAPFLAGS_PREDICTIVEDIALER is on in the **dwAddrCapFlags** member in **LINEADDRESSCAPS**). The bit must be on to enable the enhanced call progress and/or media device monitoring capabilities of the device. If this bit is not on, the call will be placed without enhanced call progress or media mode monitoring, and no automatic transfer will be initiated based on call state.

LINECALLPARAMFLAGS_ONESTEPTRANSFER
: This bit is used only in conjunction with **lineSetupTransfer**. It combines the operation of **lineSetupTransfer** followed by **lineDial** on the consultation call into a single step. The address to be dialed is specified in the **TargetAddress** member in **LINECALLPARAMS**. The original call is placed in *onholdpendingtranfer* state, just as if **lineSetupTransfer** were called normally, and the consultation call is established normally. The application must still call **lineCompleteTransfer** to effect the transfer. This feature is often used when invoking a transfer from a server over a third-party call control link, because such links frequently do not support the normal two-step process.

No extensibility. All 32 bits are reserved.

LINECALLPARTYID_ Constants

The LINECALLPARTYID_ bit-flag constants describe the nature of the information available about the parties involved in a call.

LINECALLPARTYID_BLOCKED
: Party ID information is not available because it has been blocked by the remote party.

LINECALLPARTYID_OUTOFAREA
: Caller ID information for the call is not available since it is not propagated all the way by the network.

LINECALLPARTYID_NAME
: Party ID information consists of the party's name (as, for example, from a directory kept inside the switch).

LINECALLPARTYID_ADDRESS
: Party ID information consists of the party's address in either canonical address format or dialable address format.

LINECALLPARTYID_PARTIAL
: Party ID information is valid but it is limited to partial information only.

LINECALLPARTYID_UNKNOWN
: Party ID information is currently unknown but may become known later.

LINECALLPARTYID_UNAVAIL
: Party ID information is not available and will not become available later. Information may be unavailable for unspecified reasons. For example, the information was not delivered by the network, it was ignored by the service provider, and so forth.

No extensibility. All 32 bits are reserved.

For each of the possible parties involved in a call, the LINECALLPARTYID_ constants describe how the party ID information is formatted. This information is supplied in the **LINECALLINFO** data structure.

LINECALLPRIVILEGE_ Constants

The LINECALLPRIVILEGE_ bit-flag constants describe the kinds of access rights or privileges an application with a call handle may have to the corresponding call.

LINECALLPRIVILEGE_NONE
: The application has no privileges to the call. The application's handle is void and should not be used.

LINECALLPRIVILEGE_MONITOR
: The application has monitor privileges to the call. These privileges allow the application to monitor state changes and query information and status about the call.

LINECALLPRIVILEGE_OWNER
: The application has owner privileges to the call. These privileges allow the application to manipulate the call in ways that affect the state of the call.

No extensibility. All 32 bits are reserved.

When a call handle is first provided to an application or whenever call privileges of that application are modified, the LINE_CALLSTATE message is sent to the application. When an application hands off a call, and if the receiving application does not already have a handle with owner privileges, then this message informs the application about its new privileges to the call.

LINECALLREASON_ Constants

The LINECALLREASON_ bit-flag constants describe the reason for a call.

LINECALLREASON_DIRECT
: This is a direct inbound or outbound call.

LINECALLREASON_FWDBUSY
: This call was forwarded from another extension that was busy at the time of the call.

LINECALLREASON_FWDNOANSWER
: The call was forwarded from another extension that didn't answer the call after some number of rings.

LINECALLREASON_FWDUNCOND
: The call was forwarded unconditionally from another number.

LINECALLREASON_PICKUP
: The call was picked up from another extension.

LINECALLREASON_UNPARK
: The call was retrieved as a parked call.

LINECALLREASON_REDIRECT
: The call was redirected to this station.

LINECALLREASON_CALLCOMPLETION
: The call was the result of a call completion request.

LINECALLREASON_TRANSFER
: The call has been transferred from another number.

LINECALLREASON_REMINDER
: The call is a reminder (or "recall") that the user has a call parked or on hold for (potentially) a long time.

LINECALLREASON_UNKNOWN
: The reason for the call is currently unknown but may become known later.

LINECALLREASON_UNAVAIL
: The reason for the call is unavailable and will not become known later.

LINECALLREASON_INTRUDE
: The call intruded onto the line, either by a call completion action invoked by another station or by operator action. Depending on switch implementation, the call may appear either in the *connected* state, or *conferenced* with an existing active call on the line.

LINECALLREASON_PARKED
: The call was parked on the address. Usually, it appears initially in the *onhold* state.

LINECALLREASON_CAMPEDON
: The call was camped on the address. Usually, it appears initially in the *onhold* state, and can be switched to using **lineSwapHold**. If an active call becomes *idle*, the camped-on call may change to the *offering* state and the device start ringing.

LINECALLREASON_ROUTEREQUEST
: The call appears on the address because the switch needs routing instructions from the application. The application should examine the **CalledID** member in **LINECALLINFO**, and use the **lineRedirect** function to provide a new dialable address for the call. If the call is to be blocked instead, the application may call **lineDrop**. If the application fails to take action within a switch-defined timeout period, a default action will be taken.

No extensibility. All 32 bits are reserved.

The LINECALLREASON_ constants are used in the **dwReason** field of the **LINECALLINFO** data structure.

For backward compatibility, it is the responsibility of the service provider to examine the negotiated API version on the line, and to not use these LINECALLREASON_ values if not supported on the negotiated

version (LINECALLREASON_UNAVAIL may be substituted).

LINECALLTREATMENT_ Constants

The LINECALLTREATMENT_ constants lists the manner in which calls of a certain state are treated and the sounds the calling party hears.

LINECALLTREATMENT_BUSY
: When the call is not actively connected to a device (*offering* or *onhold*), the party hears busy signal.

LINECALLTREATMENT_MUSIC
: When the call is not actively connected to a device (*offering* or *onhold*), the party hears music.

LINECALLTREATMENT_RINGBACK
: When the call is not actively connected to a device (*offering* or *onhold*), the party hears ringback tone.

LINECALLTREATMENT_SILENCE
: When the call is not actively connected to a device (*offering* or *onhold*), the party hears silence.

The value 0x00000000 is reserved to indicate that the service provider does not support call treatments. Values in the range 0x00000005 through 0x000000FF are reserved for future definition. Values in the range 0x00000100 through 0xFFFFFFFF are reserved for assignment by service providers, and may include identification of specific musical selections or recorded announcements.

LINECALLSELECT_ Constants

The LINECALLSELECT_ bit-flag constants describe which calls are to be selected.

LINECALLSELECT_LINE
: Selects calls on the specified line device.

LINECALLSELECT_ADDRESS
: Selects call on the specified address.

LINECALLSELECT_CALL
: Selects related calls to the specified call. For example, the parties in a conference call.

No extensibility. All 32 bits are reserved.

This constant is used in **lineGetNewCalls** and to specify a selection (scope) of the calls that are requested.

LINECALLSTATE_ Constants

The LINECALLSTATE_ bit-flag constants describe the call states a call can be in.

LINECALLSTATE_IDLE
: The call is idle—no call exists.

LINECALLSTATE_OFFERING
: The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the offering state does not automatically alert the user because alerting is done by the switch instructing the line to ring. It does not affect any call states.

LINECALLSTATE_ACCEPTED
: The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.

LINECALLSTATE_DIALTONE
: The call is receiving 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 being sent to the switch over the call. Note that **lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK
: The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY
: The call is receiving a busy tone. A busy tone indicates that the call cannot be completed—either a circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO
: Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED
: The call has been established and the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING
: Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD
: The call is on hold by the switch.

LINECALLSTATE_CONFERENCED
: The call is currently a member of a multi-party conference call.

LINECALLSTATE_ONHOLDPENDCONF
: The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENDTRANSFER
: The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED
: The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN
: The state of the call is not known. This may be due to limitations of the call-progress detection implementation.

The high-order 8 bits can define a device-specific substate of any of the predefined states, provided that one of the LINECALLSTATE_ bits defined above is also set. The low-order 24 bits are reserved for predefined states.

The LINECALLSTATE_ constants are used as parameters by the LINE_CALLSTATE message sent to the application. The message carries the new call state that the call transitioned to. These constants are also used as fields in the **LINECALLSTATUS** structure returned by the **lineGetCallStatus** function.

LINECARDOPTION_ Constants

The LINECARDOPTION_ constants define values used in the **dwOptions** field of the **LINECARDENTRY** structure returned as part of the **LINETRANSLATECAPS** structure returned by **lineGetTranslateCaps**. The LINECARDOPTION_ constant has the following values:

LINECARDOPTION_PREDEFINED
: This calling card is one of the predefined calling card definitions included by Microsoft with Win32 Telephony. It cannot be removed entirely using Dial Helper; if the user attempts to remove it, it will become HIDDEN. It thus continues to be accessible for copying of dialing rules.

LINECARDOPTION_HIDDEN
: This calling card has been hidden by the user. It is not shown by Dial Helper in the main listing of available calling cards, but will be shown in the list of cards from which dialing rules can be copied.

Not extensible. All 32 bits are reserved.

LINECONNECTEDMODE_ Constants

The LINECONNECTEDMODE_ bit-flag constants describe different substates of a connected call. A mode is available as call status to the application after the call state transitions to *connected*, and within the LINE_CALLSTATE message indicating the call is in LINECALLSTATE_CONNECTED. These values are used when the call is on an address that is shared (bridged) with other stations (see the LINEADDRESSSHARING_ constants), primarily electronic key systems. The LINECONNECTEDMODE_ constants have the following values:

LINECONNECTEDMODE_ACTIVE
: Indicates that the call is connected at the current station (the current station is a participant in the call). If the call state mode is 0 (zero), the application should assume that the value is "active" (which would be the situation on a non-bridged address). The mode may switch between ACTIVE and INACTIVE during a call if the user joins and leaves the call through manual action. In such a bridged situation, a **lineDrop** or **lineHold** operation may possibly not actually drop the call or place it on hold, because the status of other stations on the call may govern (for example, attempting to "hold" a call when other stations are participating won't be possible); instead, the call may simply be changed to the INACTIVE mode if it remains CONNECTED at other stations.

LINECONNECTEDMODE_INACTIVE
: Indicates that the call is active at one or more other stations, but the current station is not a participant in the call. If the call state mode is ZERO, the application should assume that the value is "active" (which would be the situation on a non-bridged address). A call in the INACTIVE state may be joined using **lineAnswer**. Many operations that are valid in calls in the CONNECTED state may be impossible in the INACTIVE mode, such as monitoring for tones and digits, because the station is not actually participating in the call; monitoring is usually suspended (although not canceled) while the call is in the INACTIVE mode.

LINECONNECTEDMODE_ACTIVEHELD
: Indicates that the station is an active participant in the call, but that the remote party has placed the call on hold (the other party considers the call to be in the *onhold* state). Normally, such information is available only when both endpoints of the call fall within the same switching domain.

LINECONNECTEDMODE_INACTIVEHELD
: Indicates that the station is not an active participant in the call, and that the remote party has placed the call on hold.

LINECONNECTEDMODE_CONFIRMED
: Indicates that the service provider received affirmative notification that the call has entered the connected state (for example, through answer supervision or similar mechanisms).

Not extensible. All 32 bits are reserved.

For backward compatibility, it is the responsibility of the service provider to examine the negotiated API version on the line, and to not use these LINECONNECTEDMODE_ values if not supported on the negotiated version. It should be noted that applications which are not cognizant of LINECONNECTEDMODE_ will most likely assume that a call that is in LINECALLSTATE_CONNECTED is in LINECONNECTEDMODE_ACTIVE.

The LINECONNECTEDMODE_ACTIVE and LINECONNECTEDMODE_INACTIVE values are used when the call is on an address that is shared with other stations (bridged; see LINEADDRESSSHARING_ constants), primarily electronic key systems. If the *connected* call state mode is "active," it means that the call is connected at the current station (the current station is a participant in the call). If the call state mode is "inactive," the call is active at one or more other stations, but the current station is not a participant in the call. If the call state mode is ZERO, the application should assume that the value is "active" (which would be the situation on a non-bridged address). The mode may switch between ACTIVE and INACTIVE during a call if the user joins and leaves the call through manual action.

In such a bridged situation, a **lineDrop** or **lineHold** operation may possibly not actually drop the call or

place it on hold, because the status of other stations on the call may govern (for example, attempting to "hold" a call when other stations are participating will not be possible); instead, the call may simply be changed to the INACTIVE mode if it remains *connected* at other stations. A call in the INACTIVE state may be joined using the **lineAnswer**.

Many operations that are valid in calls in the *connected* state may be impossible in the INACTIVE mode, such as monitoring for tones and digits, because the station is not actually participating in the call; monitoring is usually suspended (although not canceled) while the call is in the INACTIVE mode.

LINEDEVCAPFLAGS_ Constants

The LINEDEVCAPFLAGS_ bit-flag constants are a collection of Booleans describing various line device capabilities.

LINEDEVCAPFLAGS_CROSSADDRCONF
: Specifies whether calls on different addresses on this line can be conferenced.

LINEDEVCAPFLAGS_HIGHLEVCOMP
: Specifies whether high-level compatibility information elements are supported on this line.

LINEDEVCAPFLAGS_LOWLEVCOMP
: Specifies whether low-level compatibility information elements are supported on this line.

LINEDEVCAPFLAGS_MEDIACONTROL
: Specifies whether media-control operations are available for calls at this line.

LINEDEVCAPFLAGS_MULTIPLEADDR
: Specifies whether **lineMakeCall** or **lineDial** are able to deal with multiple addresses at once (as for inverse multiplexing).

LINEDEVCAPFLAGS_CLOSEDROP
: Specifies what happens when an open line is closed while the application has calls active on the line. If TRUE, the service provider drops (clears) all active calls on the line when the last application that has opened the line closes it with **lineClose**. If FALSE, the service provider does not drop active calls in such cases. Instead, the calls remain active and under control of external devices. A service provider typically sets this bit to FALSE if there is some other device that can keep the call alive, for example, if an analog line has the computer and phoneset both connect directly to them in a party-line configuration, the offhook phone will automatically keep the call active even after the computer powers down.

  Applications should check this flag to determine whether to warn the user (with an OK/Cancel dialog box) that active calls will be lost.

LINEDEVCAPFLAGS_DIALBILLING

LINEDEVCAPFLAGS_DIALQUIET

LINEDEVCAPFLAGS_DIALDIALTONE
: These flags indicate whether the "$", "@", or "W" dialable string modifier is supported for a given line device. It is TRUE if the modifier is supported; otherwise, FALSE. The "?" (prompt user to continue dialing) is never supported by a line device. These flags allow an application to determine up front which modifiers would result in the generation of a LINEERR. The application has the choice of pre-scanning dialable strings for unsupported characters or of passing the "raw" string from **lineTranslateAddress** directly to the provider as part of functions such as **lineMakeCall** or **lineDial** and let the function generate an error to tell it which unsupported modifier occurs first in the string.

No extensibility. All 32 bits are reserved.

LINEDEVSTATE_ Constants

The LINEDEVSTATE_ bit-flag constants describe various line status events.

LINEDEVSTATE_OTHER
: Device-status items other than those listed below have changed. The application should check the current device status to determine which items have 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 containing this constant. For example, in the United States, service providers send a message with this constant every six seconds.

LINEDEVSTATE_CONNECTED
: The line was previously disconnected and is now connected to TAPI.

LINEDEVSTATE_DISCONNECTED
: This line was previously connected and is now disconnected from TAPI.

LINEDEVSTATE_MSGWAITON
: The message waiting indicator is turned on.

LINEDEVSTATE_MSGWAITOFF
: The message waiting indicator is turned off.

LINEDEVSTATE_INSERVICE
: The line is connected to TAPI. This happens when TAPI is first activated or when the line wire is physically plugged in and in-service at the switch while TAPI is active.

LINEDEVSTATE_OUTOFSERVICE
: The line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE
: Maintenance is being performed on the line at the switch. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_OPEN
: The line has been opened by another application.

LINEDEVSTATE_CLOSE
: The line has been closed by another application.

LINEDEVSTATE_NUMCALLS
: The number of calls on the line device has changed.

LINEDEVSTATE_NUMCOMPLETIONS
: The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_TERMINALS
: The terminal settings have changed. This may happen, for example, if multiple line devices share terminals among them (for example, two lines sharing a phone terminal).

LINEDEVSTATE_ROAMMODE
: The roam mode of the line device has changed.

LINEDEVSTATE_BATTERY
: The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL
: The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC
: The line's device-specific information has changed.

LINEDEVSTATE_REINIT
: Items have changed in the configuration of line devices. To become aware of these changes (as for the appearance of new line devices) the application should reinitialize its use of TAPI.

LINEDEVSTATE_LOCK

The locked status of the line device has changed. (For more information, see LINEDEVSTATUSFLAGS_LOCKED in the following topic, LINEDEVSTATUSFLAGS_ Constants.)

LINEDEVSTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINEDEVCAPS** structure for the address have changed. The application should use **lineGetDevCaps** to read the updated structure. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINEDEVSTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI to obtain the updated information.

LINEDEVSTATE_CONFIGCHANGE

Indicates that configuration changes have been made to one or more of the media devices associated with the line device. The application, if it desires, may use **lineGetDevConfig** to read the updated information. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

LINEDEVSTATE_TRANSLATECHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINETRANSLATECAPS** structure have changed. The application should use **lineGetTranslateCaps** to read the updated structure. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINEDEVSTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI to obtain the updated information.

LINEDEVSTATE_COMPLCANCEL

Indicates that the call completion identified by the completion ID contained in the *dwParam2* parameter of the LINE_LINEDEVSTATE message has been externally canceled and is no longer considered valid (if that value were to be passed in a subsequent call to **lineUncompleteCall**, the function would fail with LINEERR_INVALCOMPLETIONID). If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications which have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

LINEDEVSTATE_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). A LINE_LINEDEVSTATE message with this value will normally be immediately followed by a LINE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in LINEERR_NODEVICE being returned to the application. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

No extensibility. All 32 bits are reserved.

LINEDEVSTATUSFLAGS_ Constants

The LINEDEVSTATUSFLAGS_ bit-flag constants describe a collection of Boolean line device status items.

LINEDEVSTATUSFLAGS_CONNECTED
: Specifies whether the line is connected to TAPI. If TRUE, the line is connected and TAPI is able to operate on the line device. If FALSE, the line is disconnected and the application is unable to control the line device through TAPI.

LINEDEVSTATUSFLAGS_MSGWAIT
: Indicates whether the line has a message waiting. If TRUE, a message is waiting; if FALSE, no message is waiting.

LINEDEVSTATUSFLAGS_INSERVICE
: Indicates whether the line is in service. If TRUE, the line is in service; if FALSE, the line is out of service.

LINEDEVSTATUSFLAGS_LOCKED
: Indicates whether the line is locked or unlocked. This bit is most often used with line devices associated with cellular phones. Many cellular phones have a security mechanism that requires the entry of a password to enable the phone to place calls. This bit may be used to indicate to applications that the phone is locked and cannot place calls until the password is entered on the user interface of the phone so that the application can present an appropriate alert to the user.

No extensibility. All 32 bits are reserved.

LINEDEVSTATUSFLAGS_ constants are used within the **dwDevStatusFlags** field of the **LINEDEVSTATUS** data structure.

LINEDIALTONEMODE_ Constants

The LINEDIALTONEMODE_ bit-flag constants describe different types of dial tones. A special dial tone typically carries a special meaning (as with message waiting).

LINEDIALTONEMODE_NORMAL
: This is a normal dial tone, which typically is a continuous tone.

LINEDIALTONEMODE_SPECIAL
: This is a special dial tone indicating that a certain condition (known by the switch or network) is currently in effect. Special dial tones typically use an interrupted tone. As with a normal dial tone, this indicates that the switch is ready to receive the number to be dialed.

LINEDIALTONEMODE_INTERNAL
: This is an internal dial tone, as within a PBX.

LINEDIALTONEMODE_EXTERNAL
: This is an external (public network) dial tone.

LINEDIALTONEMODE_UNKNOWN
: The dial tone mode is not currently known but may become known later.

LINEDIALTONEMODE_UNAVAIL
: The dial tone mode is unavailable and will not become known.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

The LINEDIALTONEMODE_ constants are used within the **LINECALLSTATUS** data structure for a call in the *dialtone* state.

LINEDIGITMODE_ Constants

The LINEDIGITMODE_ constants describe different types of inband digit generation.

LINEDIGITMODE_PULSE
: Uses rotary pulse sequences to signal digits. Valid digits are 0 through 9.

LINEDIGITMODE_DTMF
: Uses DTMF tones to signal digits. Valid digits are 0 through 9, '*', '#', 'A', 'B', 'C', and 'D'.

LINEDIGITMODE_DTMFEND
: Uses DTMF tones to signal digits and detect the down edges. Valid digits are 0 through 9, '*', '#', 'A', 'B', 'C', and 'D'.

No extensibility. All 32 bits are reserved.

A digit mode can be specified when generating or detecting digits. Note that pulse digits are generated by making and breaking the local loop circuit. These pulses are absorbed by the switch. The remote end merely observes this as a series of inband audio clicks. Detecting digits sent as pulses must therefore be able to detect these sequences of 1 to 10 audible clicks.

LINEDISCONNECTMODE_ Constants

The LINEDISCONNECTMODE_ bit-flag constants describe different reasons for a remote disconnect request. A disconnect mode is available as call status to the application after the call state transitions to *disconnected*.

LINEDISCONNECTMODE_NORMAL
: This is a normal disconnect request by the remote party. The call was terminated normally.

LINEDISCONNECTMODE_UNKNOWN
: The reason for the disconnect request is unknown but may become known later.

LINEDISCONNECTMODE_REJECT
: The remote user has rejected the call.

LINEDISCONNECTMODE_PICKUP
: The call was picked up from elsewhere.

LINEDISCONNECTMODE_FORWARDED
: The call was forwarded by the switch.

LINEDISCONNECTMODE_BUSY
: The remote user's station is busy.

LINEDISCONNECTMODE_NOANSWER
: The remote user's station does not answer.

LINEDISCONNECTMODE_BADADDRESS
: The destination address is invalid.

LINEDISCONNECTMODE_UNREACHABLE
: The remote user could not be reached.

LINEDISCONNECTMODE_CONGESTION
: The network is congested.

LINEDISCONNECTMODE_INCOMPATIBLE
: The remote user's station equipment is incompatible with the type of call requested.

LINEDISCONNECTMODE_UNAVAIL
: The reason for the disconnect is unavailable and will not become known later.

LINEDISCONNECTMODE_NODIALTONE
: A dial tone was not detected within a service-provider defined timeout, at a point during dialing when one was expected (such as at a "W" in the dialable string). This can also occur without a service-provider-defined timeout period or without a value specified in the **dwWaitForDialTone** member of the **LINEDIALPARAMS** structure.

LINEDISCONNECTMODE_NUMBERCHANGED
: The call could not be connected because the destination number has been changed, but automatic redirection to the new number is not provided.

LINEDISCONNECTMODE_OUTOFORDER
: The call could not be connected or was disconnected because the destination device is out of order (hardware failure).

LINEDISCONNECTMODE_TEMPFAILURE
: The call could not be connected or was disconnected because of a temporary failure in the network; the call can be reattempted later and will eventually complete.

LINEDISCONNECTMODE_QOSUNAVAIL
: The call could not be connected or was disconnected because the minimum quality of service could not be obtained or sustained. This differs from LINEDISCONNECTMODE_INCOMPATIBLE in that the lack of resources may be a temporary condition at the destination.

LINEDISCONNECTMODE_BLOCKED
: The call could not be connected because calls from the origination address are not being accepted at the destination address. This differs from LINEDISCONNECTMODE_REJECT in that blocking is implemented in the network (a passive reject) while a rejection is implemented in the destination

equipment (an active reject). The blocking may be due to a specific exclusion of the origination address, or because the destination accepts calls from only a selected set of origination address (closed user group).

LINEDISCONNECTMODE_DONOTDISTURB
The call could not be connected because the destination has invoked the Do Not Disturb feature.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

A remote disconnect request for a given call results in the call state transitioning to the *disconnected* state and a LINE_CALLSTATE message is sent to the application. The LINEDISCONNECTMODE_ information provides details about the remote disconnect request. It is available in the call's **LINECALLSTATUS** structure when the call is in the *disconnected* state. While a call is in this state, the application is still allowed to query the call's information and status. For example, user-to-user information that is received as part of the remote disconnect is available then. The application can clear a *disconnected* call by dropping the call.

For backward compatibility, it is the responsibility of the service provider to examine the negotiated API version on the line, and to not use this LINEDISCONNECTMODE_ value if it is not supported on the negotiated version (LINEDISCONNECTMODE_NORMAL or _UNKNOWN could be used instead).

LINEERR_ Constants

This is the list of error codes that TAPI may return when invoking operations on lines, addresses, or calls. Consult the individual function descriptions to determine which of these error codes a particular function may return.

LINEERR_ADDRESSBLOCKED
: The specified address is blocked from being dialed on the specified call.

LINEERR_ALLOCATED
: The line cannot be opened due to a persistent condition, such as that of a serial port being exclusively opened by another process.

LINEERR_BADDEVICEID
: The specified device ID or line device ID (such as in a *dwDeviceID* parameter) is invalid or out of range.

LINEERR_BEARERMODEUNAVAIL
: The call's bearer mode cannot be changed to the specified bearer mode.

LINEERR_CALLUNAVAIL
: All call appearances on the specified address are currently in use.

LINEERR_COMPLETIONOVERRUN
: The maximum number of outstanding call completions has been exceeded.

LINEERR_CONFERENCEFULL
: The maximum number of parties for a conference has been reached, or the requested number of parties cannot be satisfied.

LINEERR_DIALBILLING
: The dialable address parameter contains dialing control characters that are not processed by the service provider.

LINEERR_DIALQUIET
: The dialable address parameter contains dialing control characters that are not processed by the service provider.

LINEERR_DIALDIALTONE
: The dialable address parameter contains dialing control characters that are not processed by the service provider.

LINEERR_DIALPROMPT
: The dialable address parameter contains dialing control characters that are not processed by the service provider.

LINEERR_INCOMPATIBLEAPIVERSION
: The application requested an API version or version range that is either incompatible or cannot be supported by the Telephony API implementation and/or corresponding service provider.

LINEERR_INCOMPATIBLEEXTVERSION
: The application requested an extension version range that is either invalid or cannot be supported by the corresponding service provider.

LINEERR_INIFILECORRUPT
: The TELEPHON.INI file cannot be read or understood properly by TAPI because of internal inconsistencies or formatting problems. For example, the [Locations], [Cards], or [Countries] section of the TELEPHON.INI file may be corrupted or inconsistent.

LINEERR_INUSE
: The line device is in use and cannot currently be configured, allow a party to be added, allow a call to be answered, allow a call to be placed, or allow a call to be transferred.

LINEERR_INVALADDRESS
: A specified address is either invalid or not allowed. If invalid, the address contains invalid characters or digits, or the destination address contains dialing control characters (W, @, $, or ?) that are not supported by the service provider. If not allowed, the specified address is either not assigned to the

specified line or is not valid for address redirection.

LINEERR_INVALADDRESSID
The specified address ID is either invalid or out of range.

LINEERR_INVALADDRESSMODE
The specified address mode is invalid.

LINEERR_INVALADDRESSSTATE
dwAddressStates contains one or more bits that are not LINEADDRESSSTATE_ constants.

LINEERR_INVALAGENTACTIVITY
The specified agent activity is not valid.

LINEERR_INVALAGENTGROUP
The specified agent group information is not valid or contains errors. The requested action has not been carried out.

LINEERR_INVALAGENTID
The specified agent identifier is not valid.

LINEERR_INVALAGENTSKILL
The specified agent skill information is not valid.

LINEERR_INVALAGENTSTATE
The specified agent state is not valid or contains errors. No changes have been made to the agent state of the specified address.

LINEERR_INVALAGENTSUPERVISOR
The specified agent supervisor information is not valid.

LINEERR_INVALAPPHANDLE
The application handle (such as specified by a *hLineApp* parameter) or the appliction registration handle is invalid.

LINEERR_INVALAPPNAME
The specified application name is invalid. If an application name is specified by the application, it is assumed that the string does not contain any non-displayable characters, and is zero-terminated.

LINEERR_INVALBEARERMODE
The specified bearer mode is invalid.

LINEERR_INVALCALLCOMPLMODE
The specified completion is invalid.

LINEERR_INVALCALLHANDLE
The specified call handle is not valid. For example, the handle is not NULL but does not belong to the given line. In some cases, the specified call device handle is invalid.

LINEERR_INVALCALLPARAMS
The specified call parameters are invalid.

LINEERR_INVALCALLPRIVILEGE
The specified call privilege parameter is invalid.

LINEERR_INVALCALLSELECT
The specified select parameter is invalid.

LINEERR_INVALCALLSTATE
The current state of a call is not in a valid state for the requested operation.

LINEERR_INVALCALLSTATELIST
The specified call state list is invalid.

LINEERR_INVALCARD
The permanent card ID specified in *dwCard* could not be found in any entry in the [Cards] section in the registry.

LINEERR_INVALCOMPLETIONID
The completion ID is invalid.

LINEERR_INVALCONFCALLHANDLE
The specified call handle for the conference call is invalid or is not a handle for a conference call.

LINEERR_INVALCONSULTCALLHANDLE
The specified consultation call handle is invalid.

LINEERR_INVALCOUNTRYCODE
The specified country code is invalid.

LINEERR_INVALDEVICECLASS
The line device has no associated device for the given device class, or the specified line does not support the indicated device class.

LINEERR_INVALDIGITLIST
The specified digit list is invalid.

LINEERR_INVALDIGITMODE
The specified digit mode is invalid.

LINEERR_INVALDIGITS
The specified termination digits are not valid.

LINEERR_INVALFEATURE
The *dwFeature* parameter is invalid.

LINEERR_INVALGROUPID
The specified group ID is invalid.

LINEERR_INVALLINEHANDLE
The specified call, device, line device, or line handle is invalid.

LINEERR_INVALLINESTATE
The device configuration may not be changed in the current line state. The line may be in use by another application or a *dwLineStates* parameter contains one or more bits that are not LINEDEVSTATE_ constants. The LINEERR_INVALLINESTATE value can also indicate that the device is DISCONNECTED or OUTOFSERVICE. These states are indicated by setting the bits corresponding to the LINEDEVSTATUSFLAGS_CONNECTED and LINEDEVSTATUSFLAGS_INSERVICE values to 0 in the **dwDevStatusFlags** member of the **LINEDEVSTATUS** structure returned by the **lineGetLineDevStatus** function.

LINEERR_INVALLOCATION
The permanent location ID specified in *dwLocation* could not be found in any entry in the [Locations] section in the registry.

LINEERR_INVALMEDIALIST
The specified media list is invalid.

LINEERR_INVALMEDIAMODE
The list of media types to be monitored contains invalid information, the specified media mode parameter is invalid, or the service provider does not support the specified media mode. The media modes supported on the line are listed in the **dwMediaModes** field in the **LINEDEVCAPS** structure.

LINEERR_INVALMESSAGEID
The number given in *dwMessageID* is outside the range specified by the **dwNumCompletionMessages** field in the **LINEADDRESSCAPS** structure.

LINEERR_INVALPARAM
A parameter (such as *dwTollListOption*, *dwTranslateOptions*, *dwNumDigits*, or a structure pointed to by *lpDeviceConfig*) contains invalid values, a country code is invalid, a window handle is invalid, or the specified forward list parameter contains invalid information.

LINEERR_INVALPARKMODE
The specified park mode is invalid.

LINEERR_INVALPASSWORD
The specified password is not correct and the requested action has not been carried out.

LINEERR_INVALPOINTER
One or more of the specified pointer parameters (such as *lpCallList*, *lpdwAPIVersion*, *lpExtensionID*, *lpdwExtVersion*, *lphIcon*, *lpLineDevCaps*, and *lpToneList*) are invalid, or a required pointer to an output parameter is NULL.

LINEERR_INVALPRIVSELECT

An invalid flag or combination of flags was set for the *dwPrivileges* parameter.

LINEERR_INVALRATE
The specified bearer mode is invalid.

LINEERR_INVALREQUESTMODE
The specified request mode is invalid.

LINEERR_INVALTERMINALID
The specified terminal mode parameter is invalid.

LINEERR_INVALTERMINALMODE
The specified terminal modes parameter is invalid.

LINEERR_INVALTIMEOUT
Timeouts are not supported or the values of either or both of the parameters *dwFirstDigitTimeout* or *dwInterDigitTimeout* fall outside the valid range specified by the call's line-device capabilities.

LINEERR_INVALTONE
The specified custom tone does not represent a valid tone or is made up of too many frequencies or the specified tone structure does not describe a valid tone.

LINEERR_INVALTONELIST
The specified tone list is invalid.

LINEERR_INVALTONEMODE
The specified tone mode parameter is invalid.

LINEERR_INVALTRANSFERMODE
The specified transfer mode parameter is invalid.

LINEERR_LINEMAPPERFAILED
LINEMAPPER was the value passed in the *dwDeviceID* parameter, but no lines were found that match the requirements specified in the *lpCallParams* parameter.

LINEERR_NOCONFERENCE
The specified call is not a conference call handle or a participant call.

LINEERR_NODEVICE
The specified device ID, which was previously valid, is no longer accepted because the associated device has been removed from the system since TAPI was last initialized. Alternately, the line device has no associated device for the given device class.

LINEERR_NODRIVER
Either TAPIADDR.DLL could not be located or the telephone service provider for the specified device found that one of its components is missing or corrupt in a way that was not detected at initialization time. The user should be advised to use the Telephony Control Panel to correct the problem.

LINEERR_NOMEM
Insufficient memory to perform the operation, or unable to lock memory.

LINEERR_NOMULTIPLEINSTANCE
A Telephony Service Provider which does not support multiple instances is listed more than once in the [Providers] section in the registry. The application should advise the user to use the Telephony Control Panel to remove the duplicated driver.

LINEERR_NOREQUEST
There currently is no request pending of the indicated mode, or the application is no longer the highest-priority application for the specified request mode.

LINEERR_NOTOWNER
The application does not have owner privilege to the specified call.

LINEERR_NOTREGISTERED
The application is not registered as a request recipient for the indicated request mode.

LINEERR_OPERATIONFAILED
The operation failed for an unspecified or unknown reason.

LINEERR_OPERATIONUNAVAIL
The operation is not available, such as for the given device or specified line.

LINEERR_RATEUNAVAIL
The service provider currently does not have enough bandwidth available for the specified rate.

LINEERR_REINIT
If TAPI reinitialization has been requested, for example as a result of adding or removing a Telephony service provider, then **lineInitialize**, **lineInitializeEx**, or **lineOpen** requests are rejected with this error until the last application shuts down its usage of the API (using **lineShutdown**), at which time the new configuration becomes effective and applications are once again permitted to call **lineInitialize** or **lineInitializeEx**.

LINEERR_RESOURCEUNAVAIL
Insufficient resources to complete the operation. For example, a line cannot be opened due to a dynamic resource overcommitment.

LINEERR_STRUCTURETOOSMALL
The **dwTotalSize** field indicates insufficient space to contain the fixed portion of the specified structure.

LINEERR_TARGETNOTFOUND
A target for the call handoff was not found. This may occur if the named application did not open the same line with the LINECALLPRIVILEGE_OWNER bit in the *dwPrivileges* parameter of **lineOpen**. Or, in the case of media-mode handoff, no application has opened the same line with the LINECALLPRIVILEGE_OWNER bit in the *dwPrivileges* parameter of **lineOpen** and with the media mode specified in the *dwMediaMode* parameter having been specified in the *dwMediaModes* parameter of **lineOpen**.

LINEERR_TARGETSELF
The application invoking this operation is the target of the indirect handoff. That is, TAPI has determined that the calling application is also the highest priority application for the given media mode.

LINEERR_UNINITIALIZED
The operation was invoked before any application called **lineInitialize** , **lineInitializeEx**.

LINEERR_USERUSERINFOTOOBIG
The string containing user-to-user information exceeds the maximum number of bytes specified in the **dwUUIAcceptSize**, **dwUUIAnswerSize**, **dwUUIDropSize**, **dwUUIMakeCallSize**, or **dwUUISendUserUserInfoSize** field of **LINEDEVCAPS**, or the string containing user-to-user information is too long.

The values 0xC0000000 through 0xFFFFFFFF are available for device-specific extensions. The values 0x80000000 through 0xBFFFFFFF are reserved, while 0x00000000 through 0x7FFFFFFF are used as request IDs.

If an application gets an error return that it does not specifically handle (such as an error defined by a device-specific extension), it should treat the error as a LINEERR_OPERATIONFAILED (for an unspecified reason).

LINEFEATURE_ Constants

The LINEFEATURE_ constants list the operations that can be invoked on a line using this API.

LINEFEATURE_DEVSPECIFIC
: Device-specific operations can be used on the line.

LINEFEATURE_DEVSPECIFICFEAT
: Device-specific features can be used on the line.

LINEFEATURE_FORWARD
: Forwarding of all addresses can be used on the line.

LINEFEATURE_MAKECALL
: An outbound call can be placed on this line using an unspecified address.

LINEFEATURE_SETMEDIACONTROL
: Media control can be set on this line.

LINEFEATURE_SETTERMINAL
: Terminal modes for this line can be set.

LINEFEATURE_SETDEVSTATUS
: The **lineSetLineDevStatus** function may be invoked on the line device.

LINEFEATURE_FORWARDFWD
: The **lineForward** function can be used to forward calls on all address on the line to other numbers. LINEFEATURE_FORWARD will also be set.

LINEFEATURE_FORWARDDND
: The **lineForward** function (with an empty destination address) can be used to turn on the Do Not Disturb feature on all addresses on the line. LINEFEATURE_FORWARD will also be set.

> **Note** If neither of the new modified "FORWARD" bits is set in the **dwLineFeatures** member in **LINEDEVSTATUS** but the LINEFEATURE_FORWARD bit *is* set, then any of the forward modes may work; the service provider has simply not specified which ones.

No extensibility. All 32 bits are reserved.

The LINEFEATURE_ constants are used in **LINEDEVSTATUS** (returned by **lineGetLineDevStatus**). **LINEDEVSTATUS** reports, for a given line, which line features can actually be invoked while the line is in the current state. An application would make this determination dynamically after line state changes, typically caused by address or call-related activities on the line.

LINEFORWARDMODE_ Constants

The LINEFORWARDMODE_ bit-flag constants describe the conditions under which calls to an address can be forwarded.

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.

LINEFORWARDMODE_UNCONDINTERNAL
: Forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

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 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.

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 on no answer all calls that originated at a specified address (selective call forwarding).

LINEFORWARDMODE_BUSYNA
: Forward all calls on busy/no answer, irrespective of their origin. Use this value when forwarding for internal and external calls on busy and on no answer cannot be controlled separately.

LINEFORWARDMODE_BUSYNAINTERNAL
: Forward all internal calls on busy/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/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/no answer all calls that originated at a specified address (selective call forwarding).

LINEFORWARDMODE_UNKNOWN
: Calls are forwarded, but the conditions under which forwarding will occur are not known at this time. It

is possible that the conditions may become known at a future time.

LINEFORWARDMODE_UNAVAIL
Calls are forwarded, but the conditions under which forwarding will occur are not known, and will never be known by the service provider.

No extensibility. All 32 bits are reserved.

The bit flags defined by LINEFORWARDMODE_ are not orthogonal. Unconditional forwarding ignores any specific condition such as busy or no answer. If unconditional forwarding is not in effect, then forwarding on busy and on no answer can be controlled separately or not separately. If controlled separately, the LINEFORWARDMODE_BUSY and LINEFORWARDMODE_NOANSW flags can be used separately. If not controlled separately, the flag LINEFORWARDMODE_BUSYNA must be used. Similarly, if forwarding of internal and external calls can be controlled separately, then LINEFORWARDMODE_INTERNAL and LINEFORWARDMODE_EXTERNAL flags can be used separately; otherwise the combination is used.

Address capabilities indicate which forwarding modes are available for each address assigned to a line. An application can use **lineForward** to set forwarding conditions at the switch.

For backward compatibility, it is the responsibility of the service provider to examine the negotiated API version on the line, and to not use these LINEFORWARDMODE_ values if i not supported on the negotiated version.

LINEGATHERTERM_ Constants

The LINEGATHERTERM_ bit-flag constants describe the conditions under which buffered digit gathering is terminated.

LINEGATHERTERM_BUFFERFULL
: The requested number of digits has been gathered. The buffer is full.

LINEGATHERTERM_TERMDIGIT
: One of the termination digits matched a received digit. The matched termination digit is the last digit in the buffer.

LINEGATHERTERM_FIRSTTIMEOUT
: The first digit timeout expired. The buffer contains no digits.

LINEGATHERTERM_INTERTIMEOUT
: The inter-digit timeout expired. The buffer contains at least one digit.

LINEGATHERTERM_CANCEL
: The request was canceled by this application, by another application, or because the call terminated.

No extensibility. All 32 bits are reserved.

LINEGENERATETERM_ Constants

The LINEGENERATETERM_ bit-flag constants describe the conditions under which digit or tone generation is terminated.

LINEGENERATETERM_DONE
: The requested number of digits or requested tones have been generated for the requested duration.

LINEGENERATETERM_CANCEL
: The digit or tone generation request was canceled by this application, by another application, or because the call terminated. This value may also be returned when digit or tone generation cannot be completed due to internal failure of the service provider.

No extensibility. All 32 bits are reserved.

LINELOCATIONOPTION_ Constants

The LINELOCATIONOPTION_ constants define values used in the **dwOptions** field of the **LINELOCATIONENTRY** structure returned as part of the **LINETRANSLATECAPS** structure returned by **lineGetTranslateCaps**.

LINELOCATIONOPTION_PULSEDIAL
: The default dialing mode at this location is pulse dialing. If this bit is set, **lineTranslateAddress** will insert a "P" dial modifier at the beginning of the dialable string returned when this location is selected. If this bit is not set, **lineTranslateAddress** will insert a "T" dial modifier at the beginning of the dialable string.

Not extensible. All 32 bits are reserved.

LINEINITIALIZEEXOPTION_ Constants

The LINEINITIALIZEEXOPTION_ constants specify which event notification mechanism to use when initializing a session.

LINEINITIALIZEEXOPTION_USECOMPLETIONPORT
: The application desires to use the Completion Port event notification mechanism.

LINEINITIALIZEEXOPTION_USEEVENT
: The application desires to use the Event Handle event notification mechanism.

LINEINITIALIZEEXOPTION_USEHIDDENWINDOW
: The application desires to use the Hidden Window event notification mechanism.

See **lineInitializeEx** for further details on the operation of these options.

LINEMEDIACONTROL_ Constants

The LINEMEDIACONTROL_ bit-flag constants describe a set of generic operations on media streams. The interpretations are determined by the media stream. The line device must have the media-control capability for any media-control operation to be effective.

LINEMEDIACONTROL_NONE
: No change is to be made to the media stream.

LINEMEDIACONTROL_START
: Start the media stream.

LINEMEDIACONTROL_RESET
: Reset the media stream. Equivalent to an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE
: Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME
: Resume a paused media stream.

LINEMEDIACONTROL_RATEUP
: The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN
: The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL
: The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP
: The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN
: The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL
: The amplitude of the media stream is returned to normal.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Media control is provided to improve performance of actions on media streams in response to telephony-related events. The application should normally manage a media stream through the media-specific API. The media-control functionality provided here is not intended to replace the native media APIs.

Media-control actions can be associated with the detection of digits, the detection of tones, the transition into a call state, and the detection of a media mode. Consult a line's device capabilities to determine whether media control is available on the line.

LINEMEDIAMODE_ Constants

The LINEMEDIAMODE_ constants describe media modes (the data type of a media stream) on calls.

LINEMEDIAMODE_UNKNOWN
: A media stream exists but its mode is not currently known and may become known later. This would correspond to a call with an unclassified media type. In typical analog telephony environments, an inbound call's media mode may be unknown until after the call has been answered and the media stream has been filtered to make a determination.

  If the unknown media-mode flag is set, other media flags may also be set. This is used to signify that the media is unknown but that it is likely to be one of the other selected media modes.

LINEMEDIAMODE_INTERACTIVEVOICE
: The presence of voice energy on the call, and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE
: The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM
: A data modem session on the call.

LINEMEDIAMODE_G3FAX
: A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_TDD
: A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_G4FAX
: A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_DIGITALDATA
: Digital data is being sent or received over the call.

LINEMEDIAMODE_TELETEX
: A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX
: A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX
: A telex session on the call. Telex is one of the telematic services.

LINEMEDIAMODE_MIXED
: A mixed session on the call. Mixed is one of the ISDN telematic services.

LINEMEDIAMODE_ADSI
: An ADSI (Analog Display Services Interface) session on the call.

LINEMEDIAMODE_VOICEVIEW
: The media mode of the call is VoiceView.

All 32 bits are reserved.

Note that bearer mode and media mode are different notions. The bearer mode of a call is an indication of the quality of the telephone connection as provided primarily by the network. The media mode of a call is an indication of the type of information stream that is exchanged over that call. Group 3 fax or data modem are media modes that use a call with a 3.1 kHz voice bearer mode.

For backward compatibility, it is the responsibility of the service provider to examine the negotiated API version on the line, and to not use this LINEMEDIAMODE_ value if not supported on the negotiated version.

LINEOFFERINGMODE_ Constants

The LINEOFFERINGMODE_ bit-flag constants describe different substates of an offering call. A mode is available as call status to the application after the call state transitions to *offering*, and within the LINE_CALLSTATE message indicating the call is in LINECALLSTATE_OFFERING. These values are used when the call is on an address that is shared (bridged) with other stations (see LINEADDRESSSHARING_ Constants), primarily electronic key systems.

LINEOFFERINGMODE_ACTIVE
: Indicates that the call is alerting at the current station (will be accompanied by LINEDEVSTATE_RINGING messages), and if any application is set up to automatically answer, it may do so. If the call state mode is ZERO, the application should assume that the value is active (which would be the situation on a non-bridged address).

LINEOFFERINGMODE_INACTIVE
: Indicates that the call is being offered at more than one station, but the current station is not alerting (for example, it may be an attendant station where the offering status is advisory, such as blinking a light); software at the station set for automatic answering should preferably not answer the call, because this should be the prerogative at the primary (alerting) station, but **lineAnswer** may be used to connect the call.

Not extensible. All 32 bits are reserved.

For backward compatibility, it is the responsibility of the service provider to examine the negotiated API version on the line, and to not use these LINEOFFERINGMODE_ values if not supported on the negotiated version. It should be noted that applications which are not cognizant of LINEOFFERINGMODE_ will most likely assume that a call that is in LINECALLSTATE_OFFERING is in LINEOFFERINGMODE_ACTIVE.

The LINEOFFERINGMODE_ACTIVE and LINEOFFERINGMODE_INACTIVE values are used when the call is on an address that is shared with other stations (bridged; see LINEADDRESSSHARING_ Constants), primarily electronic key systems. If the *offering* call state mode is "active," it means that the call is alerting at the current station (will be accompanied by LINEDEVSTATE_RINGING messages), and if any application is set up to automatically answer, it may do so. If the call state mode is "inactive," the call is being offered at more than one station, but the current station is not alerting (for example, it may be an attendant station where the offering status is advisory, such as blinking a light); software at the station set for automatic answering should preferably not answer the call, because this should be the prerogative at the primary (alerting) station, but **lineAnswer** may be used to connect the call. If the call state mode is ZERO, the application should assume that the value is active (which would be the situation on a non-bridged address).

LINEOPENOPTION_ Constants

The LINEOPENOPTION_ constants list the available options for opening a line.

LINEOPENOPTION_SINGLEADDRESS
: The application should be informed of new calls created on the line device only if those calls appear on the address specified in the **dwAddressID** member in the **LINECALLPARAMS** structure pointed to by the *lpCallParams* parameter.

LINEOPENOPTION_PROXY
: The application is willing to handle requests from other applications that have the line open.

See **lineOpen** for further details on the operation of these options.

LINEPARKMODE_ Constants

The LINEPARKMODE_ bit-flag constants describe different ways of parking calls.

LINEPARKMODE_DIRECTED
: Specifies directed call park. The address where the call is to be parked must be supplied to the switch.

LINEPARKMODE_NONDIRECTED
: Specifies nondirected call park. The address where the call is parked is selected by the switch and provided by the switch to the application.

No extensibility. All 32 bits are reserved.

The LINEPARKMODE_ constants are used when parking a call. Consult a line's address device capabilities to find out which park mode is available.

LINEPROXYREQUEST_ Constants

These constants are used in two contexts. First, they may be used in an array of DWORD values in the **LINECALLPARAMS** structure passed in with **lineOpen** when the LINEOPENOPTION_PROXY option is specified, to indicate which functions the application is willing to handle. Second, they are used in the **LINEPROXYBUFFER** passed to the handler application by a LINE_PROXYREQUEST message to indicate the type of request that is to be processed and the format of the data in the buffer.

LINEPROXYREQUEST_SETAGENTGROUP
Associated with **lineSetAgentGroup**.

LINEPROXYREQUEST_SETAGENTSTATE
Associated with **lineSetAgentState**.

LINEPROXYREQUEST_SETAGENTACTIVITY
Associated with **lineSetAgentActivity**.

LINEPROXYREQUEST_GETAGENTCAPS
Associated with **lineGetAgentCaps**.

LINEPROXYREQUEST_GETAGENTSTATUS
Associated with **lineGetAgentStatus**.

LINEPROXYREQUEST_AGENTSPECIFIC
Associated with **lineAgentSpecific**.

LINEPROXYREQUEST_GETAGENTACTIVITYLIST
Associated with **lineGetAgentActivityList**.

LINEPROXYREQUEST_GETAGENTGROUPLIST
Associated with **lineGetAgentGroupList**.

LINEREMOVEFROMCONF_ Constants

The LINEREMOVEFROMCONF_ scalar constants describe how parties participating in a conference call can be removed from a conference call.

LINEREMOVEFROMCONF_NONE
: Parties cannot be removed from the conference call.

LINEREMOVEFROMCONF_LAST
: Only the most recently added party can be removed from the conference call

LINEREMOVEFROMCONF_ANY
: Any participating party can be removed from the conference call.

No extensibility. All 32 bits are reserved.

LINEREQUESTMODE_ Constants

The LINEREQUESTMODE_ bit-flag constants describe different types of telephony requests that can be made from one application to another.

LINEREQUESTMODE_MAKECALL
A **tapiRequestMakeCall** request.

No extensibility. All 32 bits are reserved.

LINEROAMMODE_ Constants

The LINEROAMMODE_ bit-flag constants describe the roaming status of a line device.

LINEROAMMODE_UNKNOWN
: The roam mode is currently unknown but may become known later.

LINEROAMMODE_UNAVAIL
: The roam mode is unavailable and will not be known.

LINEROAMMODE_HOME
: The line is connected to the home network node.

LINEROAMMODE_ROAMA
: The line is connected to the Roam-A carrier and calls are charged accordingly.

LINEROAMMODE_ROAMB
: The line is connected to the Roam-B carrier and calls are charged accordingly.

No extensibility. All 32 bits are reserved.

LINESPECIALINFO_ Constants

The LINESPECIALINFO_ bit-flag constants describes special information signals that the network may use to report various reporting and network observation operations. They are special coded tone sequences transmitted at the beginning of network advisory recorded announcements.

LINESPECIALINFO_NOCIRCUIT
: This special information tone precedes a no circuit or an emergency announcement (trunk blockage category).

LINESPECIALINFO_CUSTIRREG
: This special information tone precedes a vacant number, AIS, Centrex number change and nonworking station, access code not dialed or dialed in error, or manual intercept operator message (customer irregularity category). LINESPECIALINFO_CUSTIRREG is also reported when billing information is rejected and when the dialed address is blocked at the switch.

LINESPECIALINFO_REORDER
: This special information tone precedes a reorder announcement (equipment irregularity category). LINESPECIALINFO_REORDER is also reported when the telephone is kept offhook too long.

LINESPECIALINFO_UNKNOWN
: Specifics about the special information tone are currently unknown but may become known later.

LINESPECIALINFO_UNAVAIL
: Specifics about the special information tone are unavailable and will not become known.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Special information tones are defined for advisory messages and are not normally used for billing or supervisory purpose.

LINETERMDEV_ Constants

The LINETERMDEV_ bit-flag constants describe different types of terminal devices.

LINETERMDEV_PHONE
: The terminal is a phone set.

LINETERMDEV_HEADSET
: The terminal is a headset.

LINETERMDEV_SPEAKER
: The terminal is an external speaker and microphone.

No extensibility. All 32 bits are reserved.

These constants are used to characterize a line's terminal device and help an application to determine the nature of a terminal device.

LINETERMMODE_ Constants

The LINETERMMODE_ bit-flag constants describe different types of events on a phone line that can be routed to a terminal device.

LINETERMMODE_BUTTONS
: These are button-press events sent from the terminal to the line.

LINETERMMODE_LAMPS
: These are lamp events sent from the line to the terminal.

LINETERMMODE_DISPLAY
: This is display information sent from the line to the terminal.

LINETERMMODE_RINGER
: This is ringer-control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH
: These are hookswitch events sent from the terminal to the line.

LINETERMMODE_MEDIATOLINE
: This is the unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE
: This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT
: This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when the routing of both unidirectional channels of a call's media stream cannot be controlled independently.

No extensibility. All 32 bits are reserved.

These constants describe the classes of control and information streams that can be routed directly between a line device and a terminal device (such as a phone set).

LINETERMSHARING_ Constants

The LINETERMSHARING_ bit-flag constants describe different ways in which a terminal can be shared between line devices, addresses, or calls.

LINETERMSHARING_PRIVATE
: The terminal device is private to a single line device.

LINETERMSHARING_SHAREDEXCL
: The terminal device can be used by multiple lines. The last line device to do a **lineSetTerminal** to the terminal for a given terminal mode will have exclusive connection to the terminal for that mode.

LINETERMSHARING_SHAREDCONF
: The terminal device can be used by multiple lines. The **lineSetTerminal** requests of the various terminals end up being merged or conferenced at the terminal.

No extensibility. All 32 bits are reserved.

These constants describe the classes of control and information streams that can be routed directly between a line device and a terminal device (such as a phone set).

LINETOLLLISTOPTION_ Constants

The LINETOLLLISTOPTION_ bit-flag constants describe options for manipulating a toll list.

LINETOLLLISTOPTION_ADD
: A prefix is to be added to the toll list.

LINETOLLLISTOPTION_REMOVE
: A prefix is to be removed from the toll list.

No extensibility. All 32 bits are reserved.

LINETONEMODE_ Constants

The LINETONEMODE_ constants describe different selections that are used when generating line tones.

LINETONEMODE_CUSTOM
: The tone is a custom tone defined by its component frequencies, of type **LINEGENERATETONE**.

LINETONEMODE_RINGBACK
: The tone is ringback tone. Exact definition is service-provider defined.

LINETONEMODE_BUSY
: The tone is a busy tone. Exact definition is service-provider defined.

LINETONEMODE_BEEP
: The tone is a beep, such as that used to announce the beginning of a recording. Exact definition is service-provider defined.

LINETONEMODE_BILLING
: The tone is a billing information tone such as a credit card prompt tone. Exact definition is service-provider defined.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

These constants are used to define tones to be generated inband over a call to the remote party. Note that tone detection of non-custom tones does not use these constants.

LINETRANSFERMODE_ Constants

The LINETRANSFERMODE_ bit-flag constants describe different ways of resolving call transfer requests.

LINETRANSFERMODE_TRANSFER
: The transfer is resolved by transferring the initial call to the consultation call. Both calls will become idle to the application.

LINETRANSFERMODE_CONFERENCE
: The transfer is resolved by establishing a three-way conference between the application, the party connected to the initial call, and the party connected to the consultation call. A conference call is created when this option is selected.

No extensibility. All 32 bits are reserved.

LINETRANSLATEOPTION_ Constants

The LINETRANSLATEOPTION_ bit-flag constant describes an option used by address translation.

LINETRANSLATEOPTION_CARDOVERRIDE
: The default calling card is to be overidden with a specified one.

LINETRANSLATEOPTION_CANCELCALLWAITING
: If a Cancel Call Waiting string is defined for the location, setting this bit will cause that string to be inserted at the beginning of the dialable string. This is commonly used by data modem and fax applications to prevent interruption of calls by call waiting beeps. If no Cancel Call Waiting string is defined for the location, this bit has no affect. Note that applications using this bit are advised to also set the LINECALLPARAMFLAGS_SECURE bit in the **dwCallParamFlags** field of the **LINECALLPARAMS** structure passed in to **lineMakeCall** through the *lpCallParams* parameter, so that if the line device uses a mechanism other than dialable digits to suppress call interrupts that that mechanism will be invoked.

LINETRANSLATEOPTION_FORCELOCAL
: If the number is local but would have been translated as a long distance call (LINETRANSLATERESULT_INTOLLLIST bit set in the **LINETRANSLATEOUTPUT** structure), this option will force it to be translated as local. This is a temporary override of the toll list setting.

LINETRANSLATEOPTION_FORCELD
: If the address could potentially have been a toll call, but would have been translated as a local call (LINETRANSLATERESULT_NOTINTOLLLIST bit set in the **LINETRANSLATEOUTPUT** structure), this option will force it to be translated as long distance. This is a temporary override of the toll list setting.

No extensibility. All 32 bits are reserved.

LINETRANSLATERESULT_ Constants

The LINETRANSLATERESULT_ bit-flag constants describe various results of an address translation.

LINETRANSLATERESULT_CANONICAL
: Indicates that the input string was in valid canonical format.

LINETRANSLATERESULT_INTERNATIONAL
: Indicates that the call is being treated as an international call (country code specified in the destination address is different from the country code specified for the **CurrentLocation**).

LINETRANSLATERESULT_LONGDISTANCE
: Indicates that the call is being treated as a long distance call (country code specified in the destination address is the same but area code is different from those specified for the **CurrentLocation**).

LINETRANSLATERESULT_LOCAL
: Indicates that the call is being treated as a local call (country code and area code specified in the destination address are the same as those specified for the **CurrentLocation**).

LINETRANSLATERESULT_INTOLLLIST
: Indicates that the local call is being dialed as long distance because the country has toll calling and the prefix appears in the **TollPrefixList** of the **CurrentLocation**.

LINETRANSLATERESULT_NOTINTOLLLIST
: Indicates that the country supports toll calling but the prefix does not appear in the **TollPrefixList**, so the call is dialed as a local call. Note that if both INTOLLIST and NOTINTOLLIST are off, the current country does not support toll prefixes, and user-interface elements related to toll prefixes should not be presented to the user; if either such bit is on, the country does support toll lists, and the related user-interface elements should be enabled.

LINETRANSLATERESULT_DIALBILLING
: Indicates that the returned address contains a "$".

LINETRANSLATERESULT_DIALQUIET
: Indicates that the returned address contain a "@".

LINETRANSLATERESULT_DIALDIALTONE
: Indicates that the returned address contains a "W".

LINETRANSLATERESULT_DIALPROMPT
: Indicates that the returned address contains a "?".

LINETRANSLATERESULT_VOICEDETECT
: Indicates that the returned dialable address contains a ":".

> **Note** The ":" (colon) character will be added to the list of characters which can be embedded in a dialable string and passed into destination addresses. Attempting to pass it from an application to a line device that supports an API version less than 0x00020000 will most likely result in LINEERR_INVALADDRESS, or possibly in the character being ignored entirely. The meaning of this character is "Pause until a voice prompt is detected, then continue dialing"; it is intended for use when automatically dialing into systems that give voice prompts, such as long distance calling card processors.

No extensibility. All 32 bits are reserved.

LINETSPIOPTION_ Constants

LINETSPIOPTION_NONREENTRANT

TAPI should call functions in this service provider one at a time; it should wait from each function to return (but *not* for asynchronous functions to complete) before calling the same or another function, either on the same or a different thread, on the same or a different processor.

Phone Device Constants

The following reference contains the constants for phone devices.

PHONEBUTTONFUNCTION_ Constants

The PHONEBUTTONFUNCTION_ scalar constants describe the functions commonly assigned to buttons on telephone sets.

PHONEBUTTONFUNCTION_UNKNOWN
: A "dummy" function assignment that indicates that the exact function of the button is unknown or has not been assigned.

PHONEBUTTONFUNCTION_CONFERENCE
: Initiates a conference call or adds a call to a conference call.

PHONEBUTTONFUNCTION_TRANSFER
: Initiates a call transfer or completes the transfer of a call.

PHONEBUTTONFUNCTION_DROP
: Drops the active call.

PHONEBUTTONFUNCTION_HOLD
: Places the active call on hold.

PHONEBUTTONFUNCTION_RECALL
: Unholds a call.

PHONEBUTTONFUNCTION_DISCONNECT
: Disconnects a call, such as after initiating a transfer.

PHONEBUTTONFUNCTION_CONNECT
: Reconnects a call that is on consultation hold.

PHONEBUTTONFUNCTION_MSGWAITON
: Turns on a message waiting lamp.

PHONEBUTTONFUNCTION_MSGWAITOFF
: Turns off a message waiting lamp.

PHONEBUTTONFUNCTION_SELECTRING
: Allows the user to select the ring pattern of the phone.

PHONEBUTTONFUNCTION_ABBREVDIAL
: The number to be dialed will be indicated using a short, abbreviated number consisting of one digit or a few digits.

PHONEBUTTONFUNCTION_FORWARD
: Initiates or changes call forwarding to this phone.

PHONEBUTTONFUNCTION_PICKUP
: Picks up a call ringing on another phone.

PHONEBUTTONFUNCTION_RINGAGAIN
: Initiates a request to be notified if a call cannot be completed normally because of a busy signal or no answer.

PHONEBUTTONFUNCTION_PARK
: Parks the active call on another phone, placing it on hold there.

PHONEBUTTONFUNCTION_REJECT
: Rejects an inbound call before the call has been answered.

PHONEBUTTONFUNCTION_REDIRECT
: Redirects an inbound call to another extension before the call has been answered.

PHONEBUTTONFUNCTION_MUTE
: Mutes the phone's microphone device.

PHONEBUTTONFUNCTION_VOLUMEUP
: Increases the volume of audio through the phone's handset speaker or speakerphone.

PHONEBUTTONFUNCTION_VOLUMEDOWN
: Decreases the volume of audio through the phone's handset speaker or speakerphone.

PHONEBUTTONFUNCTION_SPEAKERON

Turns the phone's external speaker on.

PHONEBUTTONFUNCTION_SPEAKEROFF
Turns the phone's external speaker off.

PHONEBUTTONFUNCTION_FLASH
Generates the equivalent of an onhook/offhook sequence. A flash typically indicates that any digits typed next are to be understood as commands to the switch. On many switches, places an active call on consultation hold.

PHONEBUTTONFUNCTION_DATAON
Indicates that the next call is a data call.

PHONEBUTTONFUNCTION_DATAOFF
Indicates that the next call is not a data call.

PHONEBUTTONFUNCTION_DONOTDISTURB
Places the phone in "do not disturb" mode; incoming calls receive a busy signal or are forwarded to an operator or voice mail system.

PHONEBUTTONFUNCTION_INTERCOM
Connects to the intercom to broadcast a page.

PHONEBUTTONFUNCTION_BRIDGEDAPP
Selects a particular appearance of a bridged address.

PHONEBUTTONFUNCTION_BUSY
Makes the phone appear "busy" to incoming calls.

PHONEBUTTONFUNCTION_CALLAPP
Selects a particular call appearance.

PHONEBUTTONFUNCTION_DATETIME
Causes the phone to display current date and time; this information would be sent by the switch.

PHONEBUTTONFUNCTION_DIRECTORY
Calls up directory service from the switch.

PHONEBUTTONFUNCTION_COVER
Forwards all calls destined for this phone to another phone used for coverage.

PHONEBUTTONFUNCTION_CALLID
Requests display of caller ID on the phone's display.

PHONEBUTTONFUNCTION_LASTNUM
Redials last number dialed.

PHONEBUTTONFUNCTION_NIGHTSRV
Places the phone in the mode it is configured for during night hours.

PHONEBUTTONFUNCTION_SENDCALLS
Sends all calls to another phone used for coverage; same as PHONEBUTTONFUNCTION_COVER.

PHONEBUTTONFUNCTION_MSGINDICATOR
Controls the message indicator lamp.

PHONEBUTTONFUNCTION_REPDIAL
Repertory dialing—the number to be dialed is provided as a shorthand following pressing of this button.

PHONEBUTTONFUNCTION_SETREPDIAL
Programs the shorthand-to-phone number mappings accessible by means of repertory dialing (the "REPDIAL" button).

PHONEBUTTONFUNCTION_SYSTEMSPEED
The number to be dialed is provided as a shorthand following pressing of this button. The mappings for system speed dialing are configured inside the switch.

PHONEBUTTONFUNCTION_STATIONSPEED
The number to be dialed is provided as a shorthand following pressing of this button. The mappings for station speed dialing are specific to this station (phone).

PHONEBUTTONFUNCTION_CAMPON

Camps-on an extension that returns a busy indication. When the remote station returns to idle, the phone will be rung with a distinctive patterns. Picking up the local phone reinitiates the call.

PHONEBUTTONFUNCTION_SAVEREPEAT
When pressed while a call or call attempt is active, it will remember that call's number or command. When pressed while no call is active (such as during dial tone), it repeats the most saved command.

PHONEBUTTONFUNCTION_QUEUECALL
Queues a call to an outside number after it encounters a trunk-busy indication. When a trunk becomes later available, the phone will be rung with a distinctive pattern. Picking up the local phone reinitiates the call.

PHONEBUTTONFUNCTION_NONE
A "dummy" function assignment that indicates that the button does not have a function.

Values in the range 0x80000000 to 0xFFFFFFFF can be assigned for device-specific extensions; values in the range 0x00000000 to 0x7FFFFFFF are reserved.

The PHONEBUTTONFUNCTION_ constants have values commonly found on current telephone sets. These button functions can be used to invoke the corresponding function from the switch using **lineDevSpecificFeature**. Note that TAPI does not define the semantics of the button functions; it only provides access to the corresponding function. The behavior associated with each of the function values above is generic and may vary based on the telephony environment.

PHONEBUTTONMODE_ Constants

The PHONEBUTTONMODE_ bit-flag constants describe the button classes.

PHONEBUTTONMODE_DUMMY
: This value is used to describe a button/lamp position that has no corresponding button but has only a lamp.

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 twelve keypad buttons, 0 through 9, '*', and '#'.

PHONEBUTTONMODE_LOCAL
: The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY
: The button is a "soft" button associated with the phone's display. A phone set can have zero or more display buttons.

No extensibility. All 32 bits are reserved.

This enumeration type is used in the **PHONECAPS** data structure to describe the meaning associated with the phone's buttons.

PHONEBUTTONSTATE_ Constants

The PHONEBUTTONSTATE_ bit-flag constants describe the button's positions.

PHONEBUTTONSTATE_UP
: The button is in the "up" state.

PHONEBUTTONSTATE_DOWN
: The button is in the "down" state (pressed down).

PHONEBUTTONSTATE_UNKNOWN
: Indicates that the up or down state of the button is not known at this time, but may become known at a future time.

PHONEBUTTONSTATE_UNAVAIL
: Indicates that the up or down state of the button is not known to the service provider, and will not become known at a future time.

No extensibility. All 32 bits are reserved.

For backward compatibility, it is the responsibility of the service provider to examine the negotiated API version on the phone, and to not use these PHONEBUTTONSTATE_ values if not supported on the negotiated version.

PHONEERR_ Constants

This is the list of error codes that the implementation may return when invoking operations on phone devices. Consult the individual function descriptions to determine which of these error codes each function may return.

PHONEERR_ALLOCATED
: The specified resource is already allocated.

PHONEERR_BADDEVICEID
: The specified device ID is invalid or out of range.

PHONEERR_INCOMPATIBLEAPIVERSION
: The application requested an API version or version range that cannot be supported by the Telephony API implementation and/or corresponding service provider.

PHONEERR_INCOMPATIBLEEXTVERSION
: The application requested an extension version or version range that cannot supported by the service provider.

PHONEERR_INIFILECORRUPT
: Because of internal inconsistencies or formatting problems in the TELEPHON.INI file, it cannot be read and understood properly by TAPI.

PHONEERR_INUSE
: The device is currently in use. The device cannot be configured.

PHONEERR_INVALAPPHANDLE
: The application's specified usage handle or registration handle is invalid.

PHONEERR_INVALAPPNAME
: The specified application name is invalid. If an application name is specified by the application, it is assumed that the string does not contain any nondisplayable characters and is NULL-terminated.

PHONEERR_INVALBUTTONLAMPID
: The specified button/lamp ID is out of range or invalid.

PHONEERR_INVALBUTTONMODE
: The button mode parameter is invalid.

PHONEERR_INVALBUTTONSTATE
: The button states parameter is invalid.

PHONEERR_INVALDATAID
: The specified data ID is invalid.

PHONEERR_INVALDEVICECLASS
: The specified phone does not support the indicated device class.

PHONEERR_INVALHOOKSWITCHDEV
: The hookswitch device parameter is invalid.

PHONEERR_INVALHOOKSWITCHMODE
: The hookswitch mode parameter is invalid.

PHONEERR_INVALLAMPMODE
: The specified lamp mode parameter is invalid.

PHONEERR_INVALPARAM
: A parameter, such as a row or column value or a window handle, is invalid or out of range.

PHONEERR_INVALPHONEHANDLE
: The specified device handle is invalid.

PHONEERR_INVALPHONESTATE
: The phone device is not in a valid state for the requested operation.

PHONEERR_INVALPOINTER
: One or more of the specified pointer parameters are invalid.

PHONEERR_INVALPRIVILEGE

The *dwPrivilege* parameter is invalid.

PHONEERR_INVALRINGMODE
The ring mode parameter is invalid.

PHONEERR_NODEVICE
The specified device ID, which was previously valid, is no longer accepted because the associated device has been removed from the system since TAPI was last initialized.

PHONEERR_NODRIVER
The telephone service provider for the specified device found that one of its components is missing or corrupt in a way that was not detected at initialization time. The user should be advised to use the Telephony Control Panel to correct the problem.

PHONEERR_NOMEM
Insufficient memory to complete the requested operation, or unable to allocate or lock memory.

PHONEERR_NOTOWNER
The application does not have owner privilege to the specified phone device.

PHONEERR_OPERATIONFAILED
The operation failed for an unspecified reason.

PHONEERR_OPERATIONUNAVAIL
The operation is not available.

PHONEERR_REINIT
If TAPI reinitialization has been requested, for example as a result of adding or removing a telephony service provider, then **phoneInitialize**, **phoneInitializeEx** or **phoneOpen** requests are rejected with this error until the last application shuts down its usage of the API (using **phoneShutdown**), at which time the new configuration becomes effective and applications are once again permitted to call **phoneInitialize** or **phoneInitializeEx**.

PHONEERR_RESOURCEUNAVAIL
The operation cannot be completed because of resource overcommitment.

PHONEERR_STRUCTURETOOSMALL
The specified phone caps structure is too small.

PHONEERR_UNINITIALIZED
The operation was invoked before any application called **phoneInitialize**, **phoneInitializeEx**.

The values 0xC0000000 through 0xFFFFFFFF are available for device-specific extensions; the values 0x80000000 through 0xBFFFFFFF are reserved; and 0x00000000 through 0x7FFFFFFF are used as request IDs.

If an application gets an error return that it does not specifically handle (such as an error defined by a device-specific extension), it should treat the error as a PHONEERR_OPERATIONFAILED (for an unspecified reason).

PHONEFEATURE_ Constants

The PHONEFEATURE_ constants list the operations that can be invoked on a phone using this API. Each of the PHONEFEATURE_ values correspond to a TAPI function with an identical or similar name.

```
PHONEFEATURE_GETBUTTONINFO (0x00000001)
PHONEFEATURE_GETDATA (0x00000002)
PHONEFEATURE_GETDISPLAY (0x00000004)
PHONEFEATURE_GETGAINHANDSET (0x00000008)
PHONEFEATURE_GETGAINSPEAKER (0x00000010)
PHONEFEATURE_GETGAINHEADSET (0x00000020)
PHONEFEATURE_GETHOOKSWITCHHANDSET (0x00000040)
PHONEFEATURE_GETHOOKSWITCHSPEAKER (0x00000080)
PHONEFEATURE_GETHOOKSWITCHHEADSET (0x00000100)
PHONEFEATURE_GETLAMP (0x00000200)
PHONEFEATURE_GETRING (0x00000400)
PHONEFEATURE_GETVOLUMEHANDSET (0x00000800)
PHONEFEATURE_GETVOLUMESPEAKER (0x00001000)
PHONEFEATURE_GETVOLUMEHEADSET (0x00002000)
PHONEFEATURE_SETBUTTONINFO (0x00004000)
PHONEFEATURE_SETDATA (0x00008000)
PHONEFEATURE_SETDISPLAY (0x00010000)
PHONEFEATURE_SETGAINHANDSET (0x00020000)
PHONEFEATURE_SETGAINSPEAKER (0x00040000)
PHONEFEATURE_SETGAINHEADSET (0x00080000)
PHONEFEATURE_SETHOOKSWITCHHANDSET (0x00100000)
PHONEFEATURE_SETHOOKSWITCHSPEAKER (0x00200000)
PHONEFEATURE_SETHOOKSWITCHHEADSET (0x00400000)
PHONEFEATURE_SETLAMP (0x00800000)
PHONEFEATURE_SETRING (0x01000000)
PHONEFEATURE_SETVOLUMEHANDSET (0x02000000)
PHONEFEATURE_SETVOLUMESPEAKER (0x04000000)
PHONEFEATURE_SETVOLUMEHEADSET (0x08000000)
```

PHONEHOOKSWITCHDEV_ Constants

The PHONEHOOKSWITCHDEV_ bit-flag constants describe various audio I/O devices each with its own hookswitch controllable from the computer.

PHONEHOOKSWITCHDEV_HANDSET
: This is the ubiquitous, handheld, ear- and mouthpiece.

PHONEHOOKSWITCHDEV_SPEAKER
: This is a built-in loudspeaker and microphone. This could also be an externally connected adjunct speaker to the telephone set.

PHONEHOOKSWITCHDEV_HEADSET
: This is a headset connected to the phone set.

No extensibility. All 32 bits are reserved.

These constants are used in the **PHONECAPS** data structure to indicate the hookswitch device capabilities of a phone device. The **PHONESTATUS** structure reports the state of the phone's hookswitch devices. The function **phoneSetHookSwitch** and **phoneGetHookSwitch** use it as a parameter to select the phone's I/O device.

PHONEHOOKSWITCHMODE_ Constants

The PHONEHOOKSWITCHMODE_ bit-flag constants describe the microphone and speaker components of a hookswitch device.

PHONEHOOKSWITCHMODE_ONHOOK
: The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC
: The device's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER
: The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER
: The device's microphone and speaker are both active.

PHONEHOOKSWITCHMODE_UNKNOWN
: The device's hookswitch mode is currently unknown.

No extensibility. All 32 bits are reserved.

These constants are used to provide an individual level of control over the microphone and speaker components of a phone device.

PHONEINITIALIZEEXOPTION_ Constants

The PHONEINITIALIZEEXOPTION_ constants specify which event notification mechanism to use when initializing a session.

PHONEINITIALIZEEXOPTION_USEHIDDENWINDOW
: The application desires to use the Hidden Window event notification mechanism.

PHONEINITIALIZEEXOPTION_USEEVENT
: The application desires to use the Event Handle event notification mechanism.

PHONEINITIALIZEEXOPTION_USECOMPLETIONPORT
: The application desires to use the Completion Port event notification mechanism.

See **phoneInitializeEx** for further details on the operation of these options.

PHONELAMPMODE_ Constants

The PHONELAMPMODE_ bit-flag constants describe various ways in which a phone's lamp can be lit.

PHONELAMPMODE_DUMMY
: This value is used to describe a button/lamp position that has no corresponding lamp.

PHONELAMPMODE_OFF
: The lamp is off.

PHONELAMPMODE_STEADY
: Steady means the lamp is continuously lit.

PHONELAMPMODE_WINK
: Wink means normal rate on and off.

PHONELAMPMODE_FLASH
: Flash means slow on and off.

PHONELAMPMODE_FLUTTER
: Flutter means fast on and off.

PHONELAMPMODE_BROKENFLUTTER
: Broken flutter is the superposition of flash and flutter.

PHONELAMPMODE_UNKNOWN
: The lamp mode is currently unknown.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Where the exact on and off cadences may differ across phone sets from different vendors, mapping of actual lamp lighting patterns for most phones onto the values listed above should be straightforward.

PHONEPRIVILEGE_ Constants

The PHONEPRIVILEGE_ bit-flag constants describe the various ways in which a phone device can be opened.

PHONEPRIVILEGE_MONITOR
: An application that opens a phone device with the monitor privilege is informed about events and state changes occurring on the phone. The application cannot invoke any operations on the phone device that would change its state, so only status operations can be invoked. Multiple applications can monitor a phone device at any given time.

PHONEPRIVILEGE_OWNER
: An application that opens a phone device with the owner privilege is allowed to change the state of the lamps, ringer, display, hookswitch, and data blocks of the phone. Opening a phone device in owner mode also provides monitoring of the phone device. Only one application is allowed to be the owner of a phone device at any given time.

No extensibility. All 32 bits are reserved.

PHONESTATE_ Constants

The PHONESTATE_ bit-flag constants describe various status items for a phone device.

PHONESTATE_OTHER
: Phone-status items other than those listed below have changed. The application should check the current phone status to determine which items have 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 connecting the phone to the PC is plugged in with TAPI active.

PHONESTATE_DISCONNECTED
: The connection between the phone device and TAPI was just broken. This happens when the wire connecting the phone set to the PC is unplugged while TAPI is active.

PHONESTATE_OWNER
: The number of owners for the phone device.

PHONESTATE_MONITORS
: The number of monitors for the phone device.

PHONESTATE_DISPLAY
: The display of the phone has changed.

PHONESTATE_LAMP
: A lamp of the phone has changed.

PHONESTATE_RINGMODE
: The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME
: The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH
: The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME
: The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN
: The handset's microphone gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH
: The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME
: The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN
: The speakerphone's microphone gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH
: The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME
: The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN
: The headset's microphone gain setting has changed.

PHONESTATE_SUSPEND
: The application's use of the phone is temporarily suspended.

PHONESTATE_RESUME
: The application's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC
: The phone's device-specific information has changed.

PHONESTATE_REINIT
: Items have changed in the configuration of phone devices. To become aware of these changes (as for

the appearance of new phone devices), the application should reinitialize its use of TAPI.

PHONESTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **PHONECAPS** structure have changed. The application should use **phoneGetDevCaps** to read the updated structure. If a service provider sends a PHONE_STATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will receive PHONE_STATE messages specifying PHONESTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI to obtain the updated information.

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). A PHONE_STATE message with this value will normally be immediately followed by a PHONE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in PHONEERR_NODEVICE being returned to the application. If a service provider sends a PHONE_STATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 0x00010004 or above; applications negotiating a previous API version will not receive any notification.

No extensibility. All 32 bits are reserved.

PHONESTATUSFLAGS_ Constants

The PHONESTATUSFLAGS_ bit-flag constants describe a variety of phone device status information.

PHONESTATUSFLAGS_CONNECTED
: Specifies whether the phone is currently connected to TAPI. TRUE if connected, FALSE otherwise.

PHONESTATUSFLAGS_SUSPENDED
: Specifies whether TAPI's manipulation of the phone device is suspended. TRUE if suspended, FALSE otherwise. An application's use of a phone device may be temporarily suspended when the switch wants to manipulate the phone in a way that cannot tolerate interference from the application.

No extensibility. All 32 bits are reserved.

STRINGFORMAT_ Constants

The STRINGFORMAT_ enumeration constants describe different string formats.

STRINGFORMAT_ASCII
: Specifies standard ASCII character format using one byte per character.

STRINGFORMAT_DBCS
: Specifies standard DBCS character format using two bytes per character.

STRINGFORMAT_UNICODE
: Specifies standard Unicode character format using two bytes per character.

STRINGFORMAT_BINARY
: This is an array of unsigned characters; could be used for numeric values.

No extensibility. All 32 bits are reserved.

Assisted Telephony Constants

The following constants are used by Assisted Telephony.

TAPI Error Values

TAPIERR_INVALDESTADDRESS
: The pointer to the destination address is not valid, is NULL, or the destination address string is too long.

TAPIERR_INVALPOINTER
: The pointer does not reference a valid memory location. One or more of the pointers *lpszDestAddress*, *lpszAppName*, *lpszCalledParty*, or *lpszComment* have been specified but are invalid.

TAPIERR_NOREQUESTRECIPIENT
: No recipient application is available to handle the request. The user should start the recipient application and try again.

TAPIERR_REQUESTFAILED
: The request failed for unspecified reasons.

TAPIERR_REQUESTQUEUEFULL
: A recipient application is active, but the request queue is full or there is insufficient memory to expand the queue. The application can try again later.

Any other TAPIERR_ values are nonfunctional in Win32-based applications and must not be used.