Computer telephony integration

Pannekoek, D.R.

Award date:
1995

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Digital Systems Group

Master Thesis Report:

Computer Telephony Integration

D.R. Pannekoek

Coach : ir. M.J.M. van Weert
Supervisor : Prof. ir. M.P.J. Stevens
Period : Sept. 1994 - June 1995

Abstract

One of the developments in telecommunication systems is the integration of computers and telephony. This results in new possibilities that can be used to improve existing telephony services or to create entirely new services. A number of possible applications of this so-called Computer Telephony Integration (CTI) in the fields of call centres, office productivity and multi-media communication is discussed.

CTI implies that the computer is able to control calls and obtain information from the telephony network. This requires a communication path between the computer and a switch within the telephony network. This communication can be based on either a first-party or a third-party call control model. First party call control is typical in systems where the only interface between computer and switch is a telephone line (i.e. the call control communication consists in signalling). Third-party call control is more versatile, but requires a separate CTI link between the computer and the PBX. Configurations, interfaces and protocols for both kinds of call control are discussed.

CTI software systems can provide applications with uniform access to the call control functionality by means of a standard Application Programming Interface (API). Two such systems are discussed: Microsoft/Intel's Windows Telephony system (also known as TAPI) and Novell/AT&T's NetWare Telephony Services (also known as TSAPI). The Windows Telephony system is an open standard for first-party call control purposes. NetWare Telephony Services provides third-party call control in a NetWare LAN environment.

A CTI demo system has been developed that combines the Windows Telephony system with a third-party call control configuration like the NetWare Telephony Services configuration. Unlike the NetWare Telephony Services configuration, no LAN is required. This implies that the configuration could also be used in Centrex situations. Furthermore, remote access is possible, opening the door for teleworking.

The developed demo software consists of a demo telephony application, which implements the concepts of screen based telephony and multi-media call control, and a Telephony Service Provider that implements X.25 protocols, the ECMA CSTA call control protocol (specified in ASN.1) and a mapping between TAPI and the ECMA CSTA protocol. The entire demo system has been built and successfully demonstrated in the demo room of Ericsson Telecommunicatie b.v. in Rijen.

Contents

1. Introduction

Computer Telephony Integration (CTI) has become a popular topic in the telecommunication business. The term CTI stands for integration of the computer and telephony environments. This implies that the computer is offered a certain amount of control over the telephone network. In this manner, simple telephony-related activities can be automated. The computer can also offer entirely new telephony services.

Interest in CTI has only recently really started, with the introduction of CTI extensions for the two popular operating environments Microsoft Windows and Novell NetWare. However, the concept of CTI is not new; it is already being employed for some years in so-called call centres. An example of a call centre is a helpdesk where a group of agents provide telephonic support.

CTI can offer great benefits to companies that rely heavily on telecommunication. Dutch market research agency Ovum estimates the revenues currently earned by companies in Europe and the United States from employing CTI to be around 400 million US dollars per year. This is mainly as a result of applications in call centres. Ovum predicts that the revenues from CTI in Europe and the US will eventually rise to 6 billion US dollars per year in the year 2000, where only half of this amount will be from applications in call centres. The other half will be from office productivity and personal productivity applications.

The most useful applications of CTI are business-oriented, therefore the focus of this report will be on business applications. Besides the already mentioned call centre applications, a number of other useful applications will be discussed (chapter 2). Based on knowledge of existing CTI configurations, a new configuration is conceived in chapter 3. In order to get insight in the issues involved in implementing this new configuration, a demo CTI system (with limited functionality) has been developed. The components of this demo system are described in chapter 4. The results of this development are discussed in chapter 5.

2. CTI applications

In this chapter, a number of CTI applications is discussed. For the purpose of this discussion, three CTI application-fields are identified: call centres, office productivity and multi-media communication. The discussed applications are merely intended as examples; all kinds of variations may be possible. Also, the intention has not been to present an exhaustive list of applications; numerous other applications are possible.

2.1 Call centres

Call centres consist of groups of people called agents. Simply put, all agents within a group are assigned the same job: either answering incoming calls (inbound call centres) or making calls (outbound call centres). Inbound call centres are used for helpdesks, customer support, info services, etcetera. Outbound call centres are mainly used for telemarketing and less frequently for debt collection purposes.

2.1.1 Inbound call centres

The technique used in inbound call centres is called Automatic Call Distribution (ACD). A group of agents is assigned a single phone number: the ACD group number. Incoming calls to this ACD group number are automatically distributed to any ACD agent that is available at that moment.

A standard feature found in almost any PBX is the *hunt-group*, where a group of extension lines is assigned a single phone number. When an incoming call arrives for the hunt-group, the PBX searches ("hunts") the first extension line in the group that is free. Although it may seem at first that this standard feature is sufficient to create an ACD group, this is not the case. Whether an agent is available is not simply a matter of his phone being free. There are actually three occasions when the agent must be considered unavailable: when the agent is involved in a call, when the agent is involved with (paper-) work that results from a call and when the agent is absent. Only in the first case will the agent's extension line be busy.

In order to be able to locate an available agent, additional functionality is required to keep track of the state of each agent. Some kind of queuing mechanism is also required: if there is no agent available, the caller should be placed in a queue, usually resulting in an announcement that all agents are currently busy and a request to the caller to wait a while. There are a number of systems that can offer the required ACD functionality:

- A stand-alone ACD switch. This is basically a dedicated PBX, specialised for ACD. In general, stand-alone ACD switches are used in large call centres.
- A *Centrex*[1] switch. Many Centrex switches have built-in ACD functionality.
- A PBX with built-in ACD functionality. Some PBXs can be enhanced with ACD packages.
- A computer system that is connected to a PBX through a so-called *CTI link*. In this case, the computer provides the ACD functionality; the PBX needn't have additional ACD functionality.

In the latter case, the PBX sends a routing request to the computer system (over the CTI link) as soon as a call for the ACD group number arrives. The computer selects an available agent and returns the appropriate extension number to the PBX. Figure 1 depicts such a configuration.

---

[1] *Centrex is a service offered by Public Telecom Operators. Simply put, companies can rent a part of a public switch and use it as PBX. Centrex switches offer all the functionality of normal PBXs.*

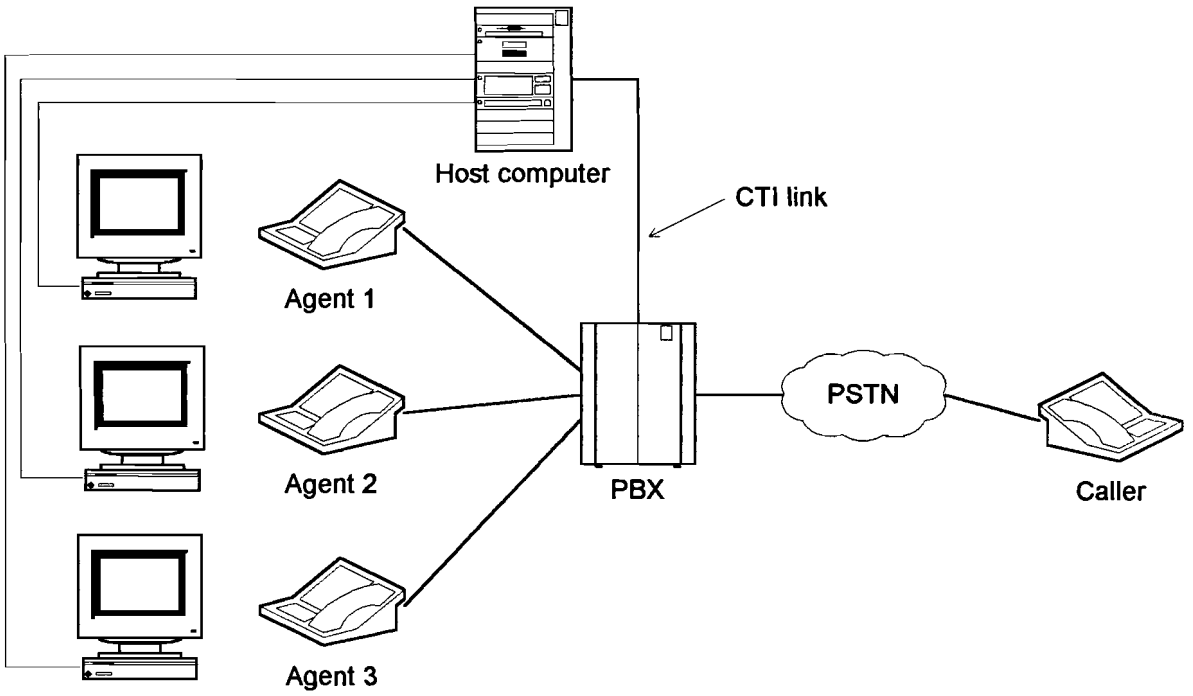


Figure 1. Call centre configuration using CTI

Each agent has a computer terminal that is connected to the host computer. The agent uses the terminal to report his current state to the host computer, which keeps track of the state of all agents.

With this configuration more sophisticated features can be implemented than with the other systems[2] mentioned earlier. For example, the host computer can automatically retrieve information concerning the caller[3] from a database and display this information on the agent's terminal. In multi-lingual call centres where agents are specialised in specific languages, the host computer can select an appropriate agent based on the caller's origin. More sophisticated queuing mechanisms can also be implemented in the host computer. For example, the host computer can already collect some information from the caller using Interactive Voice Response techniques, while all agents are occupied. As soon as an agent becomes available, this information is passed to that agent's terminal, thus shortening the time the agent needs to spend on the caller.

Over the past five years, several large computer manufacturers (DEC, IBM, HP, Tandem) have co-operated with switch manufacturers to develop CTI systems as shown in Figure 1. Recently, Novell and AT&T have jointly developed a system, called *NetWare Telephony Services*, that allows this kind of CTI applications to be run over a NetWare LAN. In this case, the agents use a NetWare client PC "connected" to a NetWare telephony server, instead of a terminal connected to a host computer.

2.1.2 Outbound call centres

In outbound call centres, all agents share a list of persons who need to be called. Agents usually spend a lot of time in trying to reach those persons in the first place. A lot of administration is necessary to keep track of which persons couldn't be reached immediately and when to call them back at what phone

[2] *Stand-alone ACDs and PBXs with built-in ACD functionality may also have a CTI link to enhance their features.*

[3] *Of course, the host computer first has to determine the identity of the caller. If no Calling Line Identification is available, the computer will have to use Interactive Voice Response techniques.*

number. Efficiency of the agent's work in an outbound call centre can be improved by automating the initial phase of the agent's task.

Using a CTI configuration like in Figure 1, the host computer can maintain the list of person to be called and instruct the PBX to call the phone-number at the top of the list. As soon as the phone is answered, a connection is established with any agent that is available at that moment (this process is also known as *predictive dialling*). Information concerning the called person is then sent to the agent's terminal. If the phone is not answered, the call will automatically be rescheduled to a later time. Even a larger part of the agent's task can be automated if the host computer employs Interactive Voice Response techniques to make sure the person on the other side of the line is the required person.

2.2 Office productivity

According to the Dutch economic institute NEI, 15 percent of all persons that call a Dutch company don't get to speak the person they wanted (4 percent can't reach the company at all). NEI has calculated a nation-wide production loss of 500 million Dutch guilders per year from calls that don't get answered within a reasonable time [20]. In other words: a lot of money can be saved by increasing the telephonic availability of employees.

2.2.1 Screen based telephony

Most PBXs contain a large number of call control functions that can increase telephonic availability: call forwarding, call back when free, transfer call, conference call, pickup call, etcetera. The problem is that the number of features is usually so large and the way they are activated is so complex that nobody can remember how to activate them. CTI can be of help, since computers with Graphical User Interfaces can offer a much easier user interface that standard phone devices.

Telephony operations can also be integrated in the computer's operating environment. A database application can offer a dial function to automatically dial the phone number of a person in a phone directory database. When an incoming call arrives, the name of the caller can be looked up in the phone directory database and presented to the user. Setting up a conference call may be realised by dragging icons (representing persons) into a *conference area* on the screen.

2.2.2 Voice mail and automated attendant

Traditionally, when somebody calls an employee that is absent, he will be connected to a secretary or attendant or alternatively to an answering machine. Most of the callers will simply leave a message containing the caller's name, his phone-number and a request to call him back. Since answering these calls is a simple, yet time-consuming business, it is desirable to automate this process. This can be done by means of CTI.

If a computer answers the phone during the absence of an employee, Interactive Voice Response techniques can be employed to allow the caller to choose whether he wants to be called back (if Calling Line Identification can be used, the caller doesn't even have to give his phone-number), leave a message in a voice mail box (the computer based equivalent of a conventional answering machine) or speak to a secretary. When the employee returns, the computer can present a list of persons that called during his absence. The computer can then assist in calling back the persons that requested that. This kind of application is called automated attendant.

2.2.3 Fax server

Although fax has become a heavily used medium for telecommunication in companies, distribution of incoming fax messages is usually still very primitive. Most organisations have one or more centralised fax machines that are used for both sending and receiving fax messages. Received fax messages are usually distributed manually (as internal mail), using the name of the addressee on the fax cover page.

This process is expensive and relatively slow; the introduction of *fax-on-demand*[4] services has even made it less adequate, since fax-on-demand systems usually don't generate a cover page with a name of the addressee. The entire process can be automated by means of CTI, as shown in Figure 2.

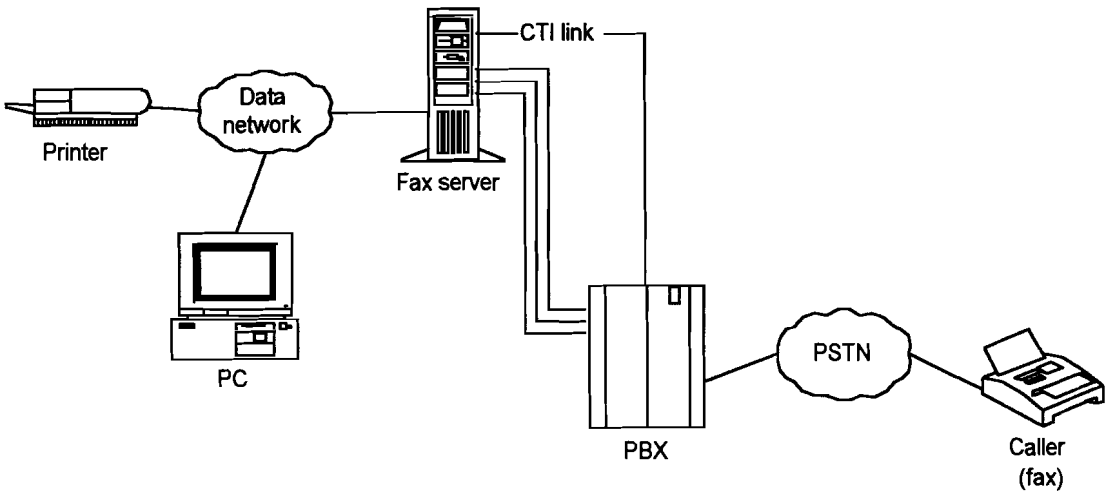


Figure 2. Fax server configuration

Incoming fax calls are routed to a computer, called fax server, equipped with one or more fax cards. This fax server receives and stores all fax messages. Employees can retrieve their fax messages on their PC via the data network. For optimal integration, a so-called *unified messaging application* can offer integrated access to stored fax, e-mail and voice mail messages. The employee can read the message on his PC or print it on either a local printer or the nearest network printer.

To enable the fax server to determine the identity of the addressee, each employee need to be assigned a personal fax number. The PBX must route calls for all personal fax numbers to the fax server and inform the fax server of the called number. For this information exchange between PBX and computer, a so-called CTI link can be used.

2.3 Multi-media communication

Multi-media communication means that a call involves more than one medium (e.g. sound, video, computer data). With multi-media communication, two topics are involved: multi-media transmission [18] and multi-media call control [3]. CTI can be applied to implement multi-media call control in current network configurations. The issues in multi-media call control are related to the issues in multi-media transmission, therefore the latter will be discussed first.

---

[4] *Fax-on-demand systems are Interactive Voice Response (IVR) systems that allow callers to retrieve information on their fax machine. During the IVR dialogue, the caller is asked to enter the phone-number of his fax machine. Afterwards, the IVR system calls back on the fax number and sends the requested information.*

2.3.1 Multi-media transmission

Each medium has its own characteristics and imposes its own demands on transmission capabilities of the network. The characteristics of three common media are discussed: voice, video and (computer) data.

- Using *Pulse Code Modulation*[5] (PCM), human voice can be transmitted digitally. A bandwidth of 64 kbps is required for *toll quality* voice reproduction. PCM voice is said to be *isochronous*, meaning that the network delay needs to be constant (within certain limits) in order to be able to reproduce the signal undistorted.
- Motion pictures (video) can also be transmitted digitally using PCM. This requires much more bandwidth, however. Depending on the quality, the frame rate (the number of pictures per second) and the compression methods used, the bandwidth may vary from hundreds of kilobits per second to tens of megabits per second. PCM video is isochronous, like PCM voice.
- Digital data generated by computers is typically *bursty*, meaning that the required bandwidth varies drastically. Depending on what is being transmitted (nothing, text, binary files, graphic pictures) the required bandwidth may vary from zero to several megabits per second. Transmission of a full-screen picture (SVGA 800x600, 256 colour) within one second, for example, requires a bandwidth of 4 Mbps. Transmission of a full screen of text in the same time only requires 32 kbps. Unlike PCM voice or video, a constant network delay is not required; computer data is said to be *asynchronous*.

Over the past years, two types of digital communication networks have evolved:

1. Telephony/voice networks. Designed to carry PCM voice, these networks are based on circuit-switching techniques, resulting in a fairly constant network delay. Circuits (channels) generally offer a bandwidth of 64 kbps. Transmitting PCM video therefore requires multiple channels in parallel. Although transmission of computer data is possible, this leads to waste of network resources, because of its "bursty" nature.
2. Computer/data networks. Designed to carry computer data, these networks typically use packet-switching techniques. Packet-switching yields optimal use of network resources in case of "bursty" traffic. However, these networks can't guarantee the constant network delay necessary to carry PCM voice or video traffic.

A network configuration for multi-media transmission can be created by employing two networks: a telephony/voice network for isochronous traffic and a computer/data network for asynchronous traffic. In order to allow for multi-media transmission over a single network, several new network techniques have emerged:

- *ISDN* can be considered a hybrid network technique: both circuit-switched channels (*Circuit Mode Bearer Services* on the B-channels) and packet-switched channels (*Packet Mode Bearer Services* on the B- or D-channel) are available. An ISDN *Basic Rate Interface* (BRI) offers two 64 kbps B-channels and one 16 kbps D-channel. If both B-channels are used in parallel in circuit-mode, this yields an aggregate bandwidth of 128 kbps for isochronous traffic and 16 kbps for asynchronous traffic. If one B-channel is used in circuit-mode and the other in packet-mode, 64 kbps is available for both types of traffic. This is hardly sufficient for multi-media communication involving video and/or computer image transfer; more appropriate would be an ISDN *Primary Rate Interface* (PRI), offering 30 B-channels (an aggregate bandwidth of approximately 2 Mbps) for isochronous or asynchronous traffic and one 64 kbps D-channel for asynchronous traffic.

[5] *A number of variations on the basic PCM method are possible (e.g. delta-modulation and ADPCM). For clarity, only the term PCM will be used in the discussion.*

- *isoEthernet* (described in the IEEE 802.9 standard) is also a hybrid network technique. An isoEthernet LAN combines a IEEE 802.3 LAN (with a bandwidth of 10 Mbps) for asynchronous traffic with 96 circuit-switched channels of 64 kbps (aggregate bandwidth of 6 Mbps) for isochronous traffic.
- *FDDI-II* LAN and *FDDI-Follow-On-LAN* (FFOL) are also hybrid networks. As with isoEthernet, the total bandwidth is divided in two logical parts. One is used for multiple 64 kbps circuit-switched channels, the other for asynchronous traffic. The total bandwidth is 100 Mbps for FDDI-II and 150 Mbps to 2.4 Gbps for FFOL.
- *ATM* is a true multi-media networks technique that will be employed in both LANs and public networks (WANs). ATM networks use packet-switching[6] techniques, but can guarantee sufficiently constant network delay to allow for transmission of isochronous traffic as well as asynchronous traffic. An ATM User Network Interface (UNI) offers a bandwidth of 155 Mbps, enough for high quality video.

2.3.2 Multi-media call control

The concept of multi-media call control is also known as *separation of call and connection*. In a multi-media call each medium will require a separate connection, but these connections should be handled as a unity (a single multi-media call). For example: when a multi-media call consisting of a speech connection, a video connection and a data connection is dropped, all three connections should be dropped automatically. Future broadband networks shall incorporate this concept in their signalling system. However, also current network configurations can be enhanced with this concept by means of CTI.

Figure 3 shows an abstract view on a multi-media communication network configuration. Two stations that want to communicate both consist of three entities: a data entity that transmits and receives asynchronous data, a voice entity that transmits and receives (isochronous) PCM voice, and a video entity that transmits and receives (isochronous) PCM video. Transport of asynchronous and isochronous traffic takes place over two functionally separate networks. Examples of such a network configuration are a combination of a LAN (for asynchronous traffic) and an ISDN (for isochronous traffic), a single ISDN where both packet-mode and circuit-mode connections are used and an isoEthernet network.

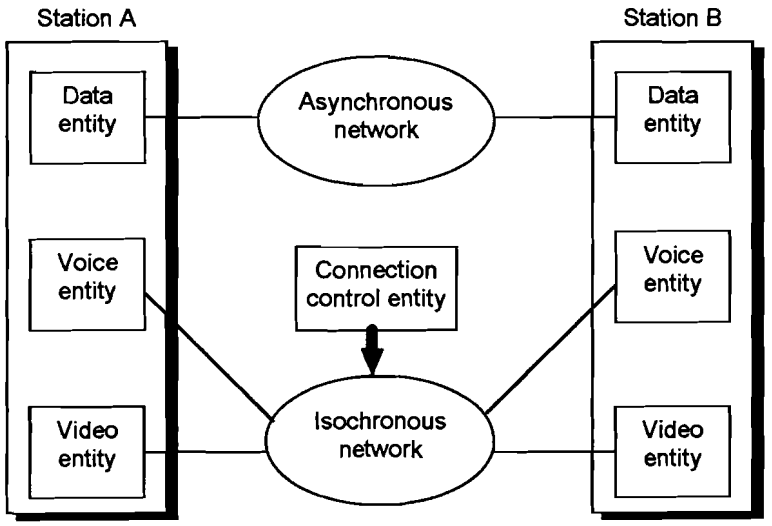


Figure 3. Multi-media transmission (abstract view)

[6] *The packet switching technique used in ATM is called cell-relay and is optimised for high speed, low-delay switching. This is achieved by using relatively short, fixed length packets (cells) and limited error-checking within the network.*

The isochronous network contains a connection control entity that is responsible for the routing of the data through the network. To initiate communication with another party, the calling party uses signalling[7] to request the connection control entity to establish a connection between the calling party itself and the called party. During connection establishment, the connection control entity informs the called party of an incoming call (by means of signalling).

In relation to multi-media call control, it should be noted that the connection control entity controls connections, not calls. In order to initiate a multi-media call consisting of multiple connections, multiple connection set-up requests need to be send to the connection control entity. The connection control entity does not regard these connections to be related in any way, leaving the responsibility of handling the connections as a unity to the user. By means of CTI, the handling of the connections that are a part of a multi-media call can be performed by a computer.

Figure 4 depicts an enhanced version of the configuration shown in Figure 3. The CTI link in Figure 4 is the part that enables the multi-media call control concept to be implemented. This link allows the call control entities located in the stations not only to control the (logical) connections in the asynchronous network, but also the connections in the isochronous network. If one of the stations from Figure 4 wants to initiate a multi-media call involving all three media, its call control entity (in a computer) can establish a connection between its data entity an the data entity of the called party. It can also requests the connection control entity of the isochronous data network to establish a connection between the two voice entities and one or more connections between the video entities. Since this entire process is performed automatically, the user only needs to be involved in the control of the multi-media call as a whole; he may be ignorant to the fact that the call actually consists of multiple connections, possibly over multiple networks.

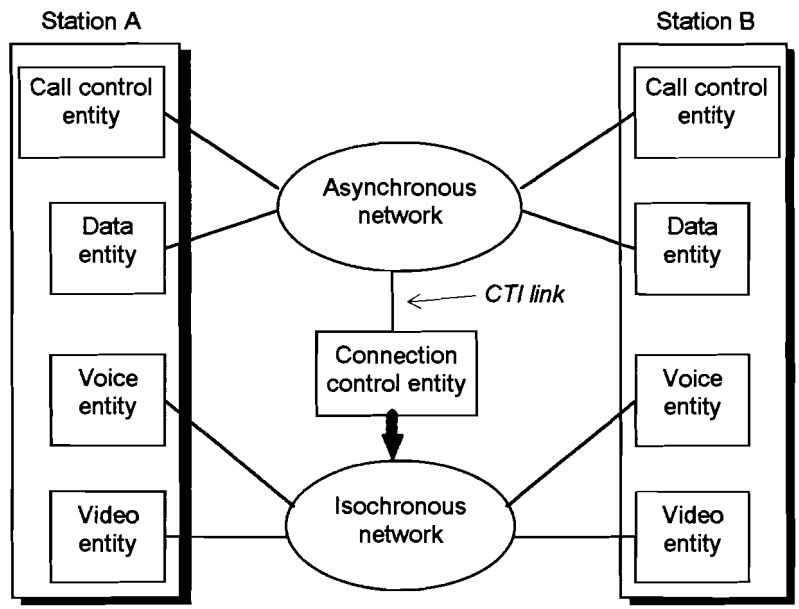


Figure 4. Multi-media call control (abstract view)

[7] *The signalling path is not shown in Figure 3.*

3. CTI configurations

A very important aspect in CTI configurations is *call control*. Call control is a process that takes place in the switches in the telephony network. The call control entity inside a switch controls the connections inside the switch and takes care of signalling towards stations and other switches in the network. Basic call control operations are originating, answering and dropping calls at a station. More advanced call control features typically offered by PBXs are call forwarding, call transfer, conferencing, etcetera. Information collected by the switch concerning calls (e.g. Calling Line Identification) and stations is also regarded as call control information.

In order to provide a computer with call control capabilities, a communication path between the computer and the call control entity inside a switch is required for the exchange of call control information. This call control communication can be based on either a first-party or a third-party call control model.

- *First-party call control* is the conventional call control model, implemented in the signalling systems used for user-network interfaces. Call control operations performed by a first party always have respect to that party itself. For example, a first party may initiate a call (from its own station) to another station, a first party may answer an incoming call (at his own station), a first party may activate a switch feature (at his own station), etc.
- *Third-party call control* is a more versatile model, where a third party can manipulate calls between two (or more) arbitrary parties without being involved in those calls. A computer can be offered third-party call control capabilities by means of a special interface to the switch's call control entity.

3.1 First-party call control

Figure 5 depicts three alternative configurations for computer supported first-party call control, where the computer is able to control voice calls at a telephone device. Configuration (a) is the most basic configuration, with a modem (with auto-dialling capabilities) and a phone-set connected in parallel to a single analogue phone-line. By sending commands[8] to the modem, the computer can perform limited call control: answering inbound calls and initiating outbound calls. When a connection is established, it may be used for voice communication (using the phone-set) and for data communication (using the modem). Simultaneous voice and data communication is not possible with conventional modems. However, new modem technology does allow simultaneous voice and data communication over a single connection; with these modems, the data transfer rate will typically drop (e.g. from 14400 baud to 4800 baud) when a voice signal is present.

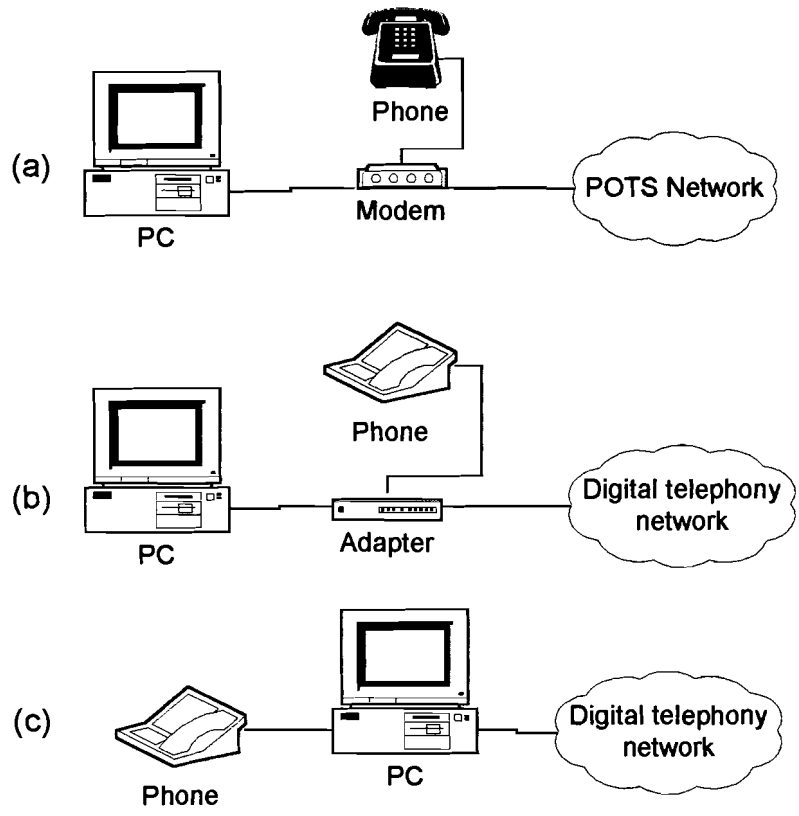


Figure 5. First-party call control configurations

Many modern PBX systems use digital extension lines. These systems often allow simultaneous transmission of PCM voice and computer data over a single line. As shown in Figure 5b, an adapter unit is used for protocol conversion[9] and (de-) multiplexing of PCM voice and computer data. The adapter unit may allow the computer to control both voice and data calls. Some adapter units (e.g. Ericsson's *Terminal Adapter Units*) accept standard Hayes AT and/or V.25bis commands, others (e.g. AT&T's *General Purpose Communication Interface*) require that the computer knows the proprietary signalling protocols used by the PBX.

In configuration (c), the computer is connected directly to the telephony network by means of a telephony board inside the computer. The phone-set is not connected to the telephony network, but to this telephony board instead. The phone-set is actually not much more that a speech input/output device of the computer. The computer may allow a user to treat the phone-set as if it were a regular phone device: as soon as the hand-set is taken off-hook, the computer detects that the hook-switch is released and generates a dial-tone. The digits dialled on the phone-set are collected by the computer and as soon as the entire number has been dialled, the computer makes a call to that number. When a connection has

[8] *The Hayes AT command set has become a de facto standard for modem control. The ITU-T also has defined a standard command set, described in recommendation V.25bis.*

[9] *The interface towards the computer is generally a standard RS-232 interface; the interface towards the PBX is usually a proprietary interface, optimised for high-speed transmission over long distances.*

been established, the computer will send speech input from the phone-set to the telephony network and vice versa.

Because the computer is connected directly to the telephony system and communicates directly with the switch, configuration (c) can offer more complex call control than the other configurations. The computer can actually perform all the operations that can be performed from a sophisticated digital phone-set; this is why this configuration is called *digital phone emulation*. The problem is that most PBX systems use proprietary line interfaces and signalling protocols, therefore each system needs its own specific hard- and/or software inside the computer to enable communication between the computer and the PBX. Figure 6 depicts the architecture of a digital phone emulation board where this problem has been minimised by locating all system-specific logic in one custom IC (ASIC) that can easily be replaced.

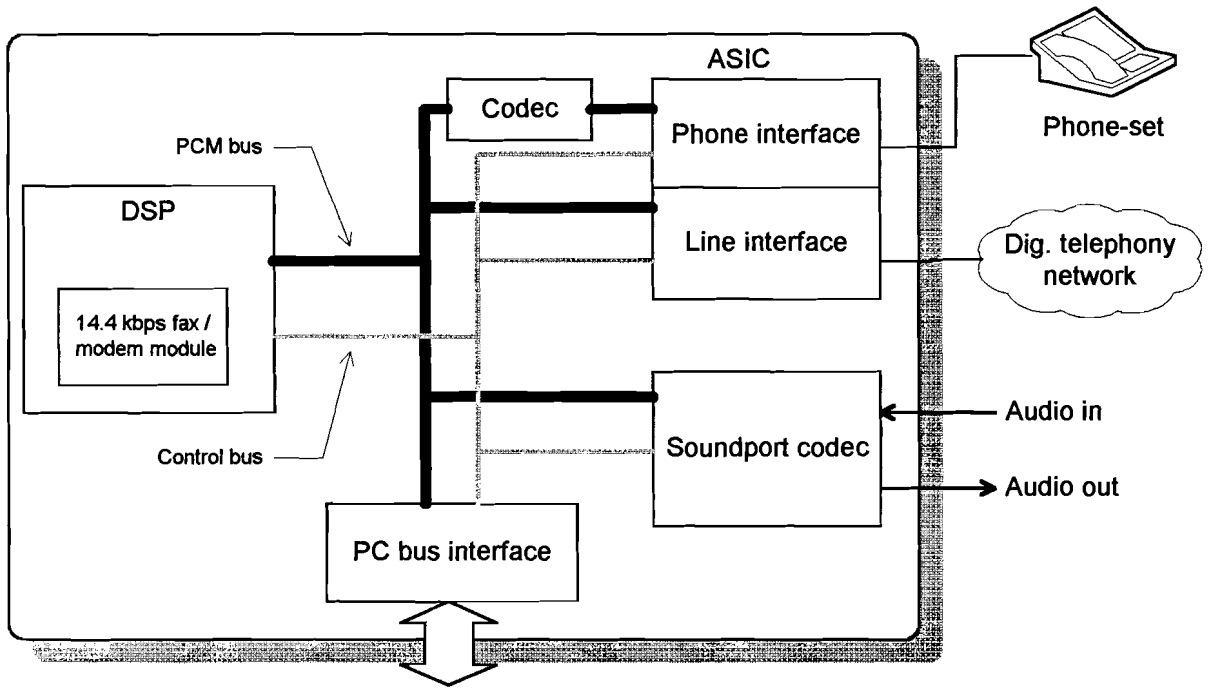


Figure 6. VTG's Scorpion digital phone emulation board architecture [23]

Introduction of standard ISDN extension lines in PBX systems can solve the problem. Interfaces and signalling protocols for public ISDN systems have been defined in international standards:

- ISDN physical layer: ITU-T recommendation I.430 (BRI) and I.431 (PRI)
- ISDN Digital Subscriber Signalling system #1: ITU-T rec. Q.921 (LAPD) and Q.931 to Q.933

For Euro-ISDN, slightly different signalling protocols (called *EDSS1*) have been standardised by ETSI. Interfaces and signalling protocols in private ISDN systems will be largely the same as in public ISDN systems. Signalling systems for private ISDNs have been defined by ECMA: SSIG for subscriber signalling and QSIG for inter-PBX signalling [22].

The use of standard ISDN interfaces and signalling protocols does not only eliminate the need for PBX-specific hard- and/or software, it also enables sophisticated computer supported call control in Centrex systems and public ISDN systems.

3.1.1 Windows Telephony system

Although all three configurations discussed in the previous section offer the same basic call control functionality, they all work with different hardware (let alone the differences between manufacturers) and therefore with different software to interface with the hardware (drivers). Unless some standardisation takes place, this means that each telephony application must include its own drivers and configuration procedures for all sorts of hardware devices, usually resulting in only the most popular devices being supported. This has been a motivation for Intel to conceive a standard telephony interface for applications running under Microsoft Windows. The idea was adopted by Microsoft and jointly they developed the *Windows Telephony* system, also known as *Telephony API* or *TAPI*.

Although the term Telephony API (TAPI) is often used instead of Windows Telephony system, the use of this term as such may cause some confusion. As is shown in Figure 7, the Windows Telephony system is more than the specification of an *Application Programming Interface* (API) alone. The Windows Telephony system is designed according to the *Windows Open Services Architecture* (WOSA) concept, and as such consists of an API used by applications and an *Service Provider Interface* (SPI) that is implemented by *Service Provider* programmers. These Service Providers are a sort of high-level device drivers and may use regular (low-level) device drivers to perform their tasks.

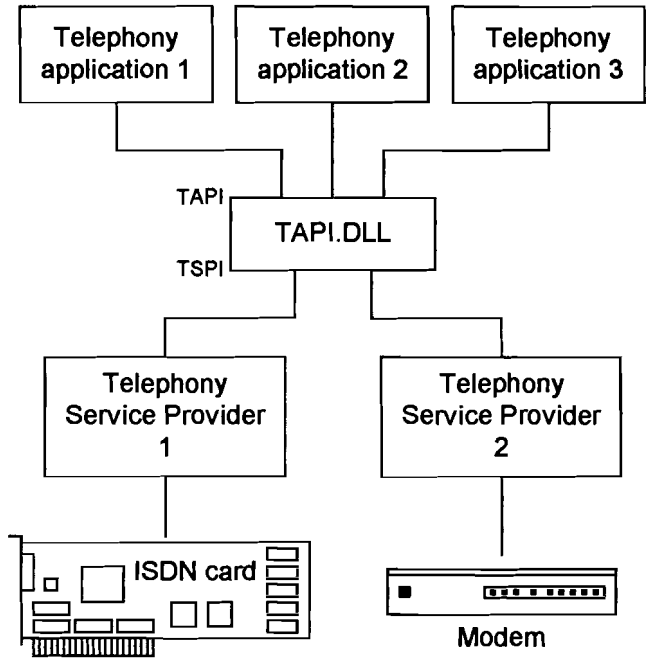


Figure 7. Windows Telephony architecture

An important advantage of the double interface (API/SPI) concept is the (de-) multiplexing performed by a *Dynamic Link Library* (TAPI.DLL in Figure 7) that is part of the system and is located between these two interfaces. There may be a number of applications on one side that communicate with a number of Service Providers on the other side through the API/SPI interface pair. Since all communication goes through the system DLL that performs the (de-) multiplexing of the information streams, applications and Service Providers don't have to be aware of the fact that there are multiple Service Providers or applications, respectively. As far as they are concerned, they are communicating with only one other party, namely the system DLL.

Another duty of the system DLL is mapping the API to the SPI and vice versa. Since these two interfaces are usually largely the same, this isn't such a difficult task. Part of this mapping is a basic validity check of parameters passed by the applications, taking this load off the Service Providers. Another mapping activity that is specific to TAPI.DLL is address translation. The Windows Telephony system recognises two address formats:

1. *The canonical address format.* This format is ideal for use in electronic address books. Country-code, area-code, subscriber-number, optionally (ISDN) sub-address and name are separately specified. By comparing these separate parts with configured current location info, the system can determine whether a call is local, long-distance or international. Parts of the number may be left out in the dialling process (e.g. the country-code and area-code if the call is local) and specific prefixes can be used for the different types of calls.

2. *The dialable address format.* This format can be used directly to dial a number on a device that supports the Hayes AT dialling command. The format contains all the digits to be dialled and additional formatting characters according to the Hayes standard (e.g. P for pulse dialling, T for tone dialling, W to wait for dial-tone).

Obviously, the canonical address format is most suited for the interface towards the applications (TAPI), while the dialable address is most suited for the interface towards the Service Providers (TSPI). As a matter of fact, the TAPI supports both formats, while the TSPI only uses the dialable address format. In case an application wants to use the canonical address format, TAPI.DLL can perform the translation to the dialable address format.

The Windows Telephony system is intended for call control[10] purposes only, it does not give access to information exchanged over a call. To manage this *media stream*, developers need to integrate functions from other APIs into telephony applications. In this manner, TAPI works in conjunctions with other Windows-based services such as Windows multi-media wave audio, Media Control Interface (MCI) or data-communication APIs. Although the Windows Telephony system is not involved in the media stream itself, it does support *media detection*. This implies that telephony applications can state the kind of media stream (voice, fax, data, etc.), called media mode, they are "interested in". When an inbound call arrives, the concerned Service Provider will try to determine the media mode of the call and then inform TAPI.DLL. Only the telephony applications that have stated interest in the concerned media mode will be informed of the call.

The number of products with Windows Telephony support (i.e. with a Telephony Service Provider) is negligible at the moment. However, a large number of companies have indicated that they are developing, or intend to develop, Telephony Service Providers for their products. Typical products concerned are:

- Terminal adapters (refer to section 3.1).
- Modems that allow simultaneous voice/data communication.
- Voice processing boards.
- Multi-purpose DSP boards (fax / modem / voice processing).
- ISDN boards.
- Digital phone emulation boards (refer to section 3.1)
- Client/server voice processing systems. A voice server on the LAN is equipped one or more voice processing boards. The clients on the LAN can use voice processing facilities on the voice server.

It should be noted that the Windows Telephony specification is an open standard: it does not impose the use of specific configurations or protocols to perform the call control operations specified. As such, it does not necessarily have to be restricted to first-party call control configurations, although it is based on a first-party call control model.

[10] *The call control capabilities of the Windows Telephony system are very extensive and are designed to take full advantage of POTS, ISDN and PBX systems.*

3.2 Third-party call control

With conventional signalling techniques, a computer can communicate with the call control entity inside a switch. This communication is based on a first-party call control for historical reasons (without CTI, third-party call control is of little use). To provide a computer with third-party call control capabilities, the signalling protocols could be enhanced, but this is not what most PBX manufacturers did. A so-called *CTI link* was preferred. This is a dedicated link that interfaces directly to the PBX's system control unit.

It should be noted that applications running on a computer that interfaces directly with the PBX's system control could also be implemented as software in the PBX itself. In fact, this is what PBX manufacturers have been doing all the time. Over the years, PBX software has become so complex that manageability is becoming a problem. Separation of core PBX functionality (in the PBX) and additional functionality (in a computer) is a solution to this software management problem:

- Thoroughly tested PBX core software doesn't have to be updated (with the risk of introducing new bugs) each time functionality is added to the system.
- PBX software needs to be upgraded less frequently.
- Common functionality can be made available as standard applications that can be used across different PBX systems.

This concept is similar to the *Intelligent Network* (IN) concept found in public networks [10]. Unlike the IN concept, third-party call control CTI configurations can also offer an alternative (computer based) communication path between the user and the network, enabling the exchange of complex information (text or graphics) in an easy manner.

3.2.1 The CTI link

Early CTI links were used for connection to a host computer and are based on proprietary call control protocols. Because of this non-standard nature of the CTI links, third-party call control systems like DEC's CIT, IBM's CallPath, HP's ACT and Tandem's CAM are typically tied to a small number of supported PBXs. Only recently, standardisation of third-party call control protocols has taken place.

The most prominent standard at the moment is the *Computer Supported Telecommunications Applications* (CSTA) standard from the *European Computer Manufacturers Association* (ECMA). Parallel standardisation work of ANSI, called *SCAI*, has got little attention. The broad support for the ECMA CSTA standard is visible from the number of companies involved in the development of the standard (see Table 1) [8]. A large number of PBX manufacturers have already implemented the standard or have committed themselves to implement it in the near future. Some PBX manufacturers that already had a CTI link based on proprietary call control protocols before the ECMA CSTA standard was intro-

Table 1. ECMA CSTA standardisation group

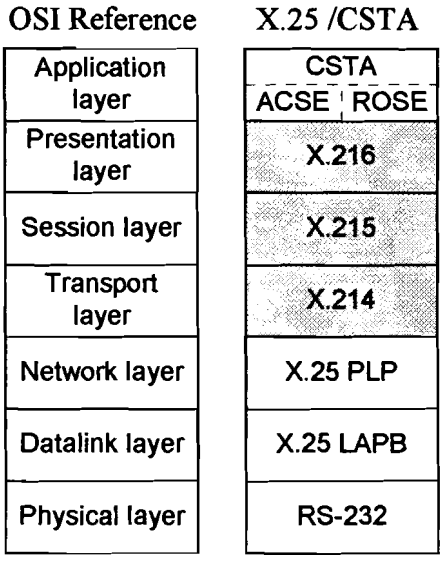
Computer manufacturers	Bull DEC HP IBM Siemens-Nixdorf
Switch manufacturers	Alcatel AT&T BT Ericsson GPT Mitel Northern Telecom Siemens Telenorma
Others	Dutch PTT Swedish PTT

duced, decided not to update their PBX software to implement the standard, but use an external *protocol converter* to convert their proprietary protocol into the standard ECMA CSTA protocol and vice versa [21].

A large number of different interfaces and transport protocols is being used at the moment. Examples are: RS-232 with X.25 protocols, Ethernet with TCP/IP or LU6.2 protocols, Token Ring with TCP/IP or LU6.2 protocols, ISDN BRI with DSS1 protocols. Refer to appendix A for an overview of current CTI links [4].

3.2.2 ECMA CSTA

The ECMA CSTA standard consists of third-party call control service definitions [1] and an OSI conformant application layer protocol [2] that uses common OSI application layer elements: *Remote Operation Service Element* (ROSE) for the client/server interaction between switch and computer and *Association Control Service Element* (ACSE) for the establishment of an association between the application layer entities in the switch and the computer. The protocol is independent of the lower-layer protocols used to carry the CSTA *Application-layer Protocol Data Units* (APDUs). A possible CSTA protocol suite is shown in Figure 8. Communication between the computer and the switch may take place via intervening networks which range from a simple point-to-point connection to a Local or Wide Area communications Network.



Greyed layers needn't be fully implemented.

Figure 8. CSTA protocol suite

CSTA Call model

In the traditional call model, when two devices are connected, they are modelled as two device objects and one connection object that connects the two device objects, as shown in Figure 9a. On the contrary, CSTA uses a call model as shown in Figure 9b. The connection of two devices is modelled with no less than five objects: two device objects, a call object and two connection objects that connect the device objects with the call object. An important advantage of this model is that a multi-party (conference) calls can easily be modelled. When a party is added to a conference call, only a new device object (for that party) and a new connection object (between the new device object and the original call object) need to be created.

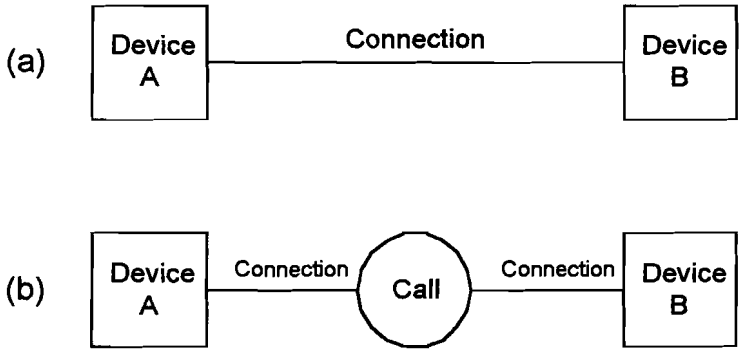


Figure 9. Traditional (a) versus CSTA (b) call model

The CSTA device objects have three properties: device class, type and state. The device class indicates the type of information generated by the device and can be one or more of the following: data, image, voice or other. The device type would normally be *station* for a traditional telephone device, but may be anything of the following: *ACD, ACD group, button, button group, line, line group, operator,*

operator group, station, station group, trunk, trunk group. The device state is a collection of the states of all related connection objects. Device objects can be identified in two ways: using a static device identifier or a dynamic device identifier. A static identifier would typically be a phone-number; a dynamic identifier is a shorthand identifier that may be created and reported by the switch after the device has been included in a call.

The CSTA connection objects have only one property: the connection state. Possible connection states are: *Null* (no relation between device and call), *Initiated* (the "dialling" state), *Alerting*, *Connected*, *Held*, *Queued* or *Failed*. Identification of connection objects is done by CSTA connection identifiers, being a combination of a device identifier and a call identifier.

The CSTA call objects also have one property: the call state. The state of a call object is a collection of all related connection object states. For common combinations of connection states in a two-party call, so-called *Simple Call States* are defined. For example, the Simple Call State of a call with two connections, both in Connected state, is defined as *Established*. Call objects are identified by call identifiers generated by the switch at the moment the object is created.

CSTA Services

ECMA CSTA uses a mutual client/server relationship between the switch (*Switching Function*) and the computer (*Computing Function*). Several types of services are defined, the most relevant being:

- *Switching Function Services*, where the switch is the server and the computer is the client. The computer sends requests to the switch, the switch performs the requested actions and sends results to the computer. Supported services are listed in Table 2. For a more detailed description, refer to appendix B.
- *Computing Function Services*, where the computer is the server and the switch is the client. The switch sends requests to the computer, the computer performs the requested actions and sends results to the switch.
- *Status Reporting Services*. Used by the computer to inform itself of the current status of calls, devices or ACD agents.

Table 2. CSTA Switching Function Services

Make Call	Establish a connection between two specified stations
Answer Call	Answer a call that is alerting on a specific station
Clear Call	Clear a specific call and release all stations involved
Clear Connection	Remove a specific station from a call
Hold Call	Put a call on a specific station on hold
Retrieve Call	Retrieves a held call
Consultation Call	Put the active call on hold an establish a connection with another station
Reconnect Call	Reconnect the held call and drop the consultation call
Alternate Call	Alternate between the two calls
Transfer Call	Transfer the held call to the station the consultation call was to
Conference Call	Start a conference call between the stations involved in both calls
Divert Call	Activate switch features to divert incoming calls from one station to another
Call Completion	Activate switch features to complete a call to a busy station
Make Predictive Call	Establish a connection between two specified stations by first connecting the called station and then (if succeeded) connecting the calling station.
Set Feature	Set device user features (Message Waiting lamp, Forwarding) or ACD agent state.
Query Device	Query the type, class and state of a specific device or the ACD agent state.

Note that the ECMA CSTA standard doesn't state that any of these services must be implemented for full compliance with the standard. It states that the services that are implemented must follow certain rules.

Computing Function Services

Supported Computing Function Services are requests for routing information. A typical application of these services is an ACD configuration as shown in Figure 1 (section 2.1.1). Note that the ECMA CSTA standard has facilities to set and query the state of ACD agents, therefore also configurations where the ACD functionality is located in the PBX are supported.

In general, these services bear a similarity to the interactions that take place between a switch (*Service Switching Point*) and a computer (*Service Control Point*) in an Intelligent Network: upon detection of a certain condition, the switch interrupts its call process and notifies the computer. Based on information provided by the switch (calling number, called number) and/or other information (e.g. time of day), the computer decides how the call process should continue and returns control to the switch.

Status Reporting Services

The computer has two methods to inform itself of the state of calls, devices and ACD agents:

1. With the **Snapshot Call** and **Snapshot Device** services, the computer can actively determine the current state of calls and devices, respectively.
2. With the **Monitor Start** service, the computer can initiate a so-called monitor on a call or a device (with associated ACD agent), after which the switch will keep the computer informed of changes in the state of that call, device or ACD agent by means of *Event Reports*. The switch keeps sending these Event Reports until the computer uses the **Monitor Stop** service to cancel the monitor. The most relevant Event Reports are listed in Table 3.

Table 3. CSTA Event Reports

Service initiated	The monitored device is taken off-hook or has invoked a switch feature
Originated	The monitored device has initiated a call
Delivered	The monitored device is ringing (inbound call) or a specific device is ringing because the monitored device has initiated a call
Established	The monitored device has been connected to a specific device
Connection Cleared	The monitored device has been disconnected
Call Cleared	A specific call has been cleared, resulting in all involved devices being disconnected
Held	A specific device has put a call on the monitored device on hold
Retrieved	An specific held call has been reconnected
Transferred	A specific call has been transferred to another device
Conferenced	Two specified calls are merged into one conference call
Queued	A specific call is being queued (for example in an ACD)
Network reached	A specific call has reached a boundary (typically an outbound trunk) after which only limited event reporting is possible
Failed	A specific call cannot be completed for some reason, for example because the called number is invalid or because the remote station is busy.
Agent state	The state of a specific ACD agent has changed. Supported states are: Logged On, Logged Off, Not Ready, Ready, Work Not Ready and Work Ready.

3.2.3 NetWare Telephony Services

As the number of host computer systems decreased (were replaced by PC-LAN systems), a need for a more distributed third-party call control configuration arose. Novell and AT&T recognised this fact and co-operated to develop an extension for the NetWare LAN operating environment that provides every NetWare client on the LAN with third-party call control capabilities. This extension is called *NetWare Telephony Services*. The third-party call control capabilities are provided by a NetWare *Telephony server* (which may be a NetWare file server at the same time) that is connected to the PBX through a CTI link. Figure 10 depicts the architecture of this system.

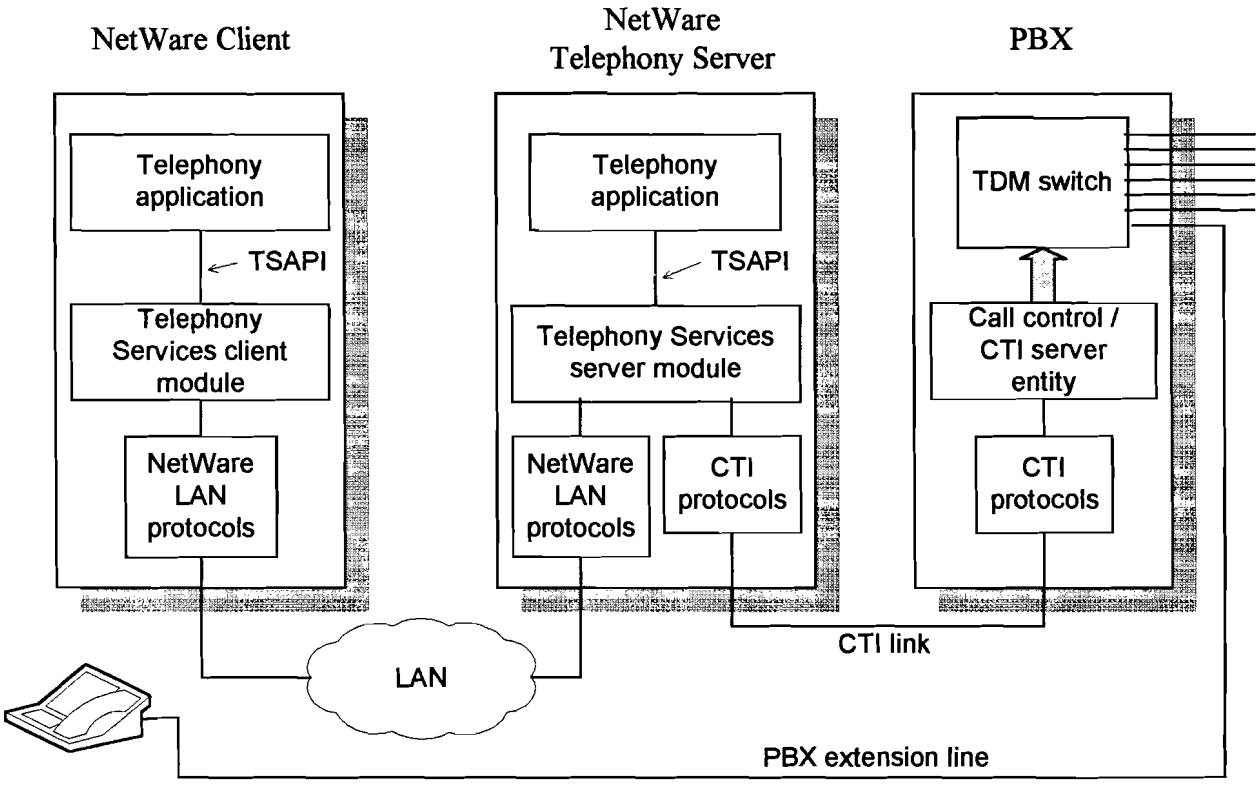


Figure 10. NetWare Telephony Services architecture

The NetWare Telephony server contains the necessary, PBX specific hardware and software (in a PBX driver module) to communicate with the PBX. Telephony applications running on the server use a well defined, PBX independent API for call control operations at a relatively high level. This API is based on the ECMA CSTA function definitions and is called *Telephony Services API* (TSAPI) [17]. Besides the Telephony server, there can be a number of NetWare clients that run telephony applications. The telephony applications on the clients use the same API and are provided the same capabilities as the ones on the Telephony server. For this purpose, the Telephony server acts as a *CTI gateway* on the LAN.

To enable telephony applications written for Microsoft's TAPI on the NetWare clients, mapping software between TAPI and TSAPI (called *Tmap*) has been developed by Northern Telecom, Microsoft, Intel and Novell Inc.

The NetWare Telephony Services configuration is one that is based on existing technology. All that is needed is a PBX with a CTI link (most PBXs already have one or can easily be equipped with one), a NetWare LAN (70% of the installed base of LANs are operating under NetWare) and, of course, the

NetWare Telephony Services extension software and PBX driver. The configuration offers three important advantages over first-party call control configurations:

1. The clients don't need additional telephony hardware; all call control information is carried over the LAN. Only the Telephony server needs to be equipped with additional hardware.
2. The clients send and receive call control information through an interface that is independent of the type of PBX used. Only the Telephony server needs PBX specific hard- and software.
3. All clients have full third-party call control capabilities, just like the Telephony server. More specifically, a client can monitor and control any station that is connected to the PBX.

3.2.4 Computer Integrated Communication Network

A disadvantage of the NetWare Telephony Services configuration, is the lack of *voice/data integration*. In a typical computer supported call control situation, a computer needs to control a telephone that may not be physically connected to the computer, but is located in its vicinity (e.g. on the same desktop). The telephone is connected to a PBX through an extension line. In the NetWare Telephony Services configuration, the computer is connected to the same PBX, but not through the same extension line. As a matter of fact, the call control information from the computer has to take a rather complex detour through a LAN, a telephony server and possibly a CSTA protocol converter (refer to section 3.2.1).

The lack of voice/data integration in the NetWare Telephony Services configuration has been a motivation to design an alternative configuration. The concept of this new configuration is shown in Figure 11. Since the computer is really integrated in the network (the computer controls and uses the network), the concept has been named *Computer Integrated Communication Network* (CICNet).

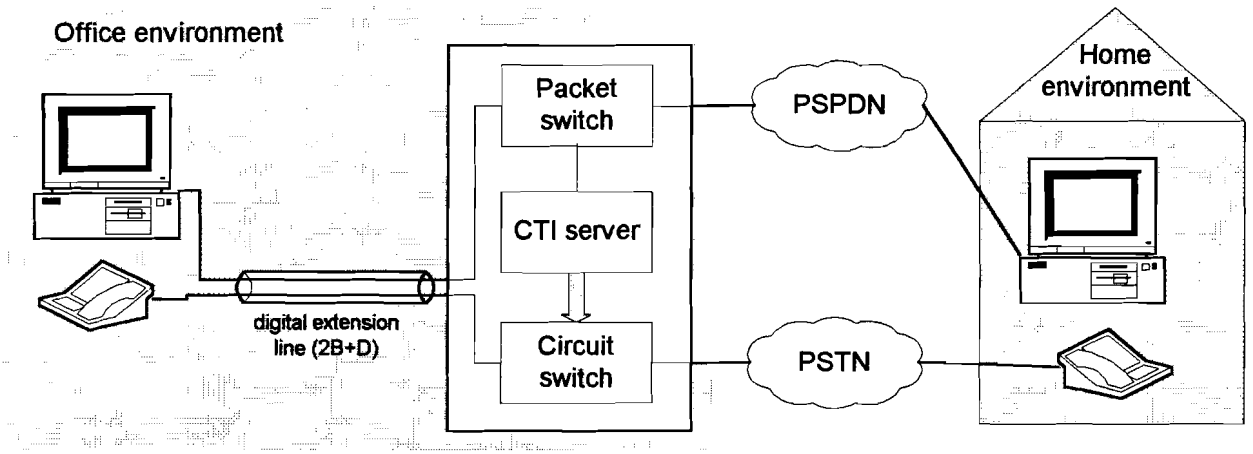


Figure 11. CICNet concept (abstract view)

In the CICNet concept configuration, both telephone and computer are connected to the digital extension line. Many modern digital PBXs allow for this kind of voice/data integration by means of ISDN(-like) interfaces (*2B+D* format). Since the only connection between the workplace (consisting of computer and telephone) and the switch is a digital extension line, this configuration is also feasible in an ISDN Centrex situation.

While the telephone is connected to a circuit switch in the usual way, the computer is connected to a packet switch. This packet switch allows the computer to communicate with the CTI server and also to other computers in the network. When both CTI server and computer implement a standard call control

protocol like ECMA CSTA, a CTI gateway like the Telephony server in the NetWare Telephony Services configuration can be disposed[11] of.

Another aspect is remote access through a public network, desirable for teleworking (e.g. ACD agents working at home). As shown in Figure 11, remote access can be achieved by connecting the packet switch to a *Packet Switched Public Data Network* (PSPDN). For a teleworking arrangement, a connection to both the *Public Switched Telephony Network* (PSTN) and the PSPDN is required. Alternatively, a public ISDN may offer integrated access.

Note that signalling capabilities across a PSTN or ISDN are typically very limited. For example, it is not possible for the PBX to initiate a call from the teleworker's phone. Therefore, it is desirable to have a PBX feature where both parties in a call are being called by the PBX. After both parties have answered, they are connected (internally in the PBX) and become involved in a normal call. In this manner, charging issues are also solved: call charges are automatically on the company's phone-bill, instead of on the teleworker's personal phone-bill. Current PBXs do not support such a feature, neither does the ECMA CSTA standard. This is a very relevant enhancement of current systems, though.

Dial-in access to CTI server

An easy way to implement the CICNet concept using an existing PBX is shown in Figure 12.

A packet switch is connected to the CTI link of the PBX. A number of PBX extension lines is connected to dial-in ports in the packet switch. To access the CTI server, a computer first has to establish a circuit-switched connection to the packet switch by dialling the phone-number associated with a dial-in port in the packet switch. All extension lines connected to the packet switch can be assigned a single phone number on the PBX (a *hunt-group*). The PBX will then automatically search an available extension (dial-in port) within that group.

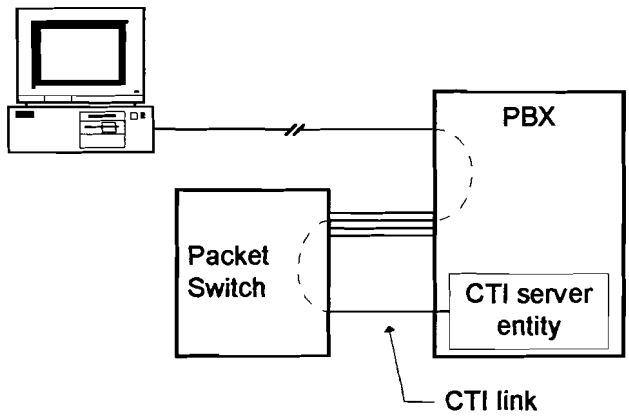


Figure 12. Dial-in access to CTI server

After the connection to the packet switch has been established, packet switching techniques should be used to gain access to the CTI server. In case of an X.25 packet switch, this means that the computer has to establish a *Virtual Circuit* with the CTI server by sending an X.25 *Call Request* packet with the a remote address field equal to the X.25 address associated with the CTI server.

An important prerequisite for this configuration to work is that the CTI server entity allows multiple sessions, i.e. it must know how to deal with incoming data calls and allow simultaneous communication with multiple computers over multiple Virtual Circuits. Because most CTI links were designed with a simple point-to-point connection to a single host computer in mind, this requirement may not be fulfilled by some of current CTI servers.

The number of connections that can simultaneously be made to the CTI server in this manner are limited by the amount of dial-in ports in the packet switch, the number of Virtual Circuits that the CTI link can carry and the bandwidth of the CTI link. In case of an X.25 link, the maximum number of Virtual Circuits is theoretically 4096 and is therefore not a real limitation. To accurately determine the required

[11] *Besides acting as a simple CTI gateway, the Telephony Server also takes care of security. Disposing the Telephony Server implies that this security function should be implemented either in the packet switch or in the CTI server.*

of bandwidth requirements, a simple calculation concerning a computer initiated call is considered here. For a computer initiated call, approximately 300 bytes have to be exchanged in total. With a 19.2 kbps CTI link, this means approximately 8 calls can be established per second (30,000 calls/h), which is a very respectable performance for a PBX.

With dial-in access to the packet switch, a remote access configurations can be realised without the need for a PSPDN. An ISDN BRI offers two B-channels, one of which can be used for a dial-in data connection to the packet switch, while the other remains available for voice communication. A problem with this approach lies in the charging aspects involved in public networks. Circuit switched connections are generally charged according to the duration of the call. Packed switched connections, on the contrary, are charged according to the amount of data exchanged. Access to the CTI server requires a *semi-permanent* connection, i.e. the connection will typically be established as soon as the workstation is turned on and it will not be released before the workstation is turned off again. However, the actual amount of data exchanged over the connection is very low. Therefore, using a PSPDN for this connection will be much more cost-effective than using a (circuit-switched) dial-in connection.

Packet mode access to CTI server

Although the dial-in configuration is easy to implement in current systems, it imposes a problem in systems with a large number of workstations. In order to communicate to the CTI server simultaneously, each workstation requires an additional extension line between the PBX and the packet switch and a dial-in port in the packet switch. Unless digital trunk lines (and matching dial-in ports) can be used to connect the PBX to the packet switch, this means a very high additional cost per workstation. Another way to solve the problem is integrating the packet switching functionality in the PBX. However, this requires an enhancement of the PBX design. The typical architecture of current digital PBXs is depicted in Figure 13.

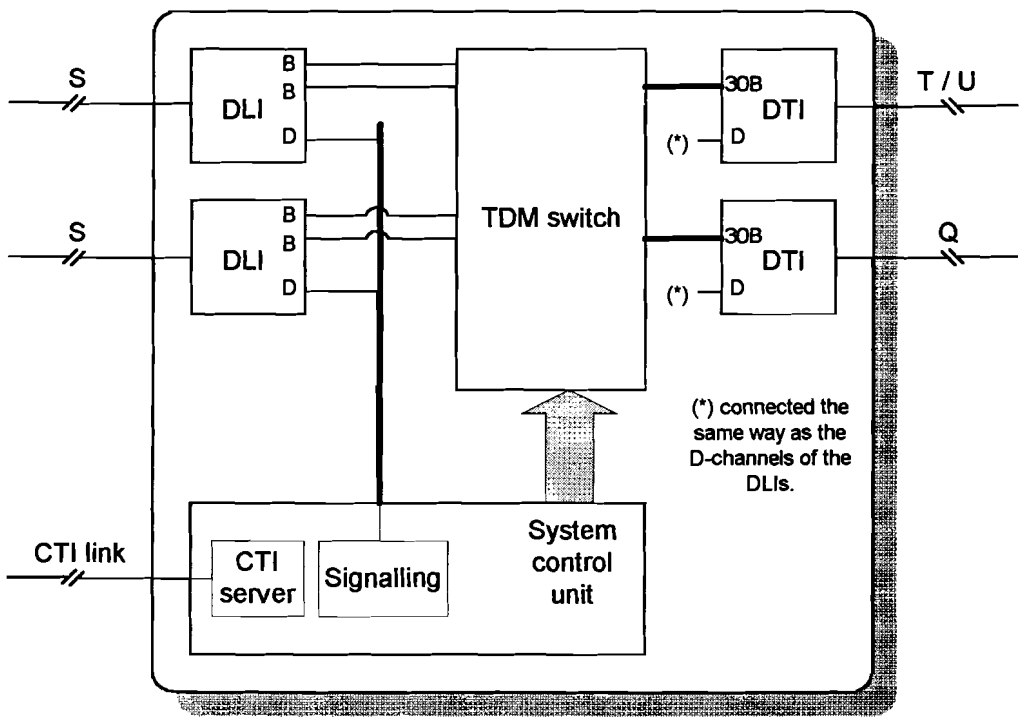


Figure 13. Typical digital PBX architecture

All extension lines (S reference points) are terminated in *Digital Line Interface* (DLI) modules. Trunks to a public ISDN (T/U reference points) and trunks between PBXs (Q reference points) are terminated in *Digital Trunk Interface* (DTI) modules. These interface modules separate the signalling channel (D-

channel) from the voice/data channels (B-channels). The B-channels are fed into a *Time Division Multiplexing* (TDM) switch. A *system control unit* controls the connections within the TDM switch. This system control unit can also address each interface module individually to exchange signalling info. The system control unit generally polls each interface module to see if signalling info has arrived on a particular interface module. If so, the corresponding D-channel data is read by a signalling entity inside the system control unit. A CTI server entity inside the system control unit allows external third-party call control through a CTI link.

This architecture is not appropriate for user packet data transport over the D-channel:

- All D-channel data must be processed by the system control unit. For user packet data, the system control unit must first read the data from a line interface and then write it to another (depending on the destination address). With some 250 DLIs, this means the system control unit may have to process an aggregate data stream of 4 Mbps, only for transport of user packet data.
- Interconnection to another PBX or to a public ISDN is done by 2 Mbps trunk lines. These lines carry 30 B-channels, so 30 stations (each using only one B-channel) can be interconnected using one trunk line. When all stations also exchange user packet data on their D-channels, a bandwidth of 384 kbps would be required for packet data on the trunk line. However, a trunk line only has one 64 kbps D-channel. The reason for this is that the trunk lines are not intended to carry user packet data on their D-channel. The D-channel on a trunk line is used for signalling purposes only.

To accommodate for user packet data in a public ISDN, the ISDN local exchanges are supposed to route all packet data over a separate *Packet Switched Public Data Network* (PSPDN). Two cases for ISDN access to a PSPDN are defined in the X.31 recommendation [11]:

A) Dial-in access to a PSPDN via an ISDN. Only B-channel access is possible. This is the kind of access described in the previous section.

B) ISDN virtual circuit service, also known as Packet Mode Bearer Services. Access to the PSPDN is integrated in the ISDN local exchanges. This configuration is depicted in Figure 14. Both B-channel and D-channel access is possible. Advantages of D-channel access are that both B-channels remain available for isochronous data and that multiple devices on the same S-bus can simultaneously use the D-channel for user packet data transfer.

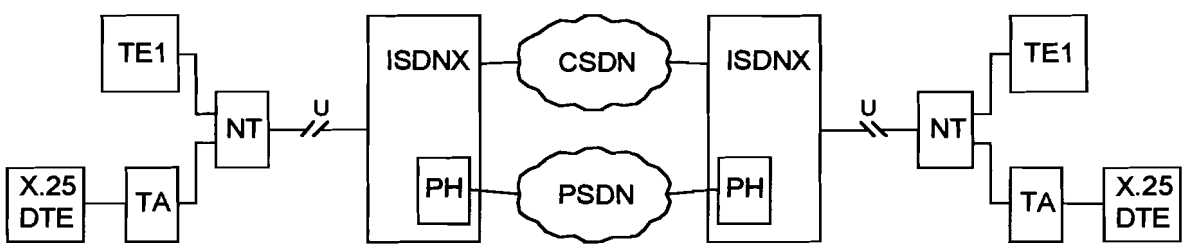


Figure 14. X.31 case B

For X.31 case B access to a X.25 PSPDN, a *Packet Handler* (PH) function needs to be integrated in the ISDN local exchanges (ISDNX). The Packet Handler actually a node of the PSPDN and interfaces to it using network-internal (X.75) protocols.

To initiate an outbound X.25 call, an ISDN terminal (TE1) or a ISDN Terminal Adapter (TA) first has to establish a connection (B-channel access) or logical link (D-channel access) to the Packet Handler in the ISDNX.

- For B-channel access, the TE1 or the TA sends an ISDN (Q.931) call set-up message to the ISDNX, requesting packet mode bearer services. The ISDNXs will then route the associated B-channel to the Packet Handler instead of to the *Circuit Switched Data Network* (CSDN).

- For D-channel access, the TE1 or the TA initiates a LAPD (Q.921) logical link with a Service Access Point Identifier (SAPI) value of 16. The ISDNX automatically routes D-channel data on this logical link to the Packet Handler. For signalling purposes, a logical link with SAPI = 0 is used.

After the connection to the Packet Handler has been established, normal X.25 call set-up procedures can be initiated by the TE1 or the X.25 DTE. When an incoming X.25 call arrives at the Packet Handler and there is no connection to the terminal yet, the ISDNX sends an ISDN (Q.931) call indication message (specifying packet mode bearer services) to the TE1 or TA. If the message is acknowledged, either a B-channel connection or a logical link on the D-channel with SAPI = 16 is established with the TE1 or the TA and normal X.25 call set-up procedures with the TE1 or the X.25 DTE follow.

A PBX architecture that incorporates the X.31 case B access concept (and therefore a Packet Handler function) is shown in Figure 15.

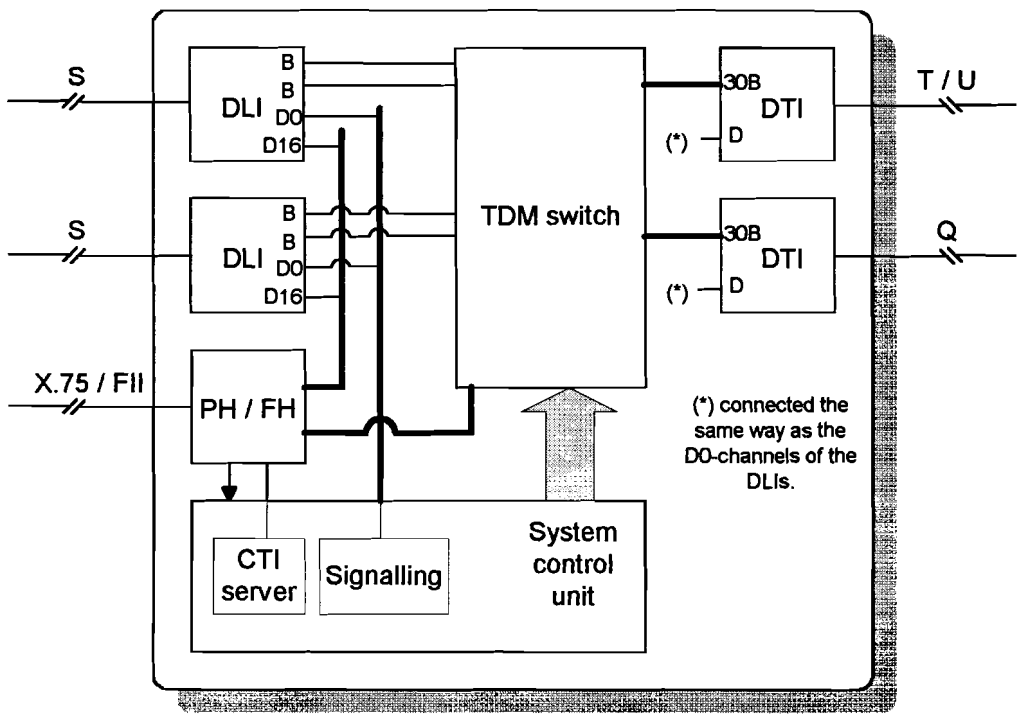


Figure 15. ISDN PBX incorporating X.31 case B access

Each DLI separates the signalling info on the D-channel (D0: SAPI = 0) from the user packet data on the D-channel (D16: SAPI = 16). Signalling info is fed into the signalling entity in the system control unit in the same way as described earlier. User packet data is fed into the Packet Handler[12], thus enabling D-channel access to the Packet Handler. To allow for B-channel access, a number of B-channels from the TDM switch must also be connected to the Packet Handler.

The CTI server entity inside the system control unit is connected internally to the Packet Handler, instead of to an external CTI link. In this manner, the CTI server can be accessed from any extension line, either on the B-channel or on the D-channel, using X.31 case B procedures. Using X.31 case A (dial-in) procedures, the CTI server can also be accessed over a trunk line (on a B-channel) and therefore through a public ISDN or another PBX.

[12] *Alternatively, this may be a Frame Handler (FH). A Frame Handler uses Frame-relay techniques instead of (X.25) packet switching.*

If the Packet Handler is externally connected to a X.25 PSPDN, the CTI server can also be accessed through that PSPDN. This, in turn, allows B-channel and D-channel access through a public ISDN that supports X.31 case B access (and uses the same PSPDN).

3.3 Conclusions

First-party call control configurations present an easy way to control telephony devices with a computer. However, in most configurations the call control possibilities are very limited. Configurations that allow more extensive call control require PBX specific hard- and software, unless standard ISDN extension lines are available. Another typical aspect of first-party call control configurations is the need for a physical connection between the computer and the device to be controlled. This imposes a problem in situations where, for example, this device is a cordless (e.g. DECT) telephone.

Third-party call control configurations are based on a concept similar to the IN concept in public networks: additional network functionality is implemented on a computer that interfaces directly with the switch, instead of in the switch itself. With the introduction of the NetWare Telephony Services configuration, third-party call control also becomes available at the desktop, making this configuration compete head-on with first-party call control configurations.

A disadvantage of the NetWare Telephony Services configuration is the lack of voice/data integration: although most PBX systems allow simultaneous voice and data communication over a single extension line, the call control data exchanged between PBX and computer is forced to take a detour through a LAN. The conceived CICNet configuration does implement the voice/data integration concept, resulting in a third-party call control configuration that does not need a LAN, allows teleworking and may also be used in a Centrex situation.

4. CTI demo system design

In order to get insight in the issues involved in programming a CTI system, i.e. both driver software and telephony application software that uses a telephony API, a demo system has been developed. Design objectives for this demo system have been:

- *Implementation of the CICNet concept.* Refer to section 3.2.4.
- *Implementation of the ECMA CSTA protocol.* Use of a standard call control protocol is part of the CICNet concept. The PBX used supports the ECMA CSTA protocol and the protocol specification is available free of charge from ECMA.
- *Use of the Windows Telephony specification* [16]. The Windows Telephony system is an open standard and as such can be used for all kinds of configurations. A developers kit for Windows 3.1 is available free of charge from Microsoft. The Windows Telephony system will be a standard component of the new Windows 95 version.
- *Demonstration of two CTI applications: screen based telephony and multi-media call control.* Refer to section 2.2.1 and section 2.3.2, respectively.
- *Use of object-oriented programming methods in C++.*

4.1 Configuration

The demo configuration is based on Ericsson equipment: MD110 PBX, ApplicationLink CTI link, ERIPAX packet/frame switch and Terminal Adapter Units. This is pre-ISDN[13] equipment: besides analogue line interfaces, the MD110 uses a proprietary digital line interface that is similar to an ISDN BRI (2B+D format). For voice communication on a digital extension line, proprietary digital phone-sets are used. For data communication, the PC is connected to a Terminal Adapter Unit (TAU) through a standard RS-232 interface. Voice and data channels (both B-channels) are multiplexed inside the TAU onto the proprietary extension line format. Although the TAU accepts standard modem commands for first-party call control applications (refer to section 3.1), these features are not used in the demo configuration. The TAU is merely used for a bit-transparent, semi-permanent connection across the MD110 PBX to the packet switch, as shown in Figure 16.

[13] *Standard ISDN PRI trunk interfaces are readily available. Standard ISDN BRI (S-bus) extension lines have been introduced very recently.*

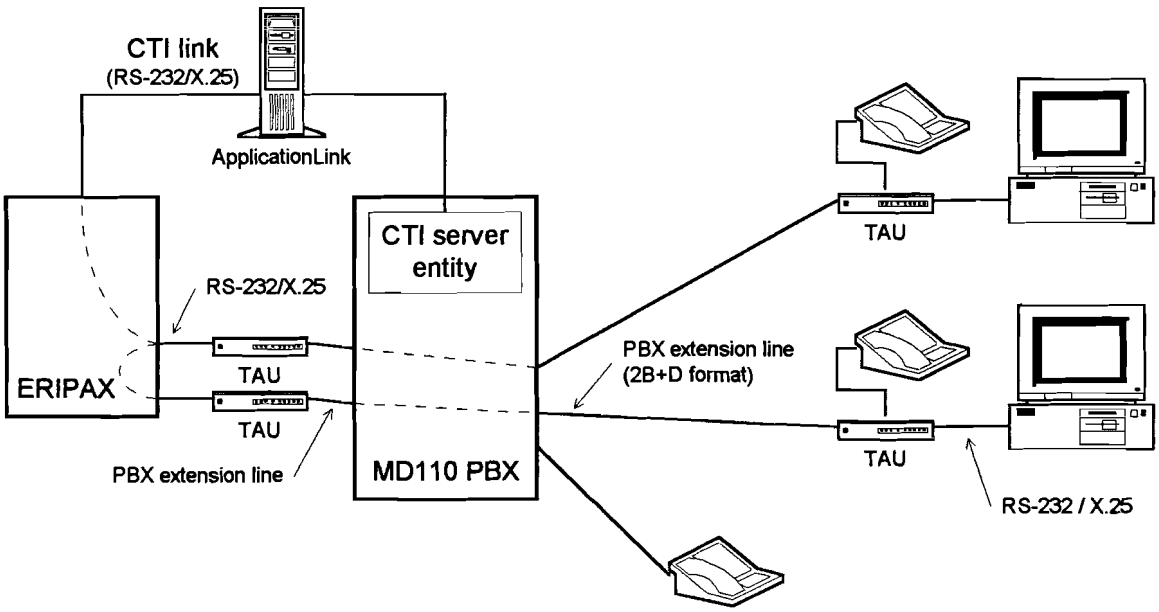


Figure 16. Demo configuration

The ERIPAX packet/frame switch can be configured with both X.25 and Frame Relay ports. For X.25 ports, several physical interfaces are supported: V.24/V.28 (RS-232) for transmission rates up to 19.2 kbps and X.21, V.24/V.35, V.36 or G.703 for transmission rates up to 2 Mbps.

The MD110's CTI link is called ApplicationLink and uses a PC as external protocol converter (refer to section 3.2.1) to support the standard ECMA CSTA protocol. Since the CTI link is based on a RS-232 /X.25 interface, direct connection to the X.25 packet switch is possible. In the demo configuration's CTI link only one X.25 Virtual Circuit is supported, being a Permanent Virtual Circuit (PVC). The ERIPAX has been configured to maintain a PVC between the ApplicationLink and one of the two computers. As a result, only one of the two computers can communicate with the ApplicationLink. The other computer is used for mutual data communication purposes only: the two computers can establish a Switched Virtual Circuit between each other.

The computers are equipped with EICON X.25 communication boards. These boards offer a physical RS-232 or X.21 interface and implement link layer (*HDLC/LAPB*), network layer (X.25 PLP) and optionally transport layer (OSI) protocols in an on-board microprocessor system. For programming purposes, an X.25 network-level developers toolkit is required, which offers an API at the network layer level.

4.2 Software design

The software that has to be implemented in the computers is illustrated in Figure 17. The stack of boxes represents the OSI 7 layers model (see also section 3.2.2). ECMA CSTA is an application layer (layer 7) protocol and relies on the services provided by the underlying layers. The ApplicationLink has been designed to work with a single, reliable connection to a host computer, which has been a reason to use a reduced OSI stack, namely the ITU-T X.25 recommendation (1980). The ApplicationLink maps the CSTA APDUs directly to X.25 packets (NPDUs) and vice versa, thus skipping layers 4 to 6. Also, the *Association Control Service Element* (ACSE) is not used by the ApplicationLink. An association is implicitly established upon receipt of an X.25 *Restart Request*.

Layers 1 to 3 are implemented in the EICON X.25 communication board. The OSI *Remote Operations Service Element* (ROSE), the ECMA CSTA protocol and a mapping between the TAPI (or actually TSPI) and ECMA CSTA need to be implemented in the form of a Service Provider that complies with the TSPI specification. The same Service Provider is also used for data communication with other computers, but a different programming interface should be used for this purpose. This interface will be discussed in section 4.2.4.

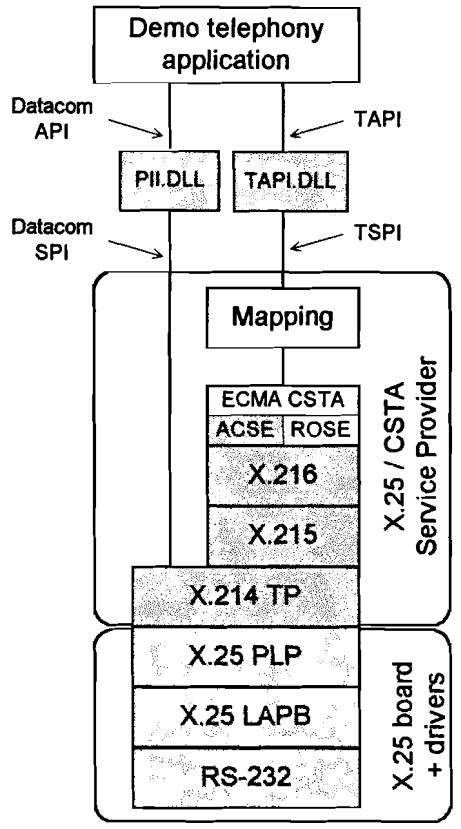


Figure 17. Software architecture

The CTI concepts to be demonstrated (Screen based telephony and Multi-media call control) will be implemented in a single telephony application which interfaces with the Windows Telephony system through the Telephony API (TAPI).

4.2.1 ECMA CSTA implementation

The ECMA CSTA protocol is (like all OSI application layer protocols) defined in *Abstract Syntax Notation #1* (ASN.1) [12]. This notation is used to define complex data structures in an abstract, machine-independent manner. Using certain encoding rules, the abstract syntax can be converted into a so-called *transfer syntax*, which contains the actual data to be transferred and some formatting information that allows the receiving side to reconstruct the original data structure. The conversion between abstract syntax and transfer syntax is typically a duty of the presentation layer. Currently, only one set of encoding rules has been standardised: the *Basic Encoding Rules* (BER) [13].

Implementation of protocols defined in ASN.1 is usually done by means of an ASN.1 compiler. This compiler typically translates the ASN.1 specification into a C or C++ module, which can then be integrated with the rest of the software. Poor availability of such ASN.1 compilers, however, has been a motive to use a different approach.

Object-oriented programming techniques and many other powerful features of C++ have been applied to create a module that extents the C++ language with a new class of data types, namely the data types available in ASN.1. These new data types (which are implemented as object classes) contain, among others, methods to encode and decode themselves using the Basic Encoding Rules. By employing this module, an ASN.1-like notation can be entered directly in a C++ module, without the help of an ASN.1

compiler. For a comprehensive discussion of this ASN.1/BER implementation module, refer to appendix D. Using the ASN.1/BER module, both the ROSE protocol [15] and the ECMA CSTA protocol [2] can easily be implemented.

Figure 18 depicts the model upon which the ROSE protocol is based [14]. In a client/server interaction, the client requests services from the server by means of a *RO-Request*[14] APDU. When the server has successfully completed the requested service, it may report a result back to the client by means of a *RO-Result* APDU. If an error occurred during the processing of the request, the server will report the error to the client by means of a *RO-Error* APDU. If either client or server receives a faulty APDU (e.g. not a ROSE APDU), it will notify the other party by means of a *RO-Rejection* APDU.

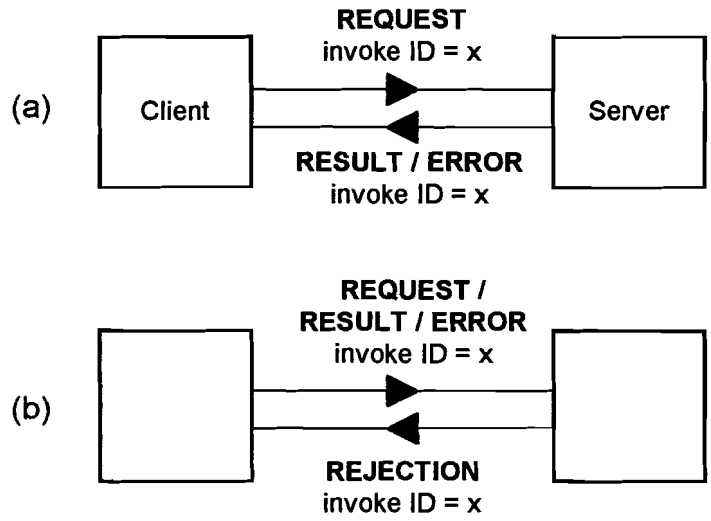


Figure 18. OSI Remote Operations model

In order to be able to relate a result or error to the associated request, so-called invoke identifiers are used: a RO-Result or RO-Error APDU will always contain the same invoke ID as the RO-Request APDU it is related to. Likewise, a RO-Rejection APDU contains the same invoke ID as the APDU that caused the rejection. Besides an identification of the type of APDU (request, result, error or rejection) and the invoke ID, the RO-Request, RO-Result and RO-Error APDUs can contain any ASN.1 data-structure as specified by the user of the *Remote Operation Service Element*, in this case the ECMA CSTA protocol.

[14] *Officially, the standard calls this RO-Invoke.*

4.2.2 Windows Telephony programming

Programming model

Figure 19 depicts the Windows Telephony programming model [16]. Both TAPI.DLL and the Telephony Service Providers are Dynamic Link Libraries (DLLs). As such, they export a number of functions that can be called from outside the DLL. An application (or other DLL) can access the functions in a DLL in two ways:

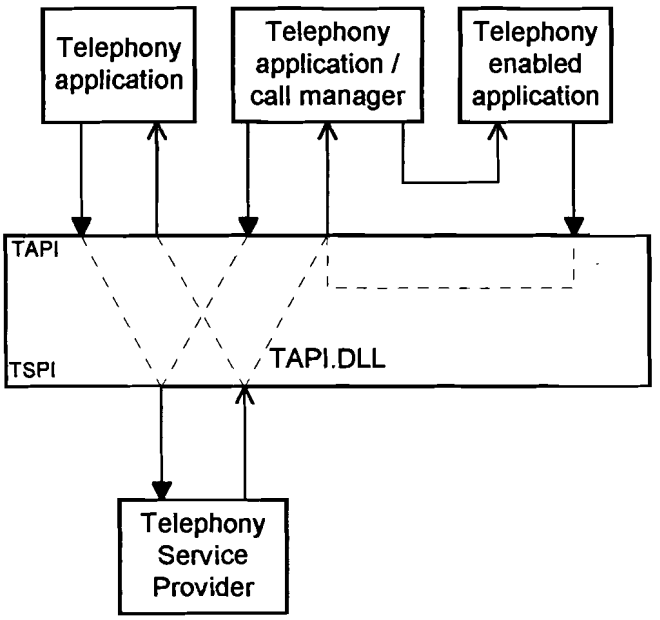


Figure 19. Windows Telephony programming model

1. Using *static linking*. This is based on the same kind of linking process as occurs with "ordinary" library files: during the development process of the application, a file is required that identifies the entry points of each exported function defined in the library. The final application "knows" these entry points. The difference between ordinary libraries and Dynamic Link Libraries is that ordinary libraries become part of the application, while Dynamic Link Libraries are shared among applications and are loaded by Windows (if necessary) at the moment the application is started.
2. Using *dynamic linking*. With dynamic linking, the application does not have former knowledge of the entry points of the functions in the DLL. These entry points are determined by the application at runtime. In this manner, the DLL is less tied to the application: changes in the DLL can be made without the need of recompilation of the application(s) that use the DLL. The determination of entry points implies a small programming overhead, however.

Typically, telephony applications would link to TAPI.DLL staticallly (this requires the file TAPI.LIB during the development process), while TAPI.DLL links to the Telephony Service Providers dynamically. This results in the advantages of both methods being combined: no programming overhead for the applications and flexibility in the link towards the Telephony Service Providers.

Using conventional C-style library function calls (shown as solid arrows in Figure 19), telephony applications can request services from TAPI.DLL. TAPI.DLL will simply translate many of these request in corresponding C-style library function calls in one of the Telephony Service Providers. Some of the services, called *synchronous* services, can be completed entirely within the function call. These functions return a value of zero if the service was completed successfully, or a negative error code otherwise. Many telecommunication services, however, require a relatively long time to complete and should therefore not be completed within one function call, since this will interrupt the system for too long a period. These services, called *asynchronous* services, will only be initiated during the function call. If this initiation was successful, the function returns a positive request identifier. At the time of asynchronous completion of the service (either successful or unsuccessful), the application will be informed by means of a *message* containing the request ID of the concerned service request.

The process of informing the application is not as trivial as it may sound, since this requires that TAPI.DLL (a library) calls the application, which is certainly not a standard feature of a library. In

order to make this possible, the application should at initialisation time pass the entry point of a function inside the application, called *callback function*, to the library by means of a function call to an initialisation function inside the library. To keep initialisation simple, usually only one callback function is used for all different kinds of notifications. This is achieved by means of so-called *messages*. The callback function defines a couple of parameters: one to identify the message and some more to carry the contents of the message. An API (or SPI) that uses a callback function typically defines the meaning of each message, its contents and the way the contents are mapped to the parameters of the callback function. The use of messaging mechanisms based on callback functions (shown as open arrows in Figure 19) is a very common practice in the Windows operating system[15].

As is shown in Figure 19, most messages being send by TAPI.DLL to a telephony application's callback function are a direct result of messages being send to TAPI.DLL's callback function by a Telephony Service Provider. Examples of such messages are the already mentioned asynchronous completion messages, but also unsolicited messages that inform the application of certain telephony events (e.g. an incoming call). If a telephony application has registered itself as a call manager, it may also receive messages from TAPI.DLL that are a result of so-called *Assisted Telephony* requests from other applications.

The Assisted Telephony concept is intended for applications (called *telephony enabled* applications) that could use some simple telephony call control capabilities (making and dropping calls) but don't want to be concerned with the complex procedures involved in the telephony process. For this purpose, another telephony application is used that acts as a call manager and processes the requests from telephony enabled applications. The telephony enabled application submits an Assisted Telephony request by invoking a simple function within TAPI.DLL. TAPI.DLL stores the request and tries to locate an active telephony application that has registered itself as a call manager (also known as request recipient). If no such application is active, TAPI.DLL will load a known request recipient application from disk and launch it. The request recipient is then informed of a waiting request by means of a message (through its callback function). The request recipient retrieves the request and performs the required telephony actions. Upon completion, the request recipient notifies the application that submitted the request by sending a message to its *window procedure*.

[15] *Probably the most common callback function is the so-called window procedure that is associated with a window on the screen and receives all the messages intended for that particular window. All user actions (keyboard and mouse input) are reported to the application as messages through the window procedure. Also direct communication between applications, e.g. Dynamic Data Exchange (DDE), is done through the window procedures. For this purpose, the Windows system API contains functions to send messages to an application's (or actually a window's) window procedure.*

Call model

In the call model used by the Windows Telephony system, three kinds (*classes* in object-oriented speak) of objects are defined, as shown in Figure 20: line devices, addresses and calls. This call model is typically implemented in both TAPI.DLL and the Telephony Service Providers.

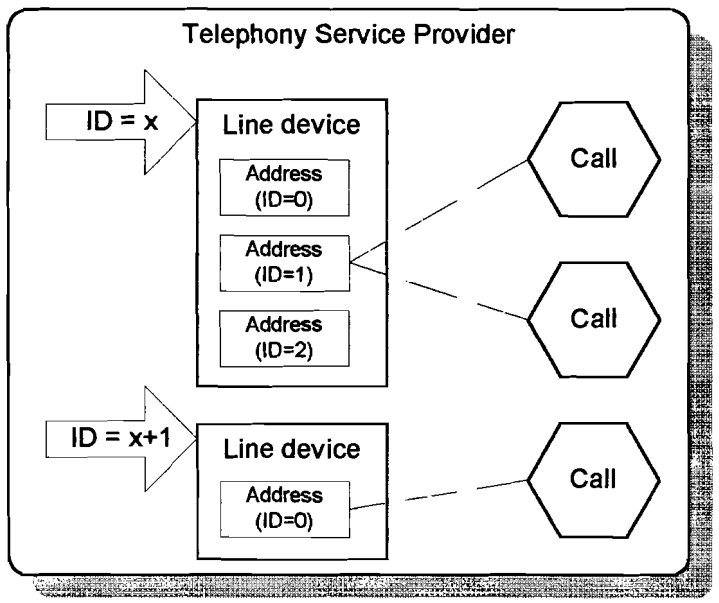


Figure 20. Windows Telephony call model

The line device object is an abstract representation of any kind of physical device (modem, fax board, digital phone emulation board, ISDN card) connected to a telephone line. In order to control a phone-set that is connected to the computer in a way described in section 3.1 (Figure 5c), another object is used: the phone device, but this object is not of interest here. Device identifiers are used to uniquely identify each line device in all Telephony Service Providers. At initialisation, TAPI.DLL informs each Telephony Service Provider of the device ID value that will be used to refer to the Service Provider's first line device. Since TAPI.DLL knows the number of line devices within each Service Provider, it can thus create a continuous, non-overlapping range of device identifiers.

In general, one line device has exactly one address (phone number) assigned to it. There are cases, however, where one line device can have multiple addresses. If a line supports multiple channels, each channel may have its own address. Multiple addresses on a single channel is also possible: in case of an incoming call, the network informs the user of the address concerned by means of *distinctive ringing* (different ringing patterns for each address) or *Direct-Dial-In* (the actual digits dialled are passed to the line device). Because each address may have different capabilities, addresses are modelled as separate objects within a line device object. Properties of address objects and the associated line object complement each other: the address objects contain the properties/capabilities that can be different for each address, the line device object contains the rest of the properties/capabilities that all addresses on that device have in common. In order to uniquely identify each addresses within a line device, address identifiers are used.

A call object represents a connection between two (or more) addresses. Zero, one or more calls can exist on a single address at any given time. An example of multiple calls on a single address is a consultation call: a call is put "on hold", while a consultation call is being made to another party. Unlike line device and address objects, call objects are dynamic. Call objects are created as a result of a service requested by a telephony application or as a result of external events such as an incoming call. As soon as a call object is created inside the Telephony Service Provider, TAPI.DLL is informed and a call handle is exchanged. This call handle is used for subsequent references to the call object and will usually be a pointer to the call object in memory (although it should not be used as such by anything but the Service Provider). Dropping a call does not automatically deallocate the associated call object since an application may still find it useful to extract information from the call object (such as for logging purposes). Applications must explicitly deallocate the call objects they own handles to. The most important aspect of a call object is its state. Possible call states are listed in Table 4. Additionally, the call object stores

information like the bearer mode (voice or data), the current media mode and party identification (calling party, called party, connected party, redirecting party).

Table 4. Windows Telephony call states

Idle	The call is inactive. This state typically occurs after the call has been dropped by the application.
Offering	The switch informs the computer of an incoming call. This does not necessarily mean the user is being alerted, e.g. in ISDN, the user not alerted before the offering call has been accepted by the phone device.
Accepted	An application has accepted the call. In ISDN, this has the side effect of alerting the users at both sides of the call. An incoming call can always be immediately answered, i.e. without being accepted first.
Dial-tone	The switch indicates it is ready to receive a number. (outbound calls)
Dialling	Digits are being dialled and collected by the switch.
Proceeding	Dialling is complete and the call is proceeding through the network.
Special info	A special info announcement is being received. This will happen if the dialled number is incorrect or another irregularity occurs.
Busy	A busy signal is received, indicating that either the network is congested or the remote station is busy.
Ringback	The destination has been reached and is being alerted.
Connected	A connection has been established.
On hold	The call is being held by the switch.
Conferenced	The call is member of a conference call and is logically in the connected state.
On hold pending conference	The call is on hold in preparation of being added to a conference call.
On hold pending transfer	The call is on hold in preparation of being transferred.
Disconnected	The call has been disconnected by the remote party.
Unknown	The call exists, but is state is currently unknown.

The Telephony Service Provider Interface

As mentioned earlier, Telephony Service Providers are Dynamic Link Libraries (as a minor variation, the filename extension is ".TSP" instead of ".DLL"). As such, they contain two basic functions that are called by Windows when the DLL is loaded (**LibMain**) and unloaded (Windows Exit Procedure: **WEP**). Within a LibMain/WEP pair, multiple telephony sessions can be performed, as shown in Figure 21a. In particular, when the telephony configuration is to be changed, the current session will be terminated. After the changes have been made, a new session will be started, resulting in the new configuration info being read. In addition two the LibMain/WEP pair, two functions are defined in the TSPI specification that are called by TAPI.DLL at the start (**TSPI_providerInit**) and the end (**TSPI_providerShutdown**) of a telephony session. All Telephony Service Provider interactions will occur within such a session. When the Service Provider needs to be reconfigured, the function TSPI_providerConfig will be called (this will always be outside a telephony session).

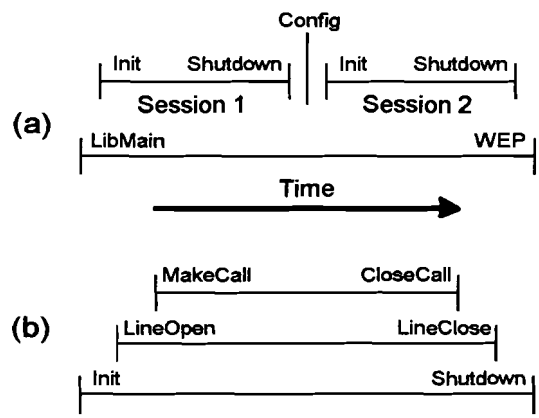


Figure 21. TSPI Sessions

While the telephony sessions just described enclose all Telephony Service Provider interactions, similar sessions enclose all interactions on a particular line device within the Telephony Service Provider, as shown in Figure 21b. A session on a particular line device is started with a call to the function **TSPI_lineOpen** (with the device ID as a parameter) and is ended with a call to the function **TSPI_lineClose**. Within such a session, the actual telephony activity on the line takes place. For example, a call may come into existence as a result of a call to the function **TSPI_lineMakeCall**. The call will always be removed (with function **TSPI_lineCloseCall**) before the session on the line is ended.

All asynchronous services, as described in section 4.2.2, expect the associated function to return a *request identifier*. These request identifiers are generated by TAPI.DLL and are passed to the Service Provider as a function parameter. In this manner, the Service Provider can store the request ID for subsequent use in an asynchronous completion message. The functions returns with either the request ID or a negative error code. In the latter case, the function failed and no asynchronous completion message will follow.

Besides functions for session begin and end and for activation of telephony services, the TSPI defines functions to retrieve information about line devices (status and capabilities), addresses (status and capabilities) and calls (status and additional call-related info). For a more detailed description of TAPI and TSPI functions, refer to appendix E.

4.2.3 Mapping TAPI to ECMA CSTA

An important precondition for a mapping to be meaningful is that the functionality of both parties match sufficiently. The result of a call control functionality comparison between ECMA CSTA and TAPI is shown in tabular form in appendix C. Briefly put, TAPI offers all the call control functionality defined in ECMA CSTA, except for typical third-party call control features like the CSTA Computing Function Services (refer to section 3.2.2) and ACD agent support.

The table in appendix C provides for each Switching Function Service defined in ECMA CSTA the name of a matching TAPI function. As such, it constitutes part of a mapping between TAPI and ECMA CSTA for call control communication in one direction: from the telephony application to the switch. This communication consists in CSTA service requests being sent from the computer (client) to the switch (server). At the TAPI level, this communication consists in the telephony application invoking asynchronous telephony services (refer to section 4.2.2) by means of C-style function calls to TAPI.DLL. The duty of the X.25/CSTA Service Provider is to assemble the proper CSTA service request, based on the function called and the parameters passed, and transmit it to the switch.

Both the ECMA CSTA protocol (based on ROSE, see section 4.2.1) and TAPI use a model where the outcome of an operation is reported asynchronously, i.e. the computer (client) doesn't have to suspend processing until the outcome is known. In order to be able to relate the outcome with the associated request, both use unique identifiers for all outstanding requests. The request identifiers used by TAPI satisfy all conditions to be used as ROSE *invoke ID* (used by the ECMA CSTA protocol) without the need for further mapping.

The communication from the switch to the telephony application consists in CSTA service results and errors and, if monitoring has been enabled, event reports being sent from the switch to the computer. The X.25/CSTA Service Provider maps the received CSTA service results (successful outcome) and service errors (unsuccessful outcome) to corresponding asynchronous completion messages, which are sent to the telephony application through TAPI.DLL. In order to be able to notify the telephony applications of incoming calls and, more generally, the changes in a call's state, a monitor must be enabled for each device managed. A mapping of CSTA event reports to Windows Telephony call states is shown in Table 5.

Table 5. CSTA event report mapping

CSTA call event report received:	New Windows Telephony call state:	Notes:
Service initiated	*dial-tone*	1
Originated	*proceeding*	
Delivered	*offering / ringback*	2
Established	*connected*	
Connection Cleared	*idle / disconnected*	3
Call Cleared	*idle / disconnected*	3
Held	*on hold / on hold pending ....*	4
Retrieved	*connected*	
Transferred	*connected / idle*	5
Conferenced	*conferenced*	6
Queued	*proceeding*	7
Network reached	*proceeding*	7
Failed	*busy / special info*	8

Notes:

1. The **Service initiated** event report is sent as soon as the monitored phone-set is taken off-hook. At this time, the Service Provider could report a new (outbound) call to TAPI.DLL with a *dial-tone* call state. The problem with this approach lies in the fact that the **Service initiated** event report may also be sent as a result of a CSTA originated call. For example, a valid scenario is (C = computer, S = switch):

   C => S: Make Call request
   C <= S: Service initiated event report (connection ID = x)
   C <= S: Make Call result (connection ID = x)

   In this example, the **Service initiated** event is sent as a result of the **Make Call** request and contains the connection ID of the new connection. The event report is followed by the result of the **Make Call** service, which also contains the connection ID of the same connection. To avoid creation of two call objects for the same call, one of the two should be ignored.

   Since the **Service initiated** event report needn't be sent as a result of a CSTA originated call, ignoring the **Make Call** result is out of the question. The problem with ignoring the **Service initiated** event report lies in the fact that at the moment the event is received, the computer has no way of knowing whether the event is a result of the **Make Call** request or not. Two alternative solutions to this problem are:

   - Ignore all **Service initiated** event reports, no matter if they were caused by the **Make Call** request or not.
   - Store the received **Service initiated** event report, but do not notify TAPI.DLL of a new call yet. After the **Make Call** result has been received, the connection ID can be compared to the connection ID of the stored **Service initiated** event report. If they match, the **Service initiated** event report was a result of the **Make Call** request and can be ignored. Otherwise, TAPI.DLL is notified of a new call in the *dial-tone* state.

2. The **Delivered** event report is sent when the monitored device is ringing or when another device is ringing as a result of a call originated by the monitored device. In the former case, the delivery of an incoming call is signalled and the Service Provider should report a new call to TAPI.DLL with an *offering* call state. In the latter case, the delivery of an outbound call to a remote station is signalled and the Service Provider should report a new call state (*ringback*) for the call concerned.

3. The **Connection Cleared** and **Call Cleared** event reports are sent whenever a connection or call related to the monitored device is cleared. Depending on the device that caused this event, two different call state transitions should be reported. The Service Provider should report a transition to *idle* state if the event was caused by the local (monitored) device, or to *disconnected* state otherwise. The events may be sent as a result of a CSTA operation (**Clear Connection** or **Clear Call**). In this case, the Service Provider has already reported a transition to *idle* state and the event report can be ignored.

4. In the Windows Telephony specification, multiple call states are defined for calls that are on hold. Depending on the reason for the call to be on hold, its state can be *on hold, on hold pending transfer* or *on hold pending conference*. ECMA CSTA does not distinguish between the three. However, the Service Provider can keep track of which calls are involved in a transfer or a conference set-up. When a **Held** event report is received, the Service Provider can check whether the concerned call is involved in any previous set-up. Depending on the outcome of this check, the Service Provider can then report a call state transition to *on hold, on hold pending transfer* or *on hold pending conference.*

5. **Transferred** event reports are sent to both the device performing the transfer and the device being transferred. If a transfer has been performed, both calls involved in the transfer should transition to *idle* state as soon as the **Transferred** event report is received. Otherwise, the event report signals that a call that was previously being held (during a consultation call at the other station) has been transferred to another station. The call should transition to *connected* state.

6. In the Windows Telephony call model, a separate conference call object is used besides the call object(s) that represent the members of the conference call. With this model, that the conference call can be handled as a whole or as separate member calls. When a **Conferenced** event report is received, the member calls should transition to *conferenced* state, while the conference call transitions to *connected* state.

7. These event reports provide additional information about the progress of a call that the Windows Telephony system does not support. Therefore, they are mapped to the more general *proceeding* call state.

8. The **Failed** event report may be sent for all kinds of reasons. One of the reasons is a busy station or trunk, in which case the concerned call should transition to *busy* state. Other reasons are covered in the *special info* call state.

4.2.4 Data communication

The X.25/CSTA Service Provider is essentially an X.25 board driver with additional functionality. The X.25 board is not only intended for call control communication, but also for data communication with other computers (via the packet switch). The X.25/CSTA Service Provider should therefore also provide data communication facilities to the applications. Although TAPI can be used for the call control of data connections (i.e. X.25 Virtual Circuits), it does not support the data communication (*media stream*, refer to section 3.1.1) itself. Another API, preferably also part of a *Windows Open System Architecture* (WOSA) component, is required for data communication functions.

An interface that meets the demands has been developed very recently by Intel Architecture Labs and is called *Protocol Independent Interface* (PII) [9]. It defines a network programming interface for Microsoft Windows, based on the WOSA concept. PII is a superset of the *Windows Sockets*[16] (version

[16] *Windows Sockets is an API for (TCP/IP) network access. It is based on the Berkeley Sockets programming model, which is the de facto standard for TCP/IP networking.*

1.1) specification and has been submitted to the Windows Sockets Forum for possible inclusion in a future Windows Sockets version 2. Extensions to the current Windows Sockets interface include:

- *Protocol independence*. PII provides a standardised interface at the transport layer (level 4) of the OSI reference model. It can be used in conjunction with any number of transport protocols (e.g. TCP/IP, NetWare IPX/SPX, X.25/OSI, ISDN, ATM) by means of one or more Service Providers. Applications can specify the type of socket (communication endpoint) desired.
- *Expanded socket types*. Sockets contain communication properties such as connectedness, reliability, directionality and isochronicity.
- *Quality of Service*. PII introduces a number of features related to the negotiation of the Quality of Service. Changes in Quality of Service as a result of network condition changes will be reported to the application(s).
- *TAPI integration*. PII is designed to work hand-in-glove with the Windows Telephony API in providing uniform and consistent access to the transport capabilities of telephony connections. Applications can establish and utilise telephony data connections without making any explicit calls to the TAPI interface (combined TAPI/PII Service Providers will silently establish telephony connections on behalf of the application). Conversely, an application may use TAPI directly to control calls and then use PII to transport data over established calls.

These features, especially the latter, make the PII a very suitable data communication interface for use in combination with TAPI. Figure 22 depicts how applications can access the X.25/CSTA Service Provider (and other Service Providers) through TAPI.DLL and PII.DLL.

Datacom applications use PII to access the data communication facilities of any Datacom Service Provider, including the X.25/CSTA Service Provider, which is a combined Datacom/Telephony Service Provider. Since the PII is designed as a transport layer interface, the X.25/CSTA Service Provider should contain a (OSI) transport layer protocol on top of the X.25 protocols.

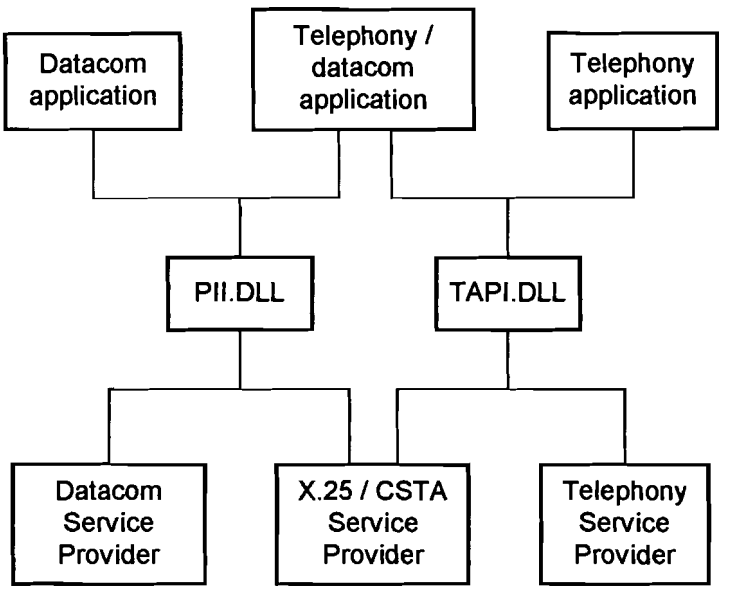


Figure 22. TAPI / PII co-operation

Telephony applications use TAPI to access the telephony facilities of any Telephony Service Provider, including the X.25/CSTA Service Provider.

Mixed telephony and datacom application that want to access the X.25/CSTA Service Providers can either use PII for all data communication operations (including control of the data calls) and TAPI exclusively for telephony call control, or they can use TAPI for all call control (including control of data calls) and PII exclusively for the transport of data over data calls established with TAPI.

4.2.5 Demo telephony application

The screen based telephony concept is demonstrated by implementing the following features:

- Buttons on the screen allow basic call control operations: originating outbound calls, answering incoming calls and dropping calls.
- Information concerning inbound and outbound call are presented in the form of a pop-up windows. Call information includes the state of the call and party identification.
- Party identification as reported by the switch consists of phone numbers. The telephony application offers *directory services*, i.e. it looks up the phone numbers in a phone directory and presents the phone numbers with associated names found in the phone directory.

The telephony application is responsible for the call control communication with the Windows Telephony system and for the call control user interface, i.e. presentation of call information and call control buttons. The phone directory will be implemented as a database in MS Access.

The telephony application performs the directory services by means of *Dynamic Data Exchange* (DDE). This is a standard Windows protocol for direct communication between applications. The DDE protocol is based on a client/server model. A DDE server application reports itself as such to the system. DDE client applications can inform themselves of the DDE server applications currently active. If the desired DDE server is active, the DDE client can establish a *conversation* with the DDE server. Note that an application may be both a DDE server and a DDE client. For the purpose of *directory services*, MS Access acts as a (SQL) DDE server and the telephony application as a DDE client.

The phone directory is also useful for originating calls. A "Dial" button on the screen can be used to dial the phone number in the currently selected record. In this case, MS Access has to send a *Make call* request (including the phone number) to the telephony application. This can be achieved by means of DDE (in this case the telephony application acts as a DDE server) or by means of the Assisted Telephony mechanism of the Windows Telephony system (refer to section 4.2.2).

For the demonstration of the multi-media call control concept, the application should take care of the creation and manipulation of voice/data calls:

- Upon establishment of an outbound voice call, the application should attempt to establish a data call to an associated data terminal.
- Upon receipt of an incoming data call, the application should attempt to associate the data call with an already received voice call.
- A voice connection and associated data connection should be handled as a unity (a single multi-media call).
- To demonstrate the use of the established multi-media (voice/data) call, information exchange between the parties should be possible via the clipboard. As soon as a party updates its clipboard as a result of a *cut* or *copy* operation, the contents will also be transmitted to the other party. The other party's clipboard is then automatically updated, as if the party had issued the *cut* or *copy* command itself. This party can use the *paste* command to view the information. At the same time, both parties can discuss things over the phone.

5. Results

5.1 Configuration

The demo configuration as shown in Figure 16 has been built in the demo room of Ericsson Telecommunicatie b.v. in Rijen. This configuration has been used to test and finally demonstrate the developed software. For this purpose, the demo configuration suffices.

For an operational configuration, the ApplicationLink protocol converter will have to be upgraded to support multiple Switched Virtual Circuits, instead of a single Permanent Virtual Circuit, in order to allow simultaneous call control communication with multiple computers.

The interconnection of PBX and packet switch in the present form is rather expensive, requiring a digital extension line interface in the PBX, a X.25 port in the packet switch and a TAU between the two. An alternative configuration is possible, where the PBX is equipped with X.21 interface boards. One such an interface board contains six X.21 interfaces for direct connections (i.e. without TAU) to the packet switch. A trunk interface (e.g. an ISDN PRI) between the two would be even a better solution.

If then PBX's extension lines are executed as standard ISDN Basic Rate Interfaces (which will eventually happen), the TAUs are no longer needed. In this case, the computers will have to be equipped with ISDN boards instead of X.25 boards. Both telephone and computer can then be connected to the ISDN S-bus in parallel.

5.2 Software

Implementation of the X.25/CSTA Service Provider has resulted in a total of approximately 3900 lines of C++ source code consisting of an X.25 interface module[17] (570 lines), an ASN.1 / BER implementation module (950 lines), an ECMA CSTA protocol implementation module (750 lines), a mapping module (840 lines) and a Telephony Service Provider Interface module (750 lines).

Implementation of the demo telephony application has resulted in 730 lines of C++ source code.

5.2.1 ECMA CSTA implementation

Using the ASN.1/BER module, the ROSE protocol and a large part of the ECMA CSTA protocol has been implemented:

- Common elements such as:
  - device identifiers
  - connection identifiers
  - call identifiers
  - error codes

[17] *Direct interfacing to the X.25 driver software from within the Service Provider DLL appeared impossible. Therefore, a separate background application that takes care of the X.25 interface is automatically started by the Service Provider DLL. The Service Provider communicates with this application by means of the Windows messaging mechanism (windows procedure).*

- The following CSTA Switching Function Services:
  - Monitor Start
  - Monitor Stop
  - Make Call
  - Answer Call
  - Consultation Call
  - Transfer Call
  - Retrieve Call
  - Clear Connection
- All call event reports defined in ECMA CSTA

Implementation of the entire ECMA CSTA protocol is possible with little extra effort, but the current implementation is sufficient to support the basic call control operations required for the demo system. All call event reports have been implemented in order to avoid errors caused by reception of unknown event reports, since these call event reports occur unsolicited. Not all event reports are processed by the rest of the software, though.

Implementation of the ROSE and ECMA CSTA protocol (both specified in ASN.1) by means of the ASN.1/BER module is very straight-forward and allows for easy integration of the protocols and the rest of the software. Since the ASN.1/BER module performs actions that are part of the duty of the presentation layer (encoding/decoding), difficulties may arise if a full OSI stack (i.e. including layers 4 to 6) is implemented, however.

5.2.2 Mapping

In order to allow for uniform call control of voice calls (CSTA) and data calls (X.25), the Service Provider supports voice line devices and data line devices. A voice line device represents a phone device connected to the PBX; the data line device represents the computer connected to the packet switch. The user can configure zero, one or more voice line devices and one data line device. Configuration info for each line device consists of the name and the address of the line device. In case of a voice line device, the address is the phone number of the associated phone device; in case of the data line device, the address is the X.25 address of the computer.

As soon as a session is started on a voice line device (the function **TSPI_lineOpen** is called), a monitor is requested on the associated phone device by sending a **Monitor Start** request to the ApplicationLink. Likewise, when the session is closed (function **TSPI_lineClose**), the monitor is cancelled by sending a **Monitor Stop** request to the ApplicationLink. The functions **TSPI_lineOpen** and **TSPI_lineClose** are specified as *synchronous* functions in the Windows Telephony specification. Therefore, the result of the **Monitor Start** and **Monitor Stop** requests must be awaited within these functions, resulting in the calling process being suspended during this period.

Other call control functions implemented in the Service Provider are: **TSPI_lineMakeCall**, **TSPI_lineAnswerCall** and **TSPI_lineCloseCall**. The way these functions are processed depends on whether they are invoked on a voice or data line device:

- If invoked on a voice line device, these functions send an appropriate CSTA APDU (which is transmitted inside an X.25 data packet) to the ApplicationLink, resulting in the associated phone device being controlled.
- If invoked on the data line device, these functions send X.25 call control packets to the packet switch, resulting in X.25 Virtual Circuits being controlled.

Table 6. Implemented call control function mapping

TSPI function:	Voice / Data:	CSTA APDU:	X.25 packet type:
TSPI_lineOpen	V	Monitor Start	DATA
	D		
TSPI_lineClose	V	Monitor Stop	DATA
	D		
TSPI_lineMakeCall	V	Make Call	DATA
	D		CALL REQUEST
TSPI_lineAnswer	V	Answer Call	DATA
	D		CALL ACCEPTED
TSPI_lineCloseCall	V	Clear Connection	DATA
	D		CLEAR REQUEST

This limited call control function implementation is sufficient to demonstrate the concepts of screen based telephony and multi-media call control. Implementation of many other functions for additional call control functionality will be possible with little effort. Implementation of functions to manipulate consultation calls and conference calls, however, may present some difficulties in the mapping, due to the different approaches for modelling these calls in ECMA CSTA and the Windows Telephony system. Typical third-party call control features like call routing and ACD agent support cannot be implemented, since these are not supported in the current version of the Windows Telephony system.

The implemented mapping between received CSTA call event reports and call state transitions reported to TAPI.DLL (and eventually to the application) is listed in Table 7 (see also section 4.2.3). Upon reception of a **Delivered** event, the state of the local CSTA connection object is inspected to determine whether the event indicates an incoming call (in which case a new call with an *offering* call state will be reported), or the delivery of an outbound call to the remote station. In the latter case, a new call is reported, unless the event is a result of an earlier **Make Call** operation (this is verified by comparing the connection identifiers).

Table 7. Implemented CSTA event report mapping

CSTA call event report received:	New Windows Telephony call state:
Delivered	*offering / ringback*
Established	*connected*
Connection Cleared	*idle / disconnected*
Held	*on hold*
Retrieved	*connected*
Failed	*busy*

For data calls, an incoming CALL REQUEST packet results is a new (data) call with an *offering* call state being reported. An application that wants to accept the call must call the function **lineAnswer**, resulting in a CALL ACCEPTED packet being sent. An incoming CLEAR REQUEST packet is immediately replied to with an CLEAR CONFIRMATION packet and results in a call state transition to *disconnected*.

5.2.3 Data communication

The proposed use of Intel's *Protocol Independent Interface* for data communication purposes is not realised in the demo software, because a PII developers kits was not available yet at the moment of demo system development. Instead, some of the extension mechanisms defined in the Windows Telephony system have been used for data communication purposes. This should be considered as a temporary solution, since it does not correspond to the philosophy behind the Windows Telephony system.

5.2.4 Demo telephony application

For demonstration of the screen based telephony and multi-media call control concepts, the following functionality is implemented in the demo telephony application:

- As soon as the application is informed of a new call (either inbound or outbound), a pop-up windows is displayed that presents information concerning the call. Call information includes the state of the call (ringing, connected, on hold, busy) and party identification (calling party ID, called party ID, connected party ID).
- The application offers *directory services*. For this purpose, a phone directory database in MS Access is consulted by means of *Dynamic Data Exchange* (DDE).
- The call-related pop-up windows contain buttons for call control (answering and dropping the call)
- The application can originate a call on behalf of other applications by means of DDE. In this manner, MS Access is provided with a "Dial" button that can be used to automatically dial the phonenumber in the selected record.
- Upon establishment of an outbound voice call (as a result of either a CSTA action of a user action at the phone-set), the application attempts to establish a data call to an associated data terminal (the X.25 address is found in the phone directory).
- Upon receipt of an incoming data call, the application attempts to associate the data call with an already received voice call by comparing the caller's X.25 addresses of the data call and the voice call (for the voice call, the caller's X.25 address is found in the phone directory).
- A voice connection and associated data connection are handled as a single multi-media call. More specifically, as soon as either of the two is dropped, the multi-media as a whole is dropped. Pressing a "Drop" button on the screen results in the multi-media call (both connections) being dropped.
- If a data connection is established, information can be exchanged via the clipboard. As soon as a party updates its clipboard, the contents will also be transmitted to the clipboard of the other party.

The way the demo telephony application presents itself to the user is illustrated in Figure 23. Two examples are depicted of pop-up windows that the demo telephony application displays as soon as it is notified of a new outbound or inbound voice call, respectively.

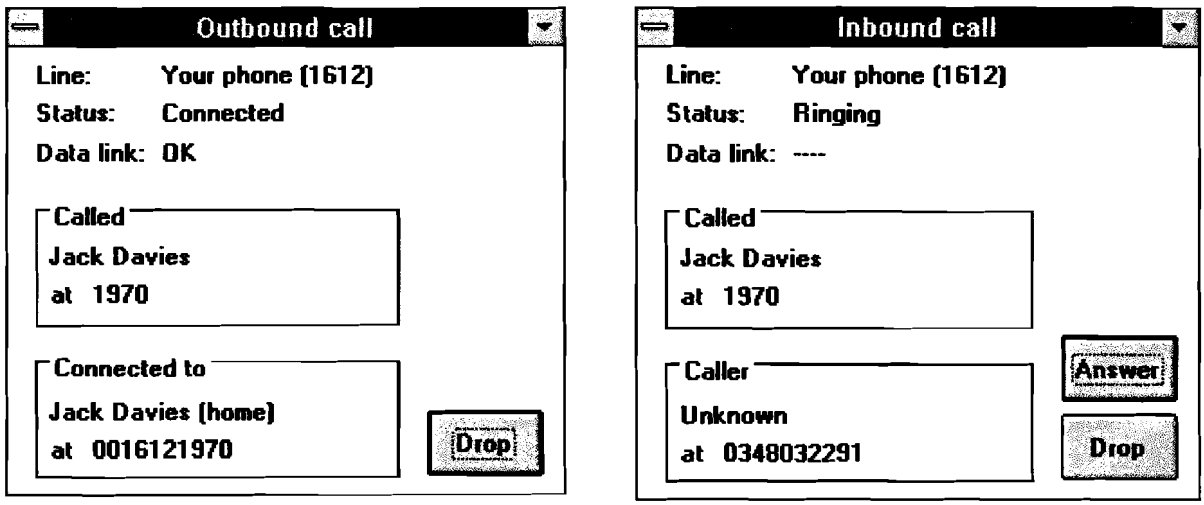


Figure 23. Pop-up window examples

Both windows display the name of the voice line device the call is related to, the status of the voice connection and the status of the associated data connection. The information in the boxes is the party identification and consists of the phonenumber (reported by the switch) and the associated name found in the phone directory. If a phonenumber is not found in the phone directory, "Unknown" is displayed instead of the name. Besides information about the call, the windows also contain buttons for call control.

In case of the outbound call in Figure 23, the switch reported that the number 1970 was called from your phone (number 1612). The demo telephony application looked up the number 1970 in the phone directory and found the name "Jack Davies". However, Jack Davies has apparently activated call forwarding to another number, since the switch reported that the call was connected to number 0016121970, instead of 1970. The telephony application also looked up this number, and found the name "Jack Davies (home)". Jack has apparently answered the phone: the status of the call is *connected*. The demo telephony application has also succeeded in establishing a data connection to Jack's computer at home: the data link is OK. At this time, the contents of your clipboard will automatically be transmitted to Jack's computer and vice versa.

In case of the inbound call in Figure 23, the switch reported that a call arrived at your phone. The phonenumber of the caller was reported by the switch, but the name of the caller couldn't be found in the phone directory. The caller actually tried to reach Jack Davies (the switch reported that the number 1970 was called), but Jack Davies has again activated call forwarding, this time to your phone, which is the reason your phone is *ringing* at the moment.

6. Conclusions

Two de-facto standards for computer supported call control have emerged: Microsoft/Intel's Windows Telephony system (better known as Telephony API or TAPI), which is an open standard for first-party call control purposes, and Novell/AT&T's NetWare Telephony Services (also known as TSAPI) that provides third-party call control functionality in a PC-LAN environment.

Although the number products that support the Windows Telephony standard is negligible at the moment, a large number of manufacturers have announced Windows Telephony support for their products. For a manufacturer of PC-based telephony hardware, supporting the Windows Telephony standard implies instant access to a group of 40 million Windows users world-wide.

To provide third-party call control functionality, the NetWare Telephony Services configuration employs a direct link to the PBX, called CTI link. The Telephony Services API (TSAPI) is based on the ECMA CSTA standard, a prominent standard protocol for third-party call control communication between computers and switches. Although the fixed costs (for the NetWare Telephony Services software licence, the PBX driver software and the CTI link hardware) are rather high, the configuration is profitable if a large number of workstations is involved, since the workstations don't need additional telephony hardware.

A CTI demo system has been developed that combines the Windows Telephony system with a third-party call control configuration like the NetWare Telephony Services configuration. Unlike the NetWare Telephony Services configuration, no LAN is required. This implies that the configuration could also be used in (ISDN) Centrex situations. Furthermore, remote access is possible, opening the door for teleworking.

Although the Windows Telephony system is intended for first-party call control purposes, it offers enough functionality to allow a mapping between TAPI and the ECMA CSTA protocol, except for the typical third-party call control features found in ECMA CSTA. A mapping between TAPI and ECMA CSTA for basic call control operations has been developed without problems. Many other functions for call control may be implemented with little effort.

The CTI demo system development also showed that programming telephony applications using the Windows Telephony API (TAPI) is very easy. The demo telephony application was developed within one week time (approximately 50 man-hours).

References

[1] ECMA Standard 179
Services for Computer-Supported Telecommunications Applications (CSTA)
Geneva: ECMA, june 1992

[2] ECMA Standard 180
Protocol for Computer-Supported Telecommunications Application (CSTA)
Geneva: ECMA, june 1992

[3] Gabriël, C.M.W. et al
Heading towards an advanced signalling system for multimedia, multiparty services in B-ISDN
in: PTT Research review, vol. 4, no. 3, august 1994, pp.41-58

[4] Gray, M.
Voice+ CTI Survey
in: Voice+, vol.1, no. 4, sept./oct. 1994, pp. 58-76

[5] Greatz, I. and Müller, H.
IN and CSTA - two sides of the same coin ?
in: Telcom report international, no. 15, 1992, pp. 42-46

[6] Grinberg, A.
Computer/Telecom Integration - The SCAI solution
New York: McGraw-Hill, 1995, ISBN 0-07-024842-7

[7] Harman, W.M. and Newman, C.F.
ISDN protocols for connection control
in: IEEE journ. on sel. areas in communications, vol. 7, no. 7, september 1989, pp. 1034-1042

[8] Ihlow, O.
Computergestützte Telekommunikation: Technologie, Einführung und Anwendungen
in: Nachrichten Elektronik & Telematik (NET), vol. 45, september 1991, pp. 349-353

[9] Intel
Protocol Independent Interface
revision 1.5, september 1994

[10] ITU-T Recommendation Q.1201
Principles of Intelligent Network architecture
ITU, october 1992

[11] ITU-T Recommendation X.31
Support of packet mode terminal equipment by an ISDN
ITU, march 1993

[12] ITU-T Recommendation X.208
Specification of Abstract Syntax Notation 1
ITU, Blue book, 1988

[13] ITU-T Recommendation X.209
Specification of Basic Encoding Rules for the abstract syntax notation
ITU, Blue book, 1988

[14] ITU-T Recommendation X.219
Remote operations - model notation and service definition
ITU, Blue book, 1988

[15] ITU-T Recommendation X.229
Remote operations - protocol specification
ITU, Blue book, 1988

[16] Microsoft / Intel
Microsoft Windows Telephony programmers guide
version 1.0, 1993

[17] Novell / AT&T
Telephony Services Application Programming Interface (TSAPI)
issue 1.9, 1994

[18] Peeters, M.C.M.
How to exchange multimedia information in its natural form
in: PTT Research review, vol. 4, no. 3, august 1994, pp. 27-40

[19] Rehin, A.
Integrated voice applications: implications for users
in: Telecommunications (international), vol. 26, february 1992, pp. 21-28

[20] Slechte telefonische bereikbaarheid berokkent veel schade
in: Bedrijfscommunicatie, vol. 3, no. 4, 1994, pp. 11-13

[21] Stagg, L.J.
Integrating the computer and telecommunications
in: Proc. of IEE conference on private switching, publication #357, 1992, pp. 191-198

[22] Trought, M.
Private networking: a roadmap to the future
in: Telecommunications (international), vol. 25, september 1991, pp. 39-46

[23] Udell, J.
Computer telephony
in: Byte, vol. 19, no. 7, july 1994, pp. 80-96

[24] Wilson, N.
PC players set the CTI standard
in: Telecommunications (international), vol. 28, no. 10, october 1994, pp. 87-91

Appendix A. CTI link survey

Table 6. CTI link overview

Manufacturer and product name	Interfaces					Transport protocols				Call control
	RS-232	RS-422	Ethernet	Token Ring	ISDN BRI	X.25	TCP / IP	LU6.2	ISDN DSS1	ECMA CSTA
Aspect CallCenter	X		X	X			X	X		X
Co-Cam SoftCall	X		X				X			+
D.T.S. Harris 20-20	X		X				X			+
Ericsson ACP1000		X				X				X
Ericsson MD110	X					X				X
GPT iSDX			X	X			X			
Melita Phoneframe			X	X			X	X		
Mitel SX-2000, Supervisor ACD	X		X	X		X	X			X
Philips SOPHO, iS3000 series					X				X	X
SDX 60N, SDX 420N	X									X
Siemens Hicom 300	X				X					
STS Supercall 2000	X		X				X			
Northern Telecom Meridian 1	X		X			X		X		+
Unisys Summa Four, SDS1000, Unaswitch	X									
Rockwell Spectrum	X		X			X	X			X

Legenda:
X = supported
+ = support planned

Appendix B. CSTA Switching Function Services

This appendix describes each CSTA Switching Function Service that affects the connection state at one or more devices involved in a call. Changes in connection states are depicted in a figure, which shows the connection states before and after the operation. These figures are based on the CSTA call model. Boxes represent CSTA device objects (greyed boxes represent devices that are unaffected by the service), circles represent CSTA call objects and lines represent CSTA connection objects. The state of each connection is indicated as a single letter, as shown in Table 9.

Table 9. Legenda

Symbol	Connection state
a	Alerting
c	Connected
f	Failed
h	Held
i	Initiated
q	Queued
*	Undefined

Alternate Call

The Alternate Call Service places the user's active call to device D2 on hold and, in a combined action, establishes or retrieves the call between device D1 and device D3 as the active call. Device D2 can be considered as being automatically placed on hold immediately prior to the retrieval/establishment of the held/active call to device D3.

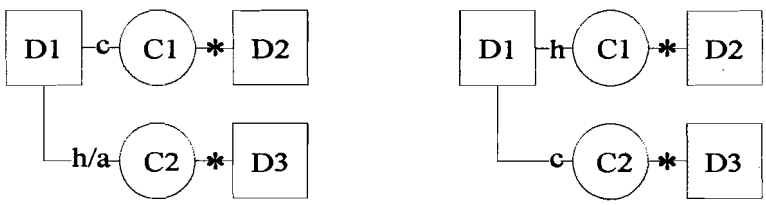


Figure 1 - Alternate Call

Answer Call

The Answer Call Service works for an incoming call that is alerting a device. In the figure the call C1 is delivered to device D1. The service is typically used with telephones that have attached speakerphone units to establish the call in a hands-free operation.

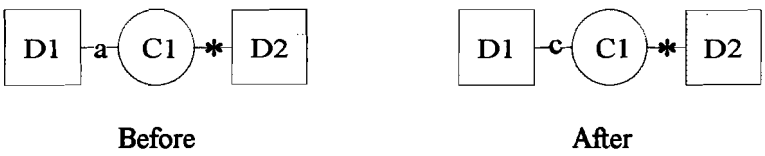


Figure 2 - Answer Call

Clear Call

The Clear Call Service will cause each device associated with a call to be released and the CSTA Connection Identifiers (and their components) are freed. The figure illustrates the results of a Clear Call (CSTA Connection ID = C1,D1), where call C1 connects devices D1, D2 and D3.

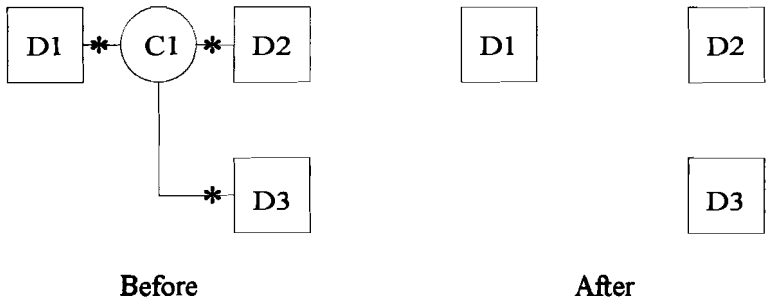


Figure 3 - Clear Call

Clear Connection

The Clear Connection Service releases the specified Connection and CSTA Connection Identifier instance from the designated call. The result is as if the device had hung up on the call. It is interesting to note that the phone may not be physically returned to the switch hook, which may result in silence, dial tone, or some other condition. Generally, if only two Connections are in the call, the effect of this service is the same as the Clear Call service. The figure depicts an example of the results of a Clear Connection (CSTA Connection Id = C1,D3), where call C1 connects devices D1, D2 and D3. Note that it is likely that the call is not cleared by this Service if it is some type of conference.

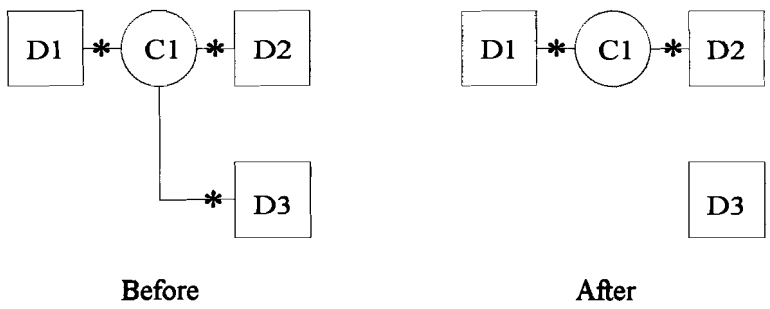


Figure 4 - Clear Connection

Conference Call

The figure is an example of the starting conditions for the Conference Call Service, which are: the call C1 from D1 to D2 is in the held state. A call C2 from D1 to D3 is in progress or active. D1, D2 and D3 are conferenced or joined together into a single call, C3. The value of the Connection identifier (D1,C3) may be that of one of the CSTA Connection Identifiers provided in the request (D1,C1 or D1,C2).

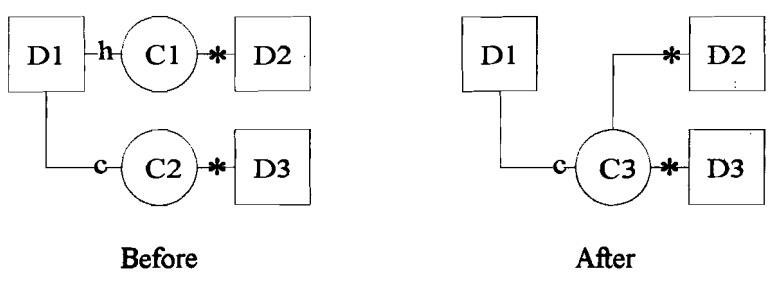
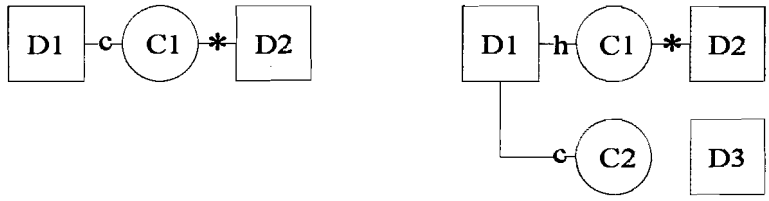


Figure 5 - Conference Call

Consultation Call

This compound service allows the application to place an existing call on hold and at the same time establish a new call to another device. In this case an active call C1 exists at D1 and a consultative call is desired to D3. After this function is called, the original active call (C1) is placed on hold and a new call, C2, is placed to device D3.



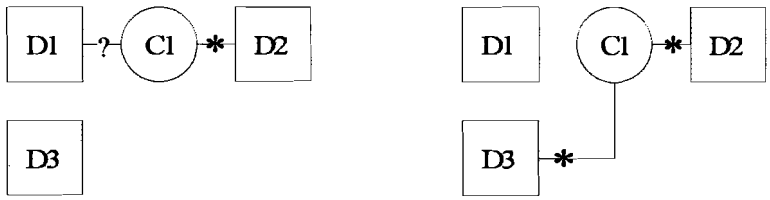
Before After

Figure 6 - Consultation Call

Divert Call

The Divert Call Service replaces the original called device with a different called device. The Divert Call Service supports at least three common call diversion services:

1. Deflection. Takes a ringing call at a device and sends it to a new destination.
2. Pickup. Takes a ringing call at another device and brings it to the device concerned.
3. Group pickup. Takes a ringing call at one or more predetermined device(s) and brings it to the device concerned.



Before After

Figure 7 - Divert Call

Hold Call

The Hold Call Service will interrupt communications for an existing call at a device. The call is usually, but not always, in the active state. A call may be placed on hold by the user some time after completion of dialing. The associated connection for the held call is made available for other uses, depending on the reservation option (ISDN-case). As shown in Figure 5.11, if the Hold Call service is invoked for device D1 on call C1, then call C1 is placed on hold at device D1. The hold relationship is affected at the holding device.

Before After

Figure 8 - Hold Call

Make Call

The Make Call Service originates a call between two application designated devices. When the service is initiated, the calling device is prompted (if necessary), and, when that device acknowledges, a call to the called device is originated. The figure illustrates the results of a Make Call service request (Calling device = D1, Called device = D2). A call is established as if D1 had called D2, and the client is returned the Connection id: (C1,D1).

Figure 9 - Make Call

Make Predictive Call

The Make Predictive Call Service first initiates a call to the called device (destination). Depending on the call's progress, the call may be connected with the calling device (originator) during the progress of the call. The point at which the switch will attempt to connect the call to the originating device is determined by the ***allocation*** parameter. If the allocation parameter is set to Call Delivered, then the call is allocated upon detection of an Alerting (or Connected) Connection state at the destination. If the allocation parameter is set to Call Established, then the call is allocated upon detection of a Connected Connection state at the recipient. The figure illustrates the results of a Make Predictive Call (Calling device = group device D1, Called device = D2.

Figure 10 - Make Predictive Call

Reconnect Call

A successful request of this service will causes an existing active call to be dropped. Once the active call has been dropped, the specified held call at the device is retrieved and becomes active. This service is typically used to drop an active call and return to a held call; however, it can also be used to cancel of a consultation call (because of no answer, called device busy, etc.) followed by returning to a held call.

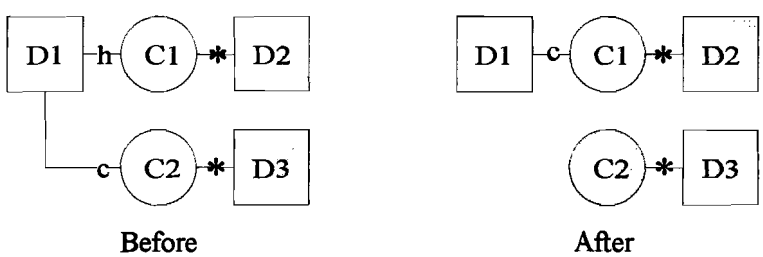


Figure 11 - Reconnect Call

Retrieve Call

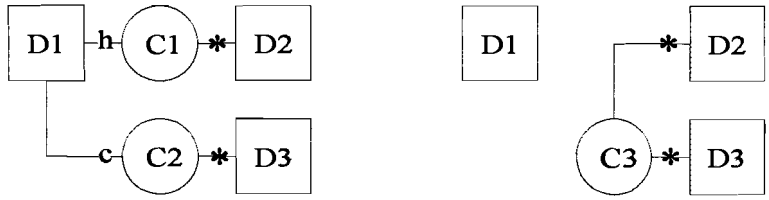
The indicated held Connection is restored to the Connected state (active). The call state can change depending on the actions of far end endpoints. If the Hold Call service reserved the Held Connection and the Retrieve Call service is requested for the same call, then the Retrieve Call service uses the reserved Connection.

Before After

Figure 12 - Retrieve Call

Transfer Call

Referring to the figure, the starting conditions for the Transfer Call service are: the call C1 from D1 to D2 is in held state. A call C2 from D1 to D3 is in progress or active. This service transfers the existing (held) call between devices D1 and D2 into a new call with a new call identifier from device D2 to a device D3. The service is used in the situation where the call from D1 to D3 is established (active) or if the call is in any state other than Failed or Null state. The Transfer Call service successfully completes, and D1 is released from the call.



Before After

Figure 13 - Transfer Call

Appendix C. CSTA / TAPI functionality comparison

Table 10. ECMA CSTA & TAPI call control functionality

Category:	Function:	ECMA CSTA	TAPI
Basic telephony	Originate call	Make Call	lineMakeCall
	Answer call	Answer Call	lineAnswer lineAccept (ISDN)
	Drop call	Clear Connection Clear Call	lineDrop lineDrop
Call hold	Hold call	Hold Call	lineHold
	Retrieve held call	Retrieve Call	lineUnhold
	Swap held & active	Alternate Call	lineSwapHold
Call park	Park call		linePark
	Unpark call		lineUnPark
Consultation call	Make consultation call	Consultation Call	lineSetupTransfer
	Reconnect old call	Reconnect Call	lineDrop
	Transfer call	Transfer Call	lineCompleteTransfer
	Conference call	Conference Call	lineCompleteTransfer lineAddToConference
Call divertion	Deflect call	Divert Call	lineRedirect
	Pickup call	Divert Call	linePickup
	Pickup group call	Divert Call	linePickup
Call completion	Camp on	Call Completion	lineCompleteCall
	Call back when free	Call Completion	lineCompleteCall
	Intrude	Call Completion	lineCompleteCall
	Leave a message		lineCompleteCall
Call forwarding	... always	Set Features	lineForward
	... on busy	Set Features	lineForward
	... on no answer	Set Features	lineForward
	Do-not-disturb	Set Features	lineForward
User-to-user signalling	Out-of-band		lineSendUserUserInfo
	Inband (DTMF)		lineGenerateDigits lineMonitorDigits lineGenerateTone lineMonitorTones
Typical third-party call control features	Make predictive call	Make Predictive Call	
	Call routing	Re-Route Route End Route Request Route Select Route Used	
	ACD agent support	Set Features Query Device	

Appendix D. ASN.1 / BER implementation module

The C++ source module "ASN1BER.CPP" and its header file "ASN1BER.H" contain class definitions for direct implementation of ASN.1 data types and associated Basic Encoding Rules (BER) in an C++ module, i.e. no ASN.1 compiler is required. These **A**SN.1 / **B**ER **C**lass **D**efinitions are shortly referred to as ABCD. This appendix briefly describes important ASN.1 features and the way they are implemented in ABCD.

Extended BNF used in this text:

< >	Text inside brackets is a place-holder for the name of an identifier
" "	Text inside double quotes is literal
[ ]	Items inside brackets are optional
...	Ellipses after an item indicate that the item may be repeated
,,,	Commas indicate that the item may be in a list, where the items are separated by commas.
|	A bar indicates a choice between items
::=	Assignment

Table 11. ASN.1 Built-in types (universal class tags)

Tag value:	ASN.1 type:	ABCD class name:
1	Boolean	BOOLEAN
2	Integer	INTEGER
3	Bit string	BITSTRING
4	Octet string	OCTETSTRING
5	Null	Null
6	Object identifier	
7	Object descriptor	
8	External	
9	Real	
10	Enumerated	ENUMERATED
16	Sequence (of)	SEQUENCE(_OF)
17	Set (of)	
18	NumericString	
19	PrintableString	
20	TeletexString (T61String)	
21	VideotexString	
22	IA5String	IA5String
23	GeneralizedTime	
24	UTCTime	
25	GraphicString	
26	VisibleString (ISO646String)	
27	GeneralString	
--	Choice	CHOICE
--	Any	ANY

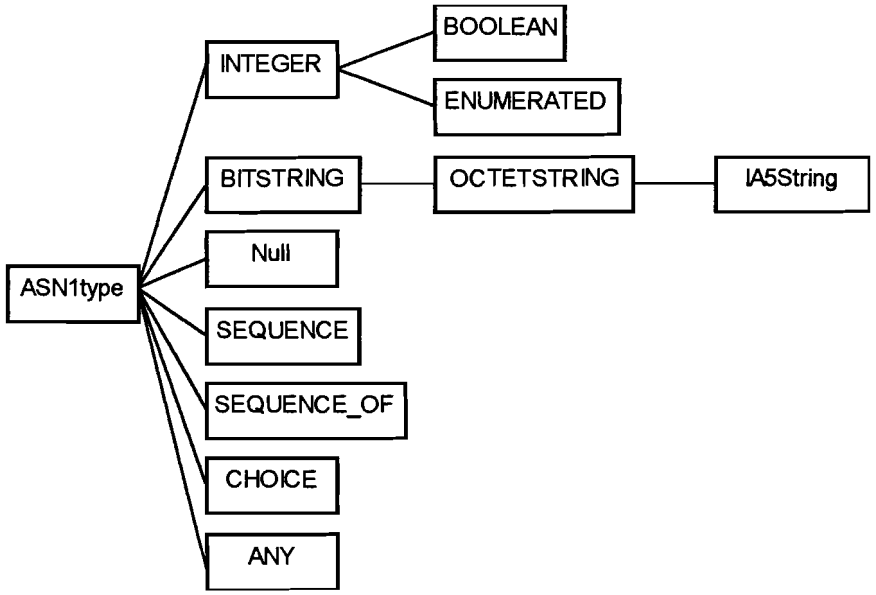


Figure 24. ABCD class hierarchy

Tags:

An ASN.1 variable is characterised by its data type and its value. In order to allow the receiving side to reconstruct a transmitted variable, both data type and value should be transmitted. The data type is encoded as a so-called tag field. Every ASN.1 type[18] has an associated tag value and tag class, shortly referred to as tag. There are four tag classes:

1. *Universal.* Only assigned within the ASN.1 recommendation.
2. *Application.* Only assigned within other recommendations and standards.
3. *Private.* Enterprise specific assignments.
4. *Context-specific.* Context specific assignments.

All built-in ASN.1 types (defined in ITU-T recommendation X.208) have *universal* class tags. Table 11 lists these built-in types, their tag values and the corresponding ABCD class names. ASN.1 offers the possibility to create a new type that is similar to another type, but with a different tag. The new type is said to be a tagged version of the other type. Two alternative ways of tagging are defined:

1. *Implicit.* This way the original tag is replaced with a new tag. The original data type can only be determined with knowledge of the new tag.
2. *Explicit.* This way the original tag is kept and the new tag is added. The original data type can be determined without knowledge of the new tag.

The keywords IMPLICIT and EXPLICIT are used to indicate which of the alternatives should be used. If neither IMPLICIT nor EXPLICIT is specified, tagging defaults to explicit. The keywords UNIVERSAL, APPLICATION and PRIVATE are used to indicate which tag class is to be used. Absence of any of these keywords implies the context-specific tag class. The complete syntax of an ASN.1 definition of a tagged type is (in BNF):

<TaggedType> ::= "[" [<Class>] <TagValue> "]" ["IMPLICIT" | "EXPLICIT"] <Type>
<Class> ::= "UNIVERSAL" | "APPLICATION" | "PRIVATE"

[18] *exceptions are "choice" and "any" types*

ABCD allows both implicit and explicit tagging by means of class constructors. A constructor function is a special class member function that is automatically called at the moment the class is instantiated, i.e. an object of that class is created. Likewise, the destructor function is called at the moment an object is disposed. Each ABCD class has a constructor function that can take zero, one or two arguments:

- Without arguments, no tagging is performed (i.e. the original tag is kept).
- If one argument is specified, the argument is expected to be a byte that contains both the tag value and the tag class. Tag values should be smaller than 32 and default to context-specific tag class. For other tag classes, the value can be "or"-ed with one of the constants UNIVERSAL, APPLICATION or PRIVATE. Explicit tagging is performed.
- If two arguments are specified, the first is expected to be a byte that contains both the tag value and the tag class. The second argument is expected to be one of the keywords EXPLICIT or IMPLICIT and indicates the way of tagging.

Examples:

ASN.1	ABCD

`IntVar :: = INTEGER`
`ExpTagInt0 ::= [0] INTEGER`
`ImpTagInt1 ::= [1] IMPLICIT INTEGER`
`ExpTagInt2 ::= [PRIVATE 2] INTEGER`
	`INTEGER IntVar;`
	`INTEGER ExpTagInt0(0);`
	`INTEGER ImpTagInt1(1,IMPLICIT);`
	`INTEGER ExpTagInt2(PRIVATE|2);`
	`INTEGER* ExpTagInt1Ptr = new INTEGER(1);`

ABCD classes and objects:

ABCD is a C++ based class hierarchy that contains classes for the most frequently used ASN.1 built-in types (see Table 11 and Figure 24). The base class is called **ASN1type** and contains virtual member functions for value assignment, value retrieval, BER encoding and decoding, etc. This means that a pointer to the **ASN1type** class can be used as a reference to any ABCD class. The compiler automatically uses the member functions of the ABCD class the pointer really points to.

An ABCD object is an instance of an ABCD class and is similar to an ASN.1 variable. Note that an ABCD object has three characteristics: class, tag and value. When an ABCD object is tagged, its tag field is changed, but its class isn't.

An ABCD class must be instantiated in order to be able to change its tag (the tag is actually a data member of the class; it is initialised with a default value). More complex ABCD classes like SEQUENCE and CHOICE must be instantiated before any items can be specified. ABCD classes therefore aren't really similar to ASN.1 types, since they don't contain all the information ASN.1 types can contain. ABCD objects, however, do contain all the ASN.1 type information (plus a value).

Since ABCD objects are more similar to ASN.1 types that ABCD classes, it should be possible to handle ABCD objects like ASN.1 types (ignoring their value). More specifically, it should be possible to use an ABCD object in the specification of another ABCD object. This is implemented by the function call operator ( ). When using this operator on an ABCD object, it returns a pointer to a duplicate of that object, i.e. an object with the same class and tag (the object's value is not copied).

Example:

```
INTEGER Int1(10,IMPLICIT); // New INTEGER type with context-specific tag
INTEGER* Int2Ptr = Int1(); // Int2Ptr points to a [10] IMPLICIT INTEGER
INTEGER* Int3Ptr = new INTEGER(10,IMPLICIT) ; // dito
```

Value assignment and retrieval:

Once an ABCD object has been created, it can be used as an ASN.1 variable. Therefore, it should be possible to assign a value to it and to retrieve its value. This is implemented by the operator = and the member function **get**, respectively.

Example:

```
int C_int_var;
INTEGER Int1(PRIVATE|1);        // In ASN.1:  Int1 ::= [PRIVATE 1] INTEGER
Int1 = 123;                     // In ASN.1:  Int1 ::= 123
Int1.get(C_int_var);            // Retrieves the value of Int1 in C_int_var
```

Which C data type should be used for value assignment and retrieval depends upon the class of the ABCD object. The C data types that can be used are called compatible types and are specified for each ABCD class.

Comparing and copying ABCD objects:

Any two ABCD objects of the same class can be compared by means of the operator = =. The result of this operation is TRUE if the tags and the values of both objects correspond. With more complex ABCD objects, consisting of a collection of other ABCD objects, this also means that each sub-object's tag and value should correspond.

The value of an ABCD object can be copied to another ABCD object of the same class and tag, by means of the operator =.

Examples:

```
INTEGER Int1(0);
INTEGER Int2(0);
INTEGER Int3(0);
INTEGER Int4(1);
Int1 = 123 ;
Int2 = 123 ;
Int3 = 234 ;
Int4 = 123 ;
if (Int1 == Int2) // Yes.
if (Int1 == Int3) // No, the values are not the same
if (Int1 == Int4) // No, the tags differ
Int3 = Int1 ; // Now the values are the same !
Int4 = Int1 ; // This is illegal, since the tags differ
```

BER encoding and decoding:

Each ABCD object can encode itself (using the Basic Encoding Rules defined in ITU-T recommendation X.209) by means of its member function **BER_encode**. This member function takes a pointer to a buffer location as an argument. After encoding, it updates this pointer to point to the first byte after the encoded value. The function returns the size of the encoded value.

Each ABCD object can be requested to decode an encoded byte-stream by means of its member function **BER_decode**. This member function takes a pointer to the encoded byte-stream as an argument. The object first determines if it is able to decode the byte-stream by comparing its tag with the encoded tag field in the byte-stream. If the tags differ, the function returns 0 (int). If the tags match, the ABCD object's value is decoded from the byte-stream and the pointer is updated to point to the first byte after the decoded value; the function returns the number of bytes decoded.

Output stream insertion:

Each ABCD object can be inserted in an output stream by means of the standard output stream operator <<. The result of this operation is a human-readable output of the object's value. The notation used is similar to the standard notation for ASN.1 values, described in ITU-T recommendation X.208. This feature is very useful for debugging purposes.

ASN.1 built-in types and ABCD classes:

Legenda:

<cstring> is a character string enclosed in double quotes. E.g. "Test"
<bstring> is a character string enclosed in quotes and with a "B" suffix, that represents a string of binary digits (bits). E.g. '01010101'B
<hstring> is a character string enclosed in quotes and with a "H" suffix, that represents a string of hexadecimal digits (nibbles). E.g. '10AF'H
<identifier> is a character string that represents a name. The first character should be lowercase.

<NamedNumber> ::= <identifier> "(" <number> ")"
<NamedBit> ::= <identifier> "(" <number> ")"
<NamedValue> ::= [<identifier>] <value>
<NamedType> ::= [<identifier>] <type>

1. Boolean

ASN.1 Boolean variables can assume two logic states: true or false.

<BooleanType> ::= "BOOLEAN"
<BooleanValue> ::= "TRUE" | "FALSE"

ABCD class name: BOOLEAN
Compatible types: BOOL (int)

Since BOOL is defined as int (in windows.h), there is really no difference between assigning a Boolean or an integer value, as long as the interpretation of the values is respected: a value of zero is interpreted as FALSE, any non-zero value as TRUE. In order to avoid confusion, it's better to use the constants TRUE and FALSE, defined in windows.h.

2. Integer

ASN.1 Integer variables can hold all cardinal values, unlimited in magnitude. The exact range can be specified using subtypes. Optionally, some values can be assigned an identifier. This identifier may be used in value assignments.

<IntegerType> ::= "INTEGER" ["{" <NamedNumber>,,, "}"]
<IntegerValue> ::= <number> | <identifier>

ABCD class name: INTEGER
Compatible types: int, long

Assigning identifiers to values is not implemented in the class INTEGER. However, this can easily be done by means of **enum** or **#define**.

3. Bit string

ASN.1 Bit string variables carry strings of bits of arbitrary length. The individual bits (least significant bit is bit 0) may be referenced by associated names.

<BitStringType> ::= "BIT STRING" ["{" <NamedBit>,,, "}"]
<BitStringValue> ::= <bstring> | <hstring> | "{" [<identifier> ,,,] "}"

ABCD class name: BITSTRING
Compatible types: BYTE, UINT, DWORD (char, int, long)

Assigning identifiers to bits is not implemented in the class BITSTRING. However, this can be done by means of **enum** or **#define**, where the defined constants may be "or"-ed to create the desired value. Bit strings represent unsigned values. Although signed types like char, int and long are compatible, they shouldn't be used. Performing value retrieval with a **DWORD** argument, while the actual length of the bit string value is more that 32 bits, results in an exception being thrown.

4. Octet string

ASN.1 Octet string variables carry strings of octets (bytes) of arbitrary length.

<OctetStringType> ::= "OCTET STRING"
<OctetStringValue> ::= <bstring> | <hstring>

ABCD class name: OCTETSTRING
Compatible types: BYTE, UINT, DWORD (char, int, long)

Octet strings represent unsigned values. Although signed types like char, int and long are compatible, they shouldn't be used. The octet string itself can contain up to 64 bytes. Performing value retrieval with a **DWORD** argument, however, will throw an exception if the actual length of the octet string is more that four bytes.

5. Null

The ASN.1 null type is used to indicate the absence of a parameter (see Sequence, Choice and Any types).

<NullType> ::= "NULL"
<NullValue> ::= "NULL"

ABCD class name: Null
Compatible types: none

Note the lowercase notation for the ABCD class name, which is used in order to avoid redefinition of the null-pointer constant **NULL**. ABCD class Null contains no member functions for value assignment or retrieval, since a Null type has no value (other than "NULL").

10. Enumerated

An ASN.1 variable with an enumerated type can assume a limited number of values, which are referenced by associated identifiers.

<EnumeratedType> ::= "ENUMERATED {" <NamedNumber>,,, "}"

<EnumeratedValue> ::= <identifier>

ABCD class name: ENUMERATED
Compatible types: int, long

Assigning identifiers is not implemented in the class ENUMERATED. This can easily be done by means of **enum** or **#define**, though. Since an enum type is essentially an integer, there is really no difference between assigning an enumerated or an integer value. However, it's better to use enum types only. Note that a typecast to int is required when using enum types with the **get** member function.

16. Sequence

The ASN.1 sequence type is used to build a structured data type, consisting of items of arbitrary types. The items may be assigned identifiers, which can be used to reference the item in a value assignment. The items may be marked as optional or may be assigned a default value, in which case they may be omitted in assignments.

<SequenceType> ::= "SEQUENCE {" [<ItemType>,,,] "}"
<ItemType> ::= <NamedType> ["OPTIONAL" | "DEFAULT" <value>] |
"COMPONENTS OF" <SequenceType>
<SequenceValue> ::= "{" [<NamedValue>,,,] "}"

ABCD class name: SEQUENCE
Compatible types: none

Instantiating the class SEQUENCE yields an empty sequence object. The items should be specified later by means of the member function "**item**". This member function is defined as:

```
SEQUENCE::item(string NameID, ASN1type* item_ptr, BOOL optional = FALSE);
```

`NameID` is the name of the item (note that it isn't optional, like in ASN.1)
`item_ptr` is a pointer to an arbitrary ABCD object
the item is marked optional if `optional` is TRUE (default is FALSE)
the constant OPTIONAL (defined as TRUE) can be used as third argument

All identifiers within a sequence should be unique, otherwise an exception is thrown.

Example:

ASN.1	ABCD

`Int1 ::= [0] IMPLICIT INTEGER`
`SeqType ::= [PRIVATE|31] SEQUENCE {`
`first INTEGER ,`
`next Int1 OPTIONAL }`
	`INTEGER Int1(0,IMPLICIT);`
	`SEQUENCE SeqType(PRIVATE|31);`
	`SeqType.item("first",new INTEGER);`
	`SeqType.item("next",Int1(),OPTIONAL);`

Note that it makes no sense to use value assignment or retrieval on the SEQUENCE object itself[19], instead the object's items should be used. Once the entire sequence has been specified, its items can be referenced by means of the operator [ ].

[19] *Attempts to do this result in an exception being thrown*

Example:

```
SEQUENCE seq ;
      seq.item("hello", SeqType() ) ;
      seq.item("bye" , new INTEGER(PRIVATE|1) ) ;
seq = 1; // This is illegal
seq["bye"] = 123 ;
seq["hello"]["second"] = 234 ;
```

Optional items are not included in the encoding unless they have been referenced (in the example, the optional item seq["hello"]["second"] is referenced and therefore will be included in the encoding). To determine if an optional item is included, the member function **contains** can be used on the SEQUENCE object:

```
if ( seq["hello"].contains("second") ) // optional item second is included
```

16. Sequence of

The sequence-of type is used to build arrays consisting of an arbitrary number of elements of the same type.

<SequenceOfType> ::= "SEQUENCE OF" <Type>
<SequenceOfValue> ::= "{" [<Value>,,,] "}"

ABCD class name: SEQUENCE_OF
Compatible types: none

The constructor of class SEQUENCE_OF takes one extra argument: a pointer to an ABCD object that serves as an ASN.1 type.

Example:

ASN.1	ABCD
`IntType = [PRIVATE 4] INTEGER`	`INTEGER IntType(PRIVATE|4)`
`SeqOfType ::= [1] SEQUENCE OF IntType`	`SEQUENCE_OF SeqOfType(IntType(), 1)`

Instantiating the class SEQUENCE_OF yields an empty array. Referencing of elements in the array is done the usual way (indexes start with number one). The array has so-called "self-expanding" behaviour: when an element is referenced with an index equal to the number of elements in the array plus one, a new element is appended.

The number of elements in the array can be retrieved using the member function **get**. When the SEQUENCE_OF object has been encoded, the array returns to empty state.

Example:

```
SEQUENCE_OF SeqOfVar(new INTEGER);  // Empty array of INTEGER is created
SeqOfVar[1] = 123;                  // Append element (array now has one element)
SeqOfVar[1] = 234;                  // This is OK; element value is overwritten
SeqOfVar[2] = 345;                  // Append (array now has two elements)
int elements;
SeqOfVar.get(elements);             // elements = 2
SeqOfVar.BER_encode(buf_ptr); // Encode object and return empty array.
```

22. IA5String

ASN.1 variables of type IA5String carry ASCII character strings of arbitrary length.

<IA5StringType> ::= "IA5String"
<IA5StringValue> ::= <cstring>

ABCD class name: IA5String
Compatible types: LPCSTR, LPSTR, const char*, char*

Value assignments should be of type LPCSTR (defined as const far char*), LPSTR (defined as far char*) can be used with member function get. The IA5String can contain up to 64 characters.

Choice

The ASN.1 choice type allows a choice between the specified types. All types should have different tags (implicit tagging may be necessary). Identifiers may be assigned to the different options, to use as a reference in the value assignments.

<ChoiceType> ::= "CHOICE {" <NamedType>,,, "}"
<ChoiceValue> ::= <NamedValue>

ABCD class name: CHOICE
Compatible types: none

Instantiating class CHOICE yields an empty choice object (i.e. no options defined). The options should be specified later using the member function **option**. This member function is very similar to member function **item** of a sequence type, the only difference being the absence of the third argument (which makes no sense with a choice type).

Since implicit tagging is illegal with a choice type, the constructor function for class CHOICE doesn't accept a second argument.

Like with class SEQUENCE, value assignment or retrieval on the choice object itself is illegal and results in an exception being thrown. The options are referenced with operator [], this automatically selects the referenced option.

Example:

```
INTEGER Int1(0,IMPLICIT);
CHOICE SelType (PRIVATE|31);
      SelType.option("first" , new INTEGER);
      SelType.option("second", Int1() );
SelType["first"] = 123;  // option first is selected
```

To determine which option is selected, the operator = = can be used on the choice object:

```
if ( SelType == "second") // option second is selected
```

Any

The ASN.1 any type is a reference to any ASN.1 type. Unless the DEFINED BY keywords are used, this will result in an incomplete specification. The identifier should reference an item of the structure the any type is part of.

<AnyType> ::= "ANY" ["DEFINED BY" <identifier>]
<AnyValue> ::= <Type> <Value>

ABCD class name: ANY
Compatible types: none

The constructor for class ANY is a little different that the other constructors. It is defined as:

```
ANY::ANY(DefFuncPtr defined_by, BYTE tag = NONE);
```

`defined_by` is a pointer to a so-called definition function
`tag` is an optional explicit tag value (implicit tagging is illegal with an any type)

The definition function should take no arguments and should return an ASN1type*. This function should determine which ABCD object the ANY object has to reference and return a pointer to it. When a BER encoding or decoding member function of an ANY object is invoked, the object calls the definition function to find the object to reference. It then passes control to that object, unless the definition function returns a null-pointer.

Note that the ANY object only contains member functions for BER encoding / decoding and output stream insertion. Any other operations on this object result in an exception being thrown.

Appendix E. TAPI / TSPI interface specification

This appendix presents an abridged version of the information found in both API and SPI programmer's reference manuals of the Windows Telephony system. In order to keep the size manageable (the original size of the information presented in this appendix is about 390 pages), only the most relevant information in relation to the subjects covered in this report is reproduced. More specifically, only functions, messages and constants concerning line devices (i.e. not phone devices) are specified. Data structures are not specified, but the constants specified give an impression of the information in the most relevant data structures.

Functions and messages defined in the Windows Telephony API (TAPI) specification generally have corresponding functions and messages in the Telephony SPI (TSPI) specification. Therefore, TAPI and corresponding TSPI functions and messages are described together in this appendix, as opposed to the original specification. Besides a comprehensive description of each function, TAPI and TSPI function prototypes are presented. A function prototype shows how the function is invoked and reveals the name of the function, its parameters and its return value. The meaning of each parameter is also described.

Parameters are named according to a naming convention known as the Hungarian notation, which is very common in Windows programming. This convention implies that the name of a variable or parameter begins with one or more lowercase letters that denote its data type. Frequently used letters are listed in Table 12. With pointers, prefixes are often combined to denote the type of pointer. For example, *lpfn* is a (far) pointer to a function.

Table 12. Hungarian notation

Prefix	Data type
b	BOOL
c	char
dw	DWORD
fn	function
h	handle
l	long
lp	far (long) pointer
i	int
n	int (number)
p	pointer
s	string
sz	string terminated by zero byte
w	WORD

Functions

lineAccept

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

TAPI function:

`LONG` **`lineAccept`**`(hCall, lpsUserUserInfo, dwSize)`

`hCall` Specifies a handle to the call to be accepted. The application must be an owner of the call.

`lpsUserUserInfo` Specifies a far 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-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` Specifies 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.

TSPI function:

`LONG` **`TSPI_lineAccept`**`(dwRequestID, hdCall, lpsUserUserInfo, dwSize)`

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call to be accepted.

`lpsUserUserInfo` Specifies a far pointer to a string containing user-to-user information to be sent to the remote party as part of the call accept. This pointer is 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**).

`dwSize` Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*. If *lpsUserUserInfo* is NULL, *dwSize* should be ignored.

lineAddToConference

This function adds the call specified by *hConsultCall* to the conference call specified by *hConfCall*. Note that 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**.
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 **linePrepareAddTo-Conference**. 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.

TAPI function:

`LONG` **`lineAddToConference`**`(hConfCall, hConsultCall)`

`hConfCall` Specifies 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.

`hConsultCall` Specifies 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**.

TSPI function:

`LONG` **`TSPI_lineAddToConference`**`(dwRequestID, hdConfCall, hdConsultCall)`

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdConfCall` Specifies the handle to the conference call.

`hdConsultCall` Specifies the handle to the call to be added to the conference call. This call cannot be either a parent of another conference or a participant in any conference. Depending on the device capabilities indicated in **LINEADDRESSCAPS**, the *hdConsultCall* may not necessarily have been established using **TSPI_lineSetupConference** or **TSPI_linePrepare-AddToConference**.

lineAnswer

This function answers the specified offering call. 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.
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.

TAPI function:

```
LONG lineAnswer(hCall, lpsUserUserInfo, dwSize)
```

`hCall` Specifies a handle to the call to be answered. The application must be an owner of this call.

`lpsUserUserInfo` Specifies a far 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-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` Specifies 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.

TSPI function:

```
LONG TSPI_lineAnswer(dwRequestID, hdCall, lpsUserUserInfo, dwSize)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the service provider's handle to the call to be answered.

`lpsUserUserInfo` Specifies a far pointer to a string containing user-to-user information to be sent to the remote party at the time of answering the call. If this pointer is NULL, it indicates that no user-to-user information is to be sent. User-to-user information is only sent if supported by the underlying network (as indicated in **LINEDEVCAPS**).

`dwSize` Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*. If *lpsUserUserInfo* is NULL *dwSize* is ignored.

lineBlindTransfer

This function performs a blind or single-step transfer of the specified call to the specified destination address.
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 **lineHandoff** for this purpose.

TAPI function:

```
LONG lineBlindTransfer(hCall, lpszDestAddress, dwCountryCode)
```

`hCall` Specifies a handle to the call to be transferred. The application must be an owner of this call.

`lpszDestAddress` Specifies a far 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` Specifies 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.

TSPI function:

```
LONG TSPI_lineBlindTransfer(dwRequestID, hdCall, lpszDestAddress,
    dwCountryCode)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the service provider's handle to the call to be transferred.

`lpszDestAddress` Specifies a far 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` Specifies the country code of the destination. This should be used by the implementation to select the call progress protocols for the destination address. If a value of zero is specified, the service provider should use a default. Note that *dwCountryCode* is not validated by TAPI.DLL when this function is called.

lineClose

This function closes the specified open line device.
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 explicity 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.
Certain environments 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.

TAPI function:

```
LONG lineClose(hLine)
```

hLine Specifies a handle to the open line device to be closed. After the line has been successfully closed, this handle is no longer valid.

TSPI function:

```
LONG TSPI_lineClose(hdLine)
```

hdLine Specifies the service provider's handle to the line to be closed. After the line has been successfully closed, this handle is no longer valid.

lineCompleteCall

This function is used to specify 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.
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**.

TAPI function:

```
LONG lineCompleteCall(hCall, lpdwCompletionID, dwCompletionMode,
    dwMessageID)
```

`hCall` Specifies a handle to the call whose completion is requested. The application must be an owner of the call.

`lpdwCompletionID` Specifies a far 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` Specifies 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: CAMPON queues the call until the call can be completed. The call remains in the *busy* state while queued; CALLBACK requests the called station to return the call when it returns to *idle*; INTRUDE adds the application to the existing physical call at the called station (barge in); MESSAGE leave a short predefined message for the called station ("Leave Word Calling"). The message to be sent is specified by *dwMessageID*.

`dwMessageID` Specifies 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.

TSPI function:

```
LONG TSPI_lineCompleteCall(dwRequestID, hdCall, lpdwCompletionID,
    dwCompletionMode, dwMessageID)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the service provider's handle to the call whose completion is requested.

`lpdwCompletionID` Specifies a far pointer to a DWORD-sized memory location where the service provider writes a completion ID.

`dwCompletionMode` Specifies the way in which the call is to be completed. This parameter uses the following **LINECALLCOMPLMODE_** constants. Only one of the indicated flags may be set at a time.

`dwMessageID` Specifies 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. Note that this parameter is not validated by TAPI.DLL when this function is called.

lineCompleteTransfer

This function completes the transfer of the specified call to the party connected in the consultation call. 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 **lineHandoff** 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.

TAPI function:

```
LONG lineCompleteTransfer(hCall, hConsultCall, lphConfCall, dwTransferMode)
```

`hCall` Specifies a handle to the call to be transferred. The application must be an owner of this call.

`hConsultCall` Specifies a handle to the call that represents a connection with the destination of the transfer. The application must be an owner of this call.

`lphConfCall` Specifies a far pointer to a memory location where an HCALL handle can be returned. If *dwTransferMode* is 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.DLL.

`dwTransferMode` Specifies how the initiated transfer request is to be resolved. This parameter uses the following LINETRANSFERMODE_ constants: TRANSFER resolve the initiated transfer by transferring the initial call to the consultation call; 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.

TSPI function:

```
LONG TSPI_lineCompleteTransfer(dwRequestID, hdCall, hdConsultCall,
    htConfCall, lphdConfCall, dwTransferMode)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the service provider's handle to the call to be transferred.

`hdConsultCall` Specifies a handle to the call that represents a connection to the destination of the transfer.

`htConfCall` This parameter is only valid if *dwTransferMode* is specified as CONFERENCE. It should be used to replace the *htCall* associated with the original *hdCall*. The service provider must save this parameter value and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call. Otherwise this parameter is ignored.

`lphdConfCall` Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the call. This parameter is only valid if *dwTransferMode* is specified as CONFERENCE. The service provider must fill this location with its handle for the new conference call.

`dwTransferMode` Specifies how the initiated transfer request is to be resolved.

lineConfigDialog

This 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.
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.

TAPI function:

LONG **lineConfigDialog**(dwDeviceID, hwndOwner, lpszDeviceClass)

dwDeviceID Specifies the line device to be configured.

hwndOwner Specifies a handle to a window to which the dialog is to be attached.

lpszDeviceClass Specifies a far 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.

TSPI function:

LONG **TSPI_lineConfigDialog**(dwDeviceID, hwndOwner, lpszDeviceClass)

dwDeviceID Specifies the line device to be configured.

hwndOwner Specifies a handle to a parent window in which the dialog window is to be placed.

lpszDeviceClass Specifies a far pointer to a NULL-terminated string that identifies a device class name. This device class allows the caller to select a specific subscreen of configuration information applicable to that device class. If this parameter is NULL or an empty string, the highest level configuration dialog should be selected. The permitted strings are the same as for **TSPI_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, to start at the corresponding point in a multilevel configuration dialog chain, so that the user doesn't have to search to find the desired parameters.)

lineDeallocateCall

This function deallocates the specified call handle.
The deallocation does not affect the call state of the physical call. It does, however, release internal resources related to the call. 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.

TAPI function:

`LONG` **`lineDeallocateCall`**`(hCall)`

`hCall` Specifies 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.

TSPI function:

`LONG` **`TSPI_lineCloseCall`**`(hdCall)`

`hdCall` Specifies the service provider's handle to the call to be closed. After the call has been successfully closed, this handle is no longer valid.

lineDevSpecific

This function is used as a general extension mechanism that 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.
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 meaning 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.

TAPI function:

```
LONG lineDevSpecific(hLine, dwAddressID, hCall, lpParams, dwSize)
```

`hLine` Specifies a handle to a line device. This parameter is required.

`dwAddressID` Specifies an address ID on the given line device.

`hCall` Specifies 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.

`lpParams` Specifies a far 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.DLL to or from the service provider.

`dwSize` The size in bytes of the parameter block area.

TSPI function:

```
LONG TSPI_lineDevSpecific(dwRequestID, hdLine, dwAddressID, hdCall,
    lpParams, dwSize)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdLine` Specifies the service provider's handle to the line to be operated on.

`dwAddressID` Specifies the address on the specified line to be operated on.

`hdCall` Specifies the service provider's handle to the call to be operated on. This field may have the value NULL.

`lpParams` Specifies a far pointer to a memory area used to hold a parameter block. The format of this parameter block is device specific.

`dwSize` The size in bytes of the parameter block area.

lineDevSpecificFeature

This function is used as an extension mechanism that 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.
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 since they would not be properly translated (thunked) when running a 16-bit application in a 32-bit version of TAPI.DLL and vice versa.

TAPI function:

```
LONG lineDevSpecificFeature(hLine, dwFeature, lpParams, dwSize)
```

`hLine` Specifies a handle to the line device.

`dwFeature` Specifies the feature to invoke on the line device.

`lpParams` Specifies a far 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.DLL to or from the service provider.

`dwSize` Specifies the size of the buffer in bytes.

TSPI function:

```
LONG TSPI_lineDevSpecificFeature(dwRequestID, hdLine, dwFeature, lpParams,
    dwSize)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdLine` Specifies the service provider's handle to the line device.

`dwFeature` Specifies the feature to invoke on the line device.

`lpParams` Specifies a far pointer to a memory area used to hold a feature-dependent parameter block. The format of this parameter block is device specific.

`dwSize` Specifies the size of the buffer in bytes.

lineDial

This function dials the specified dialable number on the specified call.
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 multi-stage dialing, if the line's device capabilities allows 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**.

TAPI function:

```
LONG lineDial(hCall, lpszDestAddress, dwCountryCode)
```

`hCall` Specifies a handle to the call on which a number is to be dialed. The application must be an owner of the call.

`lpszDestAddress` Specifies the destination to be dialed using the standard dialable number format.

`dwCountryCode` Specifies 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.

TSPI function:

```
LONG TSPI_lineDial(dwRequestID, hdCall, lpszDestAddress, dwCountryCode)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the service provider's handle to the call to be dialed.

`lpszDestAddress` Specifies the destination to be dialed using the standard dialable number format.

`dwCountryCode` Specifies 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. Note that this parameter is not validated by TAPI.DLL when this function is called.

lineDrop

This 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.
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.

TAPI function:

```
LONG lineDrop(hCall, lpsUserUserInfo, dwSize)
```

`hCall` Specifies a handle to the call to be dropped. The application must be an owner of the call.

`lpsUserUserInfo` Specifies a far 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-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` Specifies 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.

TSPI function:

```
LONG TSPI_lineDrop(dwRequestID, hdCall, lpsUserUserInfo, dwSize)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the service provider's handle to the call to be dropped.

`lpsUserUserInfo` This pointer is only valid if **dwUserUserInfoSize** is non-zero. It specifies a far pointer to a string containing user-to-user information to be sent to the remote party as part of the call disconnect. This pointer is 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**).

`dwSize` Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*. If *lpsUserUserInfo* is Null dwSize should be ignored.

lineForward

This 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.

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.

Since 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. Since 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.

TAPI function:

```
LONG lineForward(hLine, bAllAddresses, dwAddressID, lpForwardList,
    dwNumRingsNoAnswer, lphConsultCall, lpCallParams)
```

`hLine` Specifies 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` Specifies the address on the specified line whose incoming calls are to be forwarded. This parameter is ignored if *bAllAddresses* is TRUE.

`lpForwardList` Specifies a far pointer to a variably sized data structure that describes the specific forwarding instructions, of type **LINEFORWARDLIST**.

`dwNumRingsNoAnswer` Specifies 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 Specifies a far 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` Specifies a far 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*.

TSPI function:

```
LONG TSPI_lineForward(dwRequestID, hdLine, bAllAddresses, dwAddressID,
    lpForwardList, dwNumRingsNoAnswer, htConsultCall, lphdConsultCall,
    lpCallParams)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdLine` Specifies the service provider's handle to the line to be forwarded.

`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. Note that this parameter is not validated by TAPI.DLL when this function is called.

`dwAddressID` Specifies the address on the specified line whose incoming calls are to be forwarded. This parameter is ignored if *bAllAddresses* is TRUE. Note that this parameter is not validated by TAPI.DLL when this function is called.

`lpForwardList` Specifies a far pointer to a variably sized data structure of type **LINEFORWARDLIST** that describes the specific forwarding instructions.

`dwNumRingsNoAnswer` Specifies the number of rings before an incoming 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. Note that this parameter is not validated by TAPI.DLL when this function is called.

`htConsultCall` Specifies TAPI.DLL's handle to a new call, if such a call must be created by the service provider. In some telephony environments, forwarding a call has the side effect of creating a consultation call used to consult the party that is being forwarded to. In such an environment, the service provider creates the new consutation call and must save this value and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call. If no consultation call is created, this value can be ignored by the service provider.

`lphdConsultCall` Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the call. In telephony environments where forwarding a call has the side effect of creating a consultation call used to consult the party that is being forwarded to, the service provider must fill this location with its handle for the call before this procedure returns. The service provider is permitted to do callbacks regarding the new call before it returns from this procedure. If no consultation call is created, the HDRVCALL must be left NULL.

`lpCallParams` Specifies a far pointer to a structure of type **LINECALLPARAMS**. This pointer is ignored by the service provider unless **lineForward** requires the establishment of a call to the forwarding destination (and *lphdConsultCall* is returned, in which case *lpCallParams* is optional). If NULL, default call parameters are used. Otherwise, the specified call parameters are used for establishing *htConsultCall*.

lineGatherDigits

This 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.
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 and the contents of the old buffer is undefined. The mechanism for terminating digit gathering without initiating another function 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.
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.
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.

TAPI function:

```
LONG lineGatherDigits(hCall, dwDigitModes, lpsDigits, dwNumDigits,
    lpszTerminationDigits, dwFirstDigitTimeout, dwInterDigitTimeout)
```

`hCall` Specifies a handle to the call on which digits are to be gathered. The application must be an owner of the call.

`dwDigitModes` Specifies 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: 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'; DTMF detect digits as DTMF tones. Valid digits for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#'.

`lpsDigits` Specifies a far 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` Specifies 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` Specifies 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 specifies the valid range for this parameter or indicates that timeouts are not supported.

`dwInterDigitTimeout` Specifies 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 specifies the valid range for this parameter or indicates that timeouts are not supported.

TSPI function:

```
LONG TSPI_lineGatherDigits(hdCall, dwEndToEndID, dwDigitModes, lpsDigits,
    dwNumDigits, lpszTerminationDigits, dwFirstDigitTimeout,
    dwInterDigitTimeout)
```

`hdCall` Specifies the service provider's handle to the call on which digit gathering is to be performed.

`dwEndToEndID` Specifies a unique, uninterpreted identifier of the request for its entire lifetime, that is, until the matching **LINE_GATHERDIGITS** message is sent. The service provider includes this identifier as one of the parameters in the message.

`dwDigitModes` Specifies the digit mode(s) that are to be monitored.

`lpsDigits` Specifies a far pointer to the buffer where detected digits are to be stored as ASCII characters. The service provider may, but is not required to, place digits in the buffer one at a time as they are collected. When the **LINE_GATHERDIGITS** message is sent, the content of the buffer must be complete. If *lpsDigits* is specified as NULL the digit gathering currently in progress on the call is canceled and the *dwNumDigits* parameter is ignored. Otherwise, *lpsDigits* is assumed to have room for *dwNumDigits* digits.

`dwNumDigits` Specifies the number of digits to be collected before a **LINE_GATHERDIGITS** message is sent to TAPI.DLL. *dwNumDigits* is ignored when *lpsDigits* is NULL. This function must return a LINEERR_INVALPARAM 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 TAPI.DLL.

`dwFirstDigitTimeout` Specifies 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 terminated and a **LINE_GATHERDIGITS** message is sent to TAPI.DLL. A single NULL character is written to the buffer, indicating no digits were received and the first digit timeout terminated digit gathering. The call's line device capabilities specifies the valid range for this parameter or indicates that timeouts are not supported. Note that this parameter is not validated by TAPI.DLL when this function is called.

`dwInterDigitTimeout` Specifies the maximum time duration in milliseconds between consecutive digits. If no digit is received in this timeframe, digit collection is terminated and a **LINE_GATHERDIGITS** message is sent to TAPI.DLL. A single NULL character is written to the buffer, indicating that an interdigit timeout terminated digit gathering. The **LINEDEVCAPS** structure must specify the the valid range for this parameter or indicate that timeouts are not supported. Note that this parameter is not validated by TAPI.DLL when this function is called.

lineGenerateDigits

This 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.
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.
To cancel the current digit generation, the application can invoke **lineGenerateDigits** and specify NULL for the *lpszDigits* parameter.

TAPI function:

```
LONG lineGenerateDigits(hCall, dwDigitMode, lpszDigits, dwDuration)
```

`hCall` Specifies a handle to the call. The application must be an owner of the call.

`dwDigitMode` Indicates 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: PULSE uses pulse/rotary for digit signaling; DTMF uses DTMF tone signaling.

`lpszDigits` Specifies a far 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 indicates what this duration is. Multiple commas may be used to inject longer pauses.

`dwDuration` Specifies 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.

TSPI function:

```
LONG TSPI_lineGenerateDigits(hdCall, dwEndToEndID, dwDigitMode, lpszDigits,
    dwDuration)
```

`hdCall` Specifies the handle to the call on which digit generation is to be done.

`dwEndToEndID` This unique request ID should be stored by the service provider and passed back as *dwParam2* to the **LINEEVENT** procedure when the digit generation is completed.

`dwDigitMode` Indicates the format to be used for signaling these digits.

`lpszDigits` Specifies a far pointer to a NULL terminated character buffer that contains the digits to be generated..

`dwDuration` Specifies both the duration in milliseconds of DTMF digits and pulse and DTMF inter-digit spacing. A value of zero will use a default value. *dwDuration* must be within the range specified by **MinDialParams** to **MaxDialParams** in **LINEDEVCAPS**. If out of range, the actual value is set by the service provider to the nearest value in the range. Note that this parameter is not validated by TAPI.DLL when this function is called.

lineGenerateTone

This 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.
lineGenerateTone 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. Since 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.

TAPI function:

LONG **lineGenerateTone**(hCall, dwToneMode, dwDuration, dwNumTones, lpTones)

hCall Specifies a handle to the call on which a tone is to be generated. The application must be an owner of the call.

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 cancelled. This parameter uses the following LINETONEMODE_ constants: CUSTOM, defined by the specified frequencies; RINGBACK; BUSY; BEEP; BILLING. A value of zero for *dwToneMode* cancels tone generation.

dwDuration Specifies duration in milliseconds during which the tone should be sustained. A value of zero for *dwDuration* uses a default duration for the specified tone.

dwNumTones Specifies the number of entries in the *lpTones* array. This field is ignored if *dwToneMode* is not equal to CUSTOM.

lpTones Specifies a far 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.

TSPI function:

LONG **TSPI_lineGenerateTone**(*hdCall, dwEndToEndID, dwToneMode, dwDuration, dwNumTones, lpTones*)

hdCall Specifies the service provider's handle to the call on which tone generation is to be performed.

dwEndToEndID Specifies a unique, uninterpreted identifier of the request for its entire lifetime, that is, until the matching **LINE_GENERATE** message is sent. The service provider includes this identifier as one of the parameters in the message.

dwToneMode Defines the tone to be generated. If *dwToneMode* is set to zero, any digit or tone generation then in progress is cancelled.

dwDuration Specifies the duration in milliseconds during which the tone should be sustained. A value of zero for *dwDuration* uses a default duration for the specified tone. Note that this parameter is not validated by TAPI.DLL when this function is called.

dwNumTones Specifies the number of entries in the *lpTones* array. This field is ignored if *dwToneMode* is not equal to LINETONEMODE_CUSTOM.

lpTones Specifies a far 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.

lineGetAddressCaps

This function queries the specified address on the specified line device to determine its telephony capabilities.
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**.

TAPI function:

```
LONG lineGetAddressCaps(hLineApp, dwDeviceID, dwAddressID, dwAPIVersion,
     dwExtVersion, lpAddressCaps)
```

`hLineApp` Specifies the handle to the application's registration with TAPI.

`dwDeviceID` Specifies the line device containing the address to be queried.

`dwAddressID` Specifies the address on the given line device whose capabilities are to be queried.

`dwAPIVersion` Specifies 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.

`dwExtVersion` Specifies 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; the low-order word contain the minor version number.

`lpAddressCaps` Specifies a far 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.DLL for returning information.

TSPI function:

```
LONG TSPI_lineGetAddressCaps(dwDeviceID, dwAddressID, dwTSPIVersion,
     dwExtVersion, lpAddressCaps)
```

`dwDeviceID` Specifies the line device containing the address to be queried.

`dwAddressID` Specifies the address on the given line device whose capabilities are to be queried. Note that this parameter is not validated by TAPI.DLL when this function is called.

`dwTSPIVersion` Specifies the version number of the Telephony SPI to be used. The high-order word contains the major version number; the low-order word contains the minor version number.

`dwExtVersion` Specifies the version number of the service provider-specific extensions to be used. This number will be zero if no device-specific extensions are to be used. Otherwise, the high-order word contains the major version number; the low-order word contain the minor version number. Note that this parameter is not validated by TAPI.DLL when this function is called.

`lpAddressCaps` Specifies a far pointer to a variably sized structure of type **LINEADDRESSCAPS**. Upon successful completion of the request, this structure is filled with address capabilities information.

lineGetAddressID

This operation returns the address ID associated with an address in a different format on the specified line.

This 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**.

TAPI function:

```
LONG lineGetAddressID(hLine, lpdwAddressID, dwAddressMode, lpsAddress,
     dwSize)
```

`hLine` Specifies a handle to the open line device.

`lpdwAddressID` Specifies a far pointer to a DWORD-sized memory location where the address ID is returned.

`dwAddressMode` Specifies 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: DIALABLEADDR the address is specified by its dialable address. The *lpsAddress* parameter is the dialable address or canonical address format.

`lpsAddress` Specifies a far pointer to a data structure holding the address assigned to the specified line device. The format of the address is determined by *dwAddressMode*. Since the only valid value is DIALABLEADDR, *lpsAddress* uses the common dialable number format and is NULL-terminated.

`dwSize` Specifies the size of the address contained in *lpsAddress*.

TSPI function:

```
LONG TSPI_lineGetAddressID(hdLine, lpdwAddressID, dwAddressMode,
     lpsAddress, dwSize)
```

`hdLine` Specifies the service provider's handle to the line whose address is to be retrieved.

`lpdwAddressID` Specifies a far pointer to a DWORD-sized memory location where the address ID is returned.

`dwAddressMode` Specifies the address mode of the address contained in *lpsAddress*.

`lpsAddress` Specifies a far pointer to a data structure holding the address assigned to the specified line device. The format of the address is determined by *dwAddressMode* parameter.

`dwSize` Specifies the size of the address contained in *lpsAddress*. The parameter *dwSize* must be set to the length of the string (plus one for the NULL) if DIALABLEADDR is used. If an extended LINEADDRESSMODE is used, the length should match the size of whatever is actually passed in (the DLL checks to be sure it can read the number of bytes specified from the pointer given).

lineGetAddressStatus

This operation allows an application to query the specified address for its current status.

TAPI function:

LONG **lineGetAddressStatus**(hLine, dwAddressID, lpAddressStatus)

hLine Specifies a handle to the open line device.

dwAddressID Specifies an address on the given open line device. This is the address to be queried.

lpAddressStatus Specifies a far 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.DLL for returning information.

TSPI function:

LONG **TSPI_lineGetAddressStatus**(hdLine, dwAddressID, lpAddressStatus)

hdLine Specifies the service provider's handle to the line containing the address to be queried.

dwAddressID Specifies an address on the given open line device. This is the address to be queried. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpAddressStatus Specifies a far pointer to a variably sized data structure of type **LINEADDRESS-STATUS**.

lineGetCallInfo

This operation enables an application to obtain fixed information about the specified call.
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.

TAPI function:

LONG **lineGetCallInfo**(hCall, lpCallInfo)

hCall Specifies a handle to the call to be queried.

lpCallInfo Specifies a far 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.DLL for returning information.

TSPI function:

LONG **TSPI_lineGetCallInfo**(hdCall, lpCallInfo)

hdCall Specifies the service provider's handle to the call whose call information is to be retrieved.

lpCallInfo Specifies a far pointer to a variably sized data structure of type **LINECALLINFO**. Upon successful completion of the request, this structure is filled with call-related information.

lineGetCallStatus

This operation returns the current status of the specified call. 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.

TAPI function:

`LONG` **`lineGetCallStatus`**`(hCall, lpCallStatus)`

`hCall` Specifies a handle to the call to be queried.

`lpCallStatus` Specifies a far 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.DLL for returning information.

TSPI function:

`LONG` **`TSPI_lineGetCallStatus`**`(hdCall, lpCallStatus)`

`hdCall` Specifies the service provider's handle to the call to be queried for its status.

`lpCallStatus` Specifies a far pointer to a variably sized data structure of type **LINECALLSTATUS**. This structure is filled with call status information.

lineGetConfRelatedCalls

This operation 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.
The specified call can either be a conference call handle or a handle to a participant call. 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. The application can invoke **lineGetCallInfo** and **lineGetCallStatus** for each call in the list to determine the call's information and status, respectively.

TAPI function:

`LONG lineGetConfRelatedCalls(hCall, lpCallList)`

`hCall` Specifies a handle to a call. This is either a conference call or a participant call in a conference call.

`lpCallList` Specifies a far 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.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineGetDevCaps

This function queries a specified line device to determine its telephony capabilities. The returned information is valid for all addresses on the line device.
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.DLL 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.

TAPI function:

```
LONG lineGetDevCaps(hLineApp, dwDeviceID, dwAPIVersion, dwExtVersion,
     lpLineDevCaps)
```

`hLineApp` Specifies the handle to the application's registration with TAPI.

`dwDeviceID` Specifies the line device to be queried.

`dwAPIVersion` Specifies 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` Specifies 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` Specifies a far 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.DLL for returning information.

TSPI function:

```
LONG TSPI_lineGetDevCaps(dwDeviceID, dwTSPIVersion, dwExtVersion,
     lpLineDevCaps)
```

`dwDeviceID` Specifies the line device to be queried.

`dwTSPIVersion` Specifies the negotiated TSPI version number. This value has already been negotiated for this device through the **TSPI_lineNegotiateTSPIVersion** function.

`dwExtVersion` Specifies the negotiated extension version number. This value has already been negotiated for this device through the **TSPI_lineNegotiateExtVersion** function. Note that this parameter is not validated by TAPI.DLL when this function is called.

`lpLineDevCaps` Specifies a far pointer to a variably sized structure of type **LINEDEVCAPS**. Upon successful completion of the request, this structure is filled with line device capabilities information.

lineGetDevConfig

This 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.
This 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).

TAPI function:

LONG **lineGetDevConfig** (dwDeviceID, lpDeviceConfig, lpszDeviceClass)

dwDeviceID Specifies the line device to be configured.

lpDeviceConfig Specifies a far 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.DLL for returning information.

lpszDeviceClass Specifies a far 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.

TSPI function:

LONG **TSPI_lineGetDevConfig** (dwDeviceID, lpDeviceConfig, lpszDeviceClass)

dwDeviceID Specifies the line device to be configured.

lpDeviceConfig Specifies a far pointer to a data structure of type **VARSTRING** where the device configuration structure of the associated device is returned. Upon successful completion of the request, the service provider fills this data structure with the device configuration. The **dwStringFormat** field in the **VARSTRING** structure must be set to STRINGFORMAT_BINARY.

lpszDeviceClass Specifies a far pointer to a NULL-terminated ASCII string that specifies the device class of the device whose configuration is requested. Valid device class strings are the same as those specified for the **TSPI_lineGetID** function when it is applied to a "line" device (*dwSelect* has the value LINE).

lineGetIcon

This function allows an application to retrieve a service line device-specific (or provider-specific) icon for display to the user.
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.
lpszDeviceClass 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", "a", or NULL may be used to request the icon for the line service. If the provider does not return an icon, TAPI.DLL substitutes a generic Windows Telephony line device icon.

TAPI function:

```
LONG lineGetIcon(dwDeviceID, lpszDeviceClass, lphIcon)
```

`dwDeviceID` Specifies the line device whose icon is requested.

`lpszDeviceClass` Specifies a far 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` Specifies a far pointer to a memory location in which the handle to the icon is returned.

TSPI function:

```
LONG TSPI_lineGetIcon(dwDeviceID, lpszDeviceClass, lphIcon)
```

`dwDeviceID` Specifies the line device whose icon is requested.

`lpszDeviceClass` Specifies a far pointer to a NULL-terminated string that identifies a device class name. This device class allows the caller to select an icon specific to that device class. This parameter is optional and can be left NULL, in which case the highest level icon associated with the line device rather than a specified media stream device would be selected. Permitted strings are the same as for **TSPI_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.

`lphIcon` Specifies a far pointer to a memory location in which the handle to the icon is returned.

lineGetID

This function returns a device ID for the specified device class associated with the selected line, address, or call.
This 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, mciwave, or NDIS) associated with a call, address or line. This ID can then be used with the appropriate API (such as phone, mci, midi, wave) to select the corresponding media device associated with the specified call.
Note that the notion of a Windows device class is different from that of a media mode. For example, the interactive voice or stored voice media modes may be accessed using either the mci waveaudio or the low level wave device classes. A media mode describes a format of information on a call, and a device class defines a Windows API that is used to manage that stream. Often, a single media stream may be accessed using multiple device classes, or a single device class (such as the one corresponding to the Windows Comm API) may provide access to multiple media modes.
Some common device class names that are currently used are listed below. The first portion of a name identifies the "API" used to manage the device class, and the second portion is typically used to identify a specific device type extension or subset of the overall API. Names are not case sensitive:

Device	Description
comm	generic serial-device API; comm port
comm/datamodem	reserved for use in a future version of Microsoft Windows
wave	low-level waveaudio
mci/midi	high-level midi sequencer
mci/wave	high-level wave device control
tapi/line	TAPI line device
tapi/phone	TAPI phone device
ndis	network driver interface

In the future, there may be additional extensions to the Comm API, possibly to handle fax, ADSI, TDD, and other special data modes. These will be added once they are defined.
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".

TAPI function:

```
LONG lineGetID(hLine, dwAddressID, hCall, dwSelect, lpDeviceID,
    lpszDeviceClass)
```

`hLine` Specifies a handle to an open line device.

`dwAddressID` Specifies an address on the given open line device.

`hCall` Specifies 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: LINE selects the specified line device. The *hLine* parameter must be a valid line handle; *hCall* and *dwAddressID* are ignored; ADDRESS selects the specified address on the line. Both *hLine* and *dwAddressID* must be valid; *hCall* is ignored; CALL selects the specified call. *hCall* must be valid; *hLine* and *dwAddressID* are both ignored.

`lpDeviceID` Specifies a far 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.DLL for returning information.

`lpszDeviceClass` Specifies a far 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.

TSPI function:

`LONG` **`TSPI_lineGetID`**`(hdLine, dwAddressID, hdCall, dwSelect, lpDeviceID, lpszDeviceClass)`

`hdLine` Specifies the service provider's handle to the line to be queried.

`dwAddressID` Specifies an address on the given open line device. Note that this parameter is not validated by TAPI.DLL when this function is called.

`hdCall` Specifies the service provider's handle to the call to be queried.

`dwSelect` Specifies the whether the device ID requested is associated with the line, address or a single call.

`lpDeviceID` Specifies a far pointer to the 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.

`lpszDeviceClass` Specifies a far 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 (such as COM, Wave, and MCI.)

lineGetLineDevStatus

This operation enables an application to query the specified open line device for its current status.
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.

TAPI function:

`LONG` **`lineGetLineDevStatus`**`(hLine, lpLineDevStatus)`

`hLine` Specifies a handle to the open line device to be queried.

`lpLineDevStatus` Specifies a far 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.DLL for returning information.

TSPI function:

`LONG` **`TSPI_lineGetLineDevStatus`**`(hdLine, lpLineDevStatus)`

`hdLine` Specifies the service provider's handle to the line to be queried.

`lpLineDevStatus` Specifies a far pointer to a variably sized data structure of type **LINEDEVSTATUS**. This structure is filled with the line's device status.

lineGetNewCalls

This operation 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.
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.

TAPI function:

```
LONG lineGetNewCalls(hLine, dwAddressID, dwSelect, lpCallList)
```

`hLine` Specifies a handle to an open line device.

`dwAddressID` Specifies an address on the given open line device.

`dwSelect` Specifies the selection of calls that are requested. Note that *dwSelect* can only have one bit set. This parameter uses the following LINECALLSELECT_ constants: LINE selects calls on the specified line device. The *hLine* parameter must be a valid line handle; *dwAddressID* is ignored; ADDRESS selects calls on the specified address on the specified line device. Both *hLine* and *dwAddressID* must be valid.

`lpCallList` Specifies a far 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

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineGetNumRings

This function can be used by an application to determine the number of rings an inbound call on the given address should ring prior to answering the call. 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**. Since this number may vary dynamically, an application should invoke **lineGetNumRings** each time it has the option to answer a call. A separate **LINE_LINEDEVSTATE** *ringing* message is sent to the application for each ring cycle.

TAPI function:

```
LONG lineGetNumRings(hLine, dwAddressID, lpdwNumRings)
```

`hLine` Specifies a handle to the open line device.

`dwAddressID` Specifies an address on the line device.

`lpdwNumRings` Specifies the number of rings that is the minimum of all current **lineSetNumRings** requests.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineGetRequest

This function retrieves the next by-proxy request for the specified request mode.
A telephony-enabled application can request that a call be placed on its behalf by invoking **tapiRequestMakeCall** or **tapiRequestMediaCall**. These requests are queued by TAPI.DLL 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. For media mode handling, the serving application may need to send Windows messages back to the original application that made the request. The **TAPI_REPLY** message is used for this purpose.
After execution of **lineGetRequest**, TAPI.DLL 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 preceeding 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.DLL.
Also note that **tapiRequestDrop** requests are passed directly to the recipient application by the **LINE_REQUEST** message; they are not queued, and are not retrieved using **lineGetRequest**.

TAPI function:

`LONG` **`lineGetRequest`**`(hLineApp, dwRequestMode, lpRequestBuffer)`

`hLineApp` Specifies the application's usage handle for the line portion of TAPI.

`dwRequestMode` Specifies 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: MAKECALL a **tapiRequestMakeCall** request; MEDIACALL a **tapiRequestMediaCall** request.

`lpRequestBuffer` Specifies a far 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 MAKECALL, interpret the content of the request buffer using the **LINEREQMAKECALL** structure. If *dwRequestMode* is MEDIACALL, interpret the content of the request buffer using the **LINEREQMEDIACALL** structure.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineGetStatusMessages

This operation 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.
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.

TAPI function:

`LONG` **`lineGetStatusMessages`**`(hLine, lpdwLineStates, lpdwAddressStates)`

`hLine` Specifies a handle to the line device.

`lpdwLineStates` Specifies 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.

`lpdwAddressStates` Specifies 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.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineGetTranslateCaps

This function returns address translation capabilities.

TAPI function:

LONG **lineGetTranslateCaps**(*hLineApp, dwAPIVersion, lpTranslateCaps*)

`hLineApp` Specifies the application handle returned by **lineInitialize**.

`dwAPIVersion` Indicates the version of TAPI negotiated by **lineNegotiateAPIVersion**.

`lpTranslateCaps` Specifies a far 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.DLL for returning information.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineHandoff

This function is used to give ownership of the specified call to another application. The application can be either specified directly by its file name or indirectly as the highest priority application that handles calls of the specified media mode.
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 file name of the target application, it can simply specify the file name 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.
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 callback, 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.

TAPI function:

```
LONG lineHandoff(hCall, lpszFileName, dwMediaMode)
```

`hCall` Specifies a handle to the call to be handed off. The application must be an owner of the call.

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

`dwMediaMode` Specifies 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.

lineHold

This function places the specified call on hold.
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*.

TAPI function:

```
LONG lineHold(hCall)
```

`hCall` Specifies a handle to the call to be placed on hold. The application must be an owner of the call.

TSPI function:

```
LONG TSPI_lineHold(dwRequestID, hdCall)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the service provider's handle to the call to be placed on hold.

lineInitialize

This 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.
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 by **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.

TAPI function:

```
LONG lineInitialize(lphLineApp, hInstance, lpfnCallback, lpszAppName,
    lpdwNumDevs)
```

`lphLineApp` Specifies a far pointer to a location that is filled with the application's usage handle for TAPI.

`hInstance` Specifies the instance handle of the client application or DLL.

`lpfnCallback` Specifies the address of a callback function that is invoked to determine status and events on the line device, addresses, or calls.

`lpszAppName` Specifies a far 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 *lpszAppName* is NULL, the application's file name is used instead.

`lpdwNumDevs` Specifies a far 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.

TSPI function:

The first application that calls lineInitialize causes all service providers to be loaded. This in turn results in the function TSPI_providerInit to be called in each service provider.

lineMakeCall

This 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.
After dialing has completed, several **LINE_CALLSTATE** messages are usually sent to the application to notify it about the progress of the call. 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**.

TAPI function:

```
LONG lineMakeCall(hLine, lphCall, lpszDestAddress, dwCountryCode,
    lpCallParams)
```

`hLine` Specifies a handle to the open line device on which a call is to be originated.

`lphCall` Specifies a far pointer to a call handle. This location is filled with a handle identifying the new call as soon as this function call returns. 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` Specifies a far 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` Specifies the country code of the called party. If a value of zero is specified, a default is used by the implementation.

`lpCallParams` Specifies a far 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 3.1kHz 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.

TSPI function:

```
LONG TSPI_lineMakeCall(dwRequestID, hdLine, htCall, lphdCall,
    lpszDestAddress, dwCountryCode, lpCallParams)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdLine` Specifies the handle to the line on which the new call is to be originated.

`htCall` Specifies TAPI.DLL's handle to the new call. The service provider must save this and use it in all subsequent calls to the line event callback procedure reporting events on the call.

`lphdCall` Specifies a far pointer to a call handle. The service provider must fill this location with its handle for the call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error.

`lpszDestAddress` Specifies a far pointer to the destination address. This follows the standard dialable number format. This pointer may be specified as NULL for non-dialed addresses (as with a hot phone, which always automatically connects to a predefined number) or when all dialing will be performed using **TSPI_lineDial**.

`dwCountryCode` Specifies the country code of the called party. If a value of zero is specified, a default will be used by the implementation.

`lpCallParams` Specifies a far pointer to a **LINECALLPARAMS** structure. This structure allows TAPI.DLL to specify how it wants the call to be set up. If NULL is specified, a default 3.1kHz voice call is established, and an arbitrary origination address on the line is selected.

lineMonitorDigits

This function enables and disables the unbuffered detection of digits received on the call. Each time a digit of the specified digit mode(s) is detected, a message is sent to the application indicating which digit has been detected.

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.

TAPI function:

`LONG` **`lineMonitorDigits`**`(hCall, dwDigitModes)`

`hCall` Specifies a handle to the call on which digits are to be detected.

`dwDigitModes` Specifies the digit mode(s) that are to be monitored. If *dwDigitModes* is zero, digit monitoring is cancelled. This parameter can have multiple flags set, and uses the following LINEDIGITMODE_ constants: PULSE detect digits as audible clicks that are the result of rotary pulse sequences. Valid digits for pulse are '0' through '9'; DTMF detect digits as DTMF tones. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'; DTMFEND detect and provide application notification of DTMF down edges. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

TSPI function:

`LONG` **`TSPI_lineMonitorDigits`**`(hdCall, dwDigitModes)`

`hdCall` Specifies the handle to the call on which digits are to be detected.

`dwDigitModes` Specifies the digit mode(s) that are to be monitored. A *dwDigitModes* with a value of zero cancels digit monitoring.

lineMonitorMedia

This 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.
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 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. Since 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, since 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

TAPI function:

```
LONG lineMonitorMedia(hCall, dwMediaModes)
```

`hCall` Specifies a handle to the call.

`dwMediaModes` Specifies 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.

TSPI function:

```
LONG TSPI_lineMonitorMedia(hdCall, dwMediaModes)
```

`hdCall` Specifies the handle to the call for which media monitoring is to be sct.

`dwMediaModes` Specifies the media modes to be monitored. A value of zero for the *dwMediaModes* parameter cancels all media mode monitoring.

lineMonitorTones

This 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.
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

TAPI function:

`LONG` **`lineMonitorTones`**`(hCall, lpToneList, dwNumEntries)`

`hCall` Specifies a handle to the call on whose voice channel tones are to be monitored.

`lpToneList` Specifies 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` Specifies the number of entries in *lpToneList*. This parameter is ignored if *lpToneList* is NULL.

TSPI function:

`LONG` **`TSPI_lineMonitorTones`**`(hdCall, dwToneListID, lpToneList, dwNumEntries)`

`hdCall` Specifies the handle to the call for which tone detection is to be done.

`dwToneListID` Specifies the unique ID for this tone list. Several tone lists can be outstanding at once. The service provider must replace any old list having the same *dwToneListID* with the new tone list. If *lpToneList* is NULL, the tone list with *dwToneListID* is simply dropped. In any case, other tone lists with different *dwToneListIDs* are kept unchanged.

`lpToneList` Specifies 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 for the purpose of reporting 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. The service provider must copy the tone list into its own memory for later reference, rather than simply retaining the pointer into application memory.

`dwNumEntries` Specifies the number of entries in *lpToneList*. The *dwNumEntries* parameter is ignored if *lpToneList* is NULL. Note that this parameter is not validated by TAPI.DLL when this function is called.

lineNegotiateAPIVersion

This function allows an application to negotiate an API version to use.
Use **lineInitialize** 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 it 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.

TAPI function:

```
LONG lineNegotiateAPIVersion(hLineApp, dwDeviceID, dwAPILowVersion,
     dwAPIHighVersion, lpdwAPIVersion, lpExtensionID)
```

`hLineApp` Specifies the handle to the application's registration with TAPI.

`dwDeviceID` Specifies the line device to be queried.

`dwAPILowVersion` Specifies 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` Specifies 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` Specifies a far 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` Specifies a far 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.

TSPI function:

Most of the function is handled within TAPI.DLL, only the line extension ID is retrieved from the appropriate service provider by means of the function:

```
LONG TSPI_lineGetExtensionID(dwDeviceID, dwTSPIVersion, lpExtensionID)
```

`dwDeviceID` Specifies the line device to be queried.

`dwTSPIVersion` Specifies an interface version number that has already been negotiated for this device using **TSPI_lineNegotiateTSPIVersion**. This function operates according to the interface specification at this version level.

`lpExtensionID` Specifies a far pointer to a structure of type **LINEEXTENSIONID**. If the service provider supports provider-specific extensions it fills this structure with the extension ID of these extensions. If the service provider does not support extensions, it fills this structure with all zeros. (Therefore, a valid extension ID cannot consist of all zeros.)

lineNegotiateExtVersion

This 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.
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.

TAPI function:

```
LONG lineNegotiateExtVersion(hLineApp, dwDeviceID, dwAPIVersion,
    dwExtLowVersion, dwExtHighVersion, lpdwExtVersion)
```

`hLineApp` Specifies the handle to the application's registration with TAPI.

`dwDeviceID` Specifies the line device to be queried.

`dwAPIVersion` Specifies the API version number that was negotiated for the specified line device using **lineNegotiateAPIVersion**.

`dwExtLowVersion` Specifies 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` Specifies 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` Specifies a far 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*.

TSPI function:

```
LONG TSPI_lineNegotiateExtVersion(dwDeviceID, dwTSPIVersion, dwLowVersion,
    dwHighVersion, lpdwExtVersion)
```

`dwDeviceID` Identifies the line device for which interface version negotiation is to be performed.

`dwTSPIVersion` Specifies an interface version number that has already been negotiated for this device using **TSPI_lineNegotiateTSPIVersion**. This function operates according to the interface specification at this version level.

`dwLowVersion` Specifies the lowest extension version number under which TAPI.DLL or its client application is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. Note that this parameter is not validated by TAPI.DLL when this function is called.

`dwHighVersion` Specifies the highest extension version number under which TAPI.DLL or its client application is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. Note that this parameter is not validated by TAPI.DLL when this function is called.

`lpdwExtVersion` Specifies a far pointer to a DWORD. Upon a successful return from this function, the service provider fills this location with the highest extension version number, within the range requested by the caller, under which the service provider is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. If the requested range does not overlap the range supported by the service provider, the function returns an error.

lineOpen

This 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.
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 in TELEPHON.INI, an error 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).
Note that 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.
Note that 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.

TAPI function:

```
LONG lineOpen(hLineApp, dwDeviceID, lphLine, dwAPIVersion, dwExtVersion,
     dwCallbackInstance, dwPrivileges, dwMediaModes, lpCallParams)
```

`hLineApp` Specifies 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` Specifies a far 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` Specifies the API version number under which the application and Telephony API have agreed to operate. This number is obtained with **lineNegotiateAPIVersion**.

`dwExtVersion` Specifies 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` Specifies user-instance data passed back to the application's callback. This parameter is not interpreted by the Telephony API.

`dwPrivileges` Specifies the privilege the application wants for the calls it is notified for. This parameter can have multiple flags set, and it uses the following LINECALLPRIVILEGE_ constants: NONE the application wants to make only outbound calls; MONITOR the application only wants to monitor inbound and outbound calls; OWNER the application wants to own inbound calls of the types specified in *dwMediaModes* ; MONITOR + 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.

`dwMediaModes` Specifies the media mode(s) of interest to the application. The *dwMediaModes* 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).

`lpCallParams` Specifies a far 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.

TSPI function:

The first application that opens a line causes the conserned service provider to be called by TAPI.DLL with the function:

```
LONG TSPI_lineOpen(dwDeviceID, htLine, lphdLine, dwTSPIVersion,
    lpfnEventProc)
```

`dwDeviceID` Identifies the line device to be opened.

`htLine` Specifies TAPI.DLL's handle for the line device to be used in subsequent calls to the **LINEEVENT** callback procedure to identify the device.

`lphdLine` A far pointer to a HDRVLINE where the service provider fills in its handle for the line device.

`dwTSPIVersion` The TSPI version.

`lpfnEventProc` A far pointer to the **LINEEVENT** callback procedure supplied by TAPI.DLL that the service provider will call to report subsequent events on the line.

linePark

This function parks the specified call according to the specified park mode.
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*.

TAPI function:

```
LONG linePark(hCall, dwParkMode, lpszDirAddress, lpNonDirAddress)
```

`hCall` Specifies a handle to the call to be parked. The application must be an owner of the call.

`dwParkMode` Specifies 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: DIRECTED the application specifies at which address the call is to be parked in *lpszDirAddress*; NONDIRECTED this operation reports to the application where the call has been parked in *lpNonDirAddress*.

`lpszDirAddress` Specifies a far 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` Specifies a far 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.DLL for returning information.

TSPI function:

```
LONG TSPI_linePark(dwRequestID, hdCall, dwParkMode, lpszDirAddress,
    lpNonDirAddress)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call to be parked.

`dwParkMode` Specifies the park mode with which the call is to be parked.

`lpszDirAddress` Specifies a far pointer to NULL-terminated ASCII string that indicates the address where the call is to be parked when using directed park. The address is in dialable address format. This parameter is ignored for nondirected park.

`lpNonDirAddress` Specifies a far 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 is accounted for in the *dwStringSize*.

linePickup

This 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.
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**.
Once **linePickup** has been used to pick up the second call, **lineSwapHold** can be used to toggle between them. **lineDrop** 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.

TAPI function:

`LONG` **`linePickup`**`(hLine, dwAddressID, lphCall, lpszDestAddress, lpszGroupID)`

`hLine` Specifies a handle to the open line device on which a call is to be picked up.

`dwAddressID` Specifies the address on *hLine* at which the pickup is to be originated.

`lphCall` Specifies a far 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` Specifies a far 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` Specifies a far 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.

TSPI function:

`LONG` **`TSPI_linePickup`**`(dwRequestID, hdLine, dwAddressID, htCall, lphdCall, lpszDestAddress, lpszGroupID)`

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdLine` Specifies the handle to the line on which a call is to be picked up.

`dwAddressID` Specifies the address on *hdLine* at which the pickup is to be originated.

`htCall` Specifies TAPI.DLL's handle to the new call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call.

`lphdCall` Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the call. The service provider must fill this location with its handle for the call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error.

`lpszDestAddress` Specifies a far pointer to a NULL terminated ASCII string that contains the address whose call is to be picked up. The address is standard link format.

`lpszGroupID` Specifies a far pointer to a NULL-terminated ASCII string 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. It can also be NULL itself.

linePrepareAddToConference

This function prepares an existing conference call for the addition of another party.
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**.

TAPI function:

```
LONG linePrepareAddToConference(hConfCall, lphConsultCall, lpCallParams)
```

`hConfCall` Specifies a handle to a conference call. The application must be an owner of this call.

`lphConsultCall` Specifies a far 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` Specifies a far 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.

TSPI function:

```
LONG TSPI_linePrepareAddToConference(dwRequestID, hdConfCall,
     htConsultCall, lphdConsultCall, lpCallParams)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdConfCall` Specifies the handle to a conference call.

`htConsultCall` Specifies TAPI.DLL's handle to the new, temporary consultation call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the new call.

`lphdConsultCall` Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the new, temporary consultation call. The service provider must fill this location with its handle for the new call before this procedure returns. This handle is invalid if the function results in an error.

`lpCallParams` Specifies a far pointer to call parameters to be used when establishing the consultation call. This parameter will be set to NULL if no special call setup parameters are desired.

lineRedirect

This function redirects the specified offering call to the specified destination address.
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.

TAPI function:

`LONG` **`lineRedirect`**`(hCall, lpszDestAddress, dwCountryCode)`
`hCall` Specifies a handle to the call to be redirected. The application must be an owner of the call.
`lpszDestAddress` Specifies a far pointer to the destination address. This follows the standard dialable number format.
`dwCountryCode` Specifies 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.

TSPI function:

`LONG` **`TSPI_lineRedirect`**`(dwRequestID, hdCall, lpszDestAddress, dwCountryCode)`

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call to be redirected.

`lpszDestAddress` Specifies a far pointer to the destination address. This follows the standard link format.

`dwCountryCode` Specifies the country code of the party the call is redirected to. If a value of zero is specified, a default will be used by the implementation. Note that this parameter is not validated by TAPI.DLL when this function is called.

lineRegisterRequestRecipient

This function registers the invoking application as a recipient of requests for the specified request mode. A telephony-enabled application can request that a call be placed on its behalf by invoking **tapiRequestMakeCall**, **tapiRequestMediaCall**, and drop a media call with **tapiRequestDrop**. Additionally, other applications can request that information be logged with a given call. **tapiRequestMakeCall** and **tapiRequestMediaCall** requests are queued by TAPI.DLL, 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. (Note that **tapiRequestDrop** requests are passed directly to the recipient application by the **LINE_REQUEST** message; they are not queued, and are not retrieved using **lineGetRequest**.) The call-control application then interprets and executes the request. For media mode handling, the serving application may need to send Windows messages back to the original application that made the request. The **TAPI_REPLY** message is used for this purpose.

The recipient application is also automatically deregistered for all requests when it does a **lineShutdown**.

TAPI function:

```
LONG lineRegisterRequestRecipient(hLineApp, dwRegistrationInstance,
    dwRequestMode, bEnable)
```

`hLineApp` Specifies the application's usage handle for the line portion of TAPI.

`dwRegistrationInstance` Specifies 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` Specifies the type(s) of request for which the application registers. One or both bits may be set. This parameter uses the following LINEREQUESTMODE_ constants: MAKECALL a **tapiRequestMakeCall** request; MEDIACALL **tapiRequestMediaCall / tapiRequestDrop** request.

`bEnable` If TRUE, the application registers; if FALSE, the application deregisters for the specified request modes.

TSPI function:

None. The function is handled entirely by TAPI.DLL.

lineRemoveFromConference

This function removes the specified call from the conference call to which it currently belongs. The remaining calls in the conference call are unaffected.
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**.

TAPI function:

```
LONG lineRemoveFromConference(hCall)
```

`hCall` Specifies a handle to the call to be removed from the conference. The application must be an owner of this call.

TSPI function:

```
LONG TSPI_lineRemoveFromConference(dwRequestID, hdCall)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call to be removed from the conference.

lineSecureCall

This function secures the call from any interruptions or interference that may affect the call's media stream.
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 function **lineSecureCall** 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.

TAPI function:

```
LONG lineSecureCall(hCall)
```

`hCall` Specifies a handle to the call to be secured. The application must be an owner of the call.

TSPI function:

```
LONG TSPI_lineSecureCall(dwRequestID, hdCall)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call to be secured.

lineSendUserUserInfo

This function send user-to-user information to the remote party on the specified 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. Whenever user-to-user information arrives, 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.

TAPI function:

```
LONG lineSendUserUserInfo(hCall, lpsUserUserInfo, dwSize)
```

`hCall` Specifies a handle to the call on which to send user-to-user information. The application must be an owner of the call.

`lpsUserUserInfo` Specifies a far 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**).

`dwSize` Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*.

TSPI function:

```
LONG TSPI_lineSendUserUserInfo(dwRequestID, hdCall, lpsUserUserInfo,
    dwSize)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call on which to send user-to-user information.

`lpsUserUserInfo` Specifies a far pointer to a string containing the user-to-user information to be sent.

`dwSize` Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*.

lineSetAppSpecific

This operation enables an application to set the application-specific field of the specified call's call-information record. 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**.

TAPI function:

```
LONG lineSetAppSpecific(hCall, dwAppSpecific)
```

`hCall` Specifies a handle to the call whose application-specific field needs to be set. The application must be an owner of the call.

`dwAppSpecific` Specifies the new content of the **dwAppSpecific** field.

TSPI function:

```
LONG TSPI_lineSetAppSpecific(hdCall, dwAppSpecific)
```

`hdCall` Specifies the handle to the call whose application-specific field is to be set.

`dwAppSpecific` Specifies the new content of the **dwAppSpecific** field.

lineSetCallParams

This function allows an application to change bearer mode and/or the rate parameters of an existing call. 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.

TAPI function:

```
LONG lineSetCallParams(hCall, dwBearerMode, dwMinRate, dwMaxRate,
    lpDialParams)
```

`hCall` Specifies a handle to the call whose parameters are to be changed. The application must be an owner of the call.

`dwBearerMode` Specifies the new bearer mode for the call. This parameter can have only a single bit set. and it uses the LINEBEARERMODE_ constants.

`dwMinRate` Specifies 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` Specifies 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` Specifies a far 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.

TSPI function:

```
LONG TSPI_lineSetCallParams(dwRequestID, hdCall, dwBearerMode, dwMinRate,
    dwMaxRate, lpDialParams)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call whose parameters are to be changed.

`dwBearerMode` Specifies the new bearer mode for the call. The *dwBearerMode* parameter can have only one of the LINEBEARERMODE_ flags set.

`dwMinRate` Specifies a lower bound for the call's new data rate. TAPI.DLL is willing to accept a new rate as low as this one. Note that this parameter is not validated by TAPI.DLL when this function is called.

`dwMaxRate` Specifies an upper bound for the call's new data rate. This is the maximum data rate TAPI.DLL would like. Equal values for *dwMinRate* and *dwMaxRate* indicate that an exact data rate is required. Note that this parameter is not validated by TAPI.DLL when this function is called.

`lpDialParams` Specifies a far pointer to the new dial parameters for the call, of type **LINEDIALPARAMS**. If this parameter is NULL, it indicates that the call's current dialing parameters are to be used.

lineSetCallPrivilege

This operation sets the application's privilege to the specified privilege.
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.

TAPI function:

`LONG` **`lineSetCallPrivilege`**`(hCall, dwCallPrivilege)`

`hCall` Specifies a handle to the call whose privilege is to be set.

`dwCallPrivilege` Specifies 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: 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; 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.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineSetCurrentLocation

This operation sets the location used as the context for address translation.

TAPI function:

`LONG` **`lineSetCurrentLocation`**`(`*`hLineApp, dwLocation`*`)`

`hLineApp` Specifies the application handle returned by **lineInitialize**.

`dwLocation` Specifies a new value for the CurrentLocation entry in the [Locations] section of TELEPHON.INI. It must contain a valid permanent ID of a Location entry in the [Locations] section, as obtained from **lineTranslateCaps**. If it is valid, the CurrentLocation entry is updated.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineSetDevConfig

This 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.
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 wants to call the same destination again, this function **lineSetDevConfig** can be used to restore the configuration settings selected by the user. **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 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 interchangability 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.

TAPI function:

```
LONG lineSetDevConfig (dwDeviceID, lpDeviceConfig, dwSize, lpszDeviceClass)
```

`dwDeviceID` Specifies the line device to be configured.

`lpDeviceConfig` Specifies a far pointer to the opaque configuration data structure that was returned by **lineGetDevConfig** in the variable portion of the **VARSTRING** structure.

`dwSize` Specifies 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` Specifies a far 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.

TSPI function:

```
LONG TSPI_lineSetDevConfig (dwDeviceID, lpDeviceConfig, dwSize,
    lpszDeviceClass)
```

`dwDeviceID` Specifies the line device to be configured.

`lpDeviceConfig` Specifies a far pointer to the configuration data structure which had been returned in the variable portion of the **VARSTRING** structure by **TSPI_lineGetDevConfig**.

`dwSize` Specifies the number of bytes in the structure pointed to be *lpDeviceConfig*. This value will have been returned in the **dwStringSize** field in the **VARSTRING** structure returned by **TSPI_lineGetDevConfig**.

`lpszDeviceClass` Specifies a far pointer to a NULL-terminated ASCII string that specifies the device class of the device whose configuration is to be restored. Valid device class strings are the same as those specified for the **TSPI_lineGetID** function when it is applied to a "line" device (that is, when *dwSelect* has the value LINECALLSELECT_LINE).

lineSetMediaControl

This 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.
This 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.

TAPI function:

```
LONG lineSetMediaControl(hLine, dwAddressID, hCall, dwSelect, lpDigitList,
    dwDigitNumEntries, lpMediaList, dwMediaNumEntries, lpToneList,
    dwToneNumEntries, lpCallStateList, dwCallStateNumEntries)
```

`hLine` Specifies a handle to an open line device.

`dwAddressID` Specifies an address on the given open line device.

`hCall` Specifies a handle to a call. The application must be an owner of the call.

`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: LINE selects the specified line device. The *hLine* parameter must be a valid line handle; *hCall* and *dwAddressID* are ignored; ADDRES selects the specified address on the line. Both *hLine* and *dwAddressID* must be valid; *hCall* is ignored; CALL selects the specified call. *hCall* must be valid; *hLine* and *dwAddressID* are both ignored.

`lpDigitList` Specifies a far pointer to the array that contains the digits that are to trigger media control actions, of type **LINEMEDIACONTROLDIGIT**. Each time a digit listed 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` Specifies the number of entries in the *lpDigitList*.

`lpMediaList` Specifies a far pointer to an array with entries of type **LINEMEDIA-CONTROLMEDIA**. 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` Specifies the number of entries in *lpMediaList*.

`lpToneList` Specifies a far pointer to an array with entries of type **LINEMEDIA-CONTROLTONE**. 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` Specifies the number of entries in *lpToneList*.

`lpCallStateList` Specifies a far pointer to an array with entries are of type **LINEMEDIA-CONTROLCALLSTATE**. 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` Specifies the number of entries in *lpCallStateList*.

TSPI function:

```
LONG TSPI_lineSetMediaControl(hdLine, dwAddressID, hdCall, dwSelect,
     lpDigitList, dwDigitNumEntries, lpMediaList, dwMediaNumEntries,
     lpToneList, dwToneNumEntries, lpCallStateList, dwCallStateNumEntries)
```

lineSetMediaMode

This function sets the media mode(s) of the specified call in its **LINECALLINFO** structure.
This 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).

TAPI function:

```
LONG lineSetMediaMode(hCall, dwMediaModes)
```

`hCall` Specifies a handle to the call whose media mode is to be changed. The application must be an owner of the call.

`dwMediaModes` Specifies 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.

TSPI function:

```
LONG TSPI_lineSetMediaMode(hdCall, dwMediaMode)
```

`hdCall` Specifies the handle to the call undergoing a change in media mode.

`dwMediaMode` Specifies the new media mode(s) for the call. As long as the LINEMEDIAMODE_UNKNOWN media mode flag is set, multiple 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 just a small set of specified media modes. If the LINEMEDIAMODE_UNKNOWN flag is not set, only a single media mode can be specified.

lineSetNumRings

This function is used by an application to set the number of rings it wants an incoming call to ring prior to answering the call. 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.
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 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.

TAPI function:

`LONG` **`lineSetNumRings`**`(hLine, dwAddressID, dwNumRings)`

`hLine` Specifies a handle to the open line device.

`dwAddressID` Specifies an address on the line device.

`dwNumRings` Specifies the number of rings before a call should be answered in order to honor the toll-saver requests from all applications.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineSetStatusMessages

This function 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.
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. **lineSetStatusMessages** can be used to select which messages the application wants to receive. By default, address and line status reporting is disabled.

TAPI function:

`LONG` **`lineSetStatusMessages`**`(hLine, dwLineStates, dwAddressStates)`

`hLine` Specifies a handle to the line device.

`dwLineStates` Specifies a bit array that identifies for which line-device status changes a message is to be sent to the application.

`dwAddressStates` Specifies a bit array that identifies for which address status changes a message is to be sent to the application.

TSPI function:

LONG **TSPI_lineSetStatusMessages**(*hdLine, dwLineStates, dwAddressStates*)

lineSetTerminal

This function enables an application to specify which terminal information related to the specified line, address, or call is to be routed. **lineSetTerminal** can be used while calls are in progress on the line to allow an application to route these events to different devices as required.
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 function:

```
LONG lineSetTerminal(hLine, dwAddressID, hCall, dwSelect, dwTerminalModes,
     dwTerminalID, bEnable)
```

`hLine` Specifies a handle to an open line device.

`dwAddressID` Specifies an address on the given open line device.

`hCall` Specifies a handle to a 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.

`dwTerminalModes` Specifies the class(es) of low-level events to be routed to the given terminal.

`dwTerminalID` Specifies 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*.

TSPI function:

```
LONG TSPI_lineSetTerminal(dwRequestID, hdLine, dwAddressID, hdCall,
     dwSelect, dwTerminalModes, dwTerminalID, bEnable)
```

lineSetTollList

This function manipulates the toll list.

TAPI function:

```
LONG lineSetTollList(hLineApp, dwDeviceID, lpszAddressIn, dwTollListOption)
```

`hLineApp` Specifies the application handle returned by **lineInitialize**.

`dwDeviceID` Specifies 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` Specifies a far 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` Specifies the toll list operation to be performed. Only a single flag can be set. This parameter uses the following LINETOLLLISTOPTION_ constants: ADD causes the prefix contained within the string pointed to by *lpszAddressIn* to be added to the toll list for the current location; 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.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineSetupConference

This function sets up a conference call for the addition of the third party. **lineSetupConference** 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 **lineDrop** on it. When dropping a consultation call, the existing conference call typically transitions back to the *connected* state.

TAPI function:

```
LONG lineSetupConference(hCall, hLine, lphConfCall, lphConsultCall,
    dwNumParties, lpCallParams)
```

`hCall` Specifies the initial call that identifies the first party of a conference call. In some environments (as described in device capabilities), a call must exist in order 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.

`hLine` Specifies 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` Specifies a far 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.

`lphConsultCall` Specifies a far 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` Specifies the expected number of parties in the conference call. This number is passed to the service provider.

`lpCallParams` Specifies a far 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.

TSPI function:

```
LONG TSPI_lineSetupConference(dwRequestID, hdCall, hdLine, htConfCall,
    lphdConfCall, htConsultCall, lphdConsultCall, dwNumParties,
    lpCallParams)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the initial call that identifies the first party of a conference call.

`hdLine` Specifies the handle to the line device on which to originate the conference call if *hdCall* is NULL. The *hdLine* parameter is ignored if *hdCall* is non-NULL. The service provider reports which model it supports through the **setupConfNull** flag of the **LINEADDRESSCAPS** data structure.

`htConfCall` Specifies TAPI.DLL's handle to the new conference call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the new call.

`lphdConfCall` Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the newly created conference call. The service provider must fill this location with its handle for the new call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error.

`htConsultCall` Specifies TAPI.DLL's handle to the consultation call. When setting up a call for the addition of a new party, a new temporary call (consultation call) is automatically allocated. The service provider must save the *htConsultCall* and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the new consultation call.

`lphdConsultCall` Specifies a far pointer to an HDRVCALL representing the service provider's identifier for a call. When setting up a call for the addition of a new party, a new temporary call (consultation call) is automatically allocated. The service provider must fill this location with its handle for the new consultation call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error.

`dwNumParties` Specifies the expected number of parties in the conference call. The service provider is free to do with this number as it pleases. For example, the service provider can ignore it, or use it as a hint to allocate the right size conference bridge inside the switch. Note that this parameter is not validated by TAPI.DLL when this function is called.

`lpCallParams` Specifies a far pointer to call parameters to be used when establishing the consultation call. This parameter will be set to NULL if no special call setup parameters are desired and the service provider uses default parameters.

lineSetupTransfer

This 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*.
This 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
The application may also transfer calls in a single step, without having to deal with the intervening consultation call by using **lineBlindTransfer**.

TAPI function:

```
LONG lineSetupTransfer(hCall, lphConsultCall, lpCallParams)
```

`hCall` Specifies the handle of the call to be transferred. The application must be an owner of the call.

`lphConsultCall` Specifies a far 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. 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. 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` Specifies a far 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.

TSPI function:

```
LONG TSPI_lineSetupTransfer(dwRequestID, hdCall, htConsultCall,
    lphdConsultCall, lpCallParams)
```

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call to be transferred.

`htConsultCall` Specifies TAPI.DLL's handle to the new, temporary consultation call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the new consultation call.

`lphdConsultCall` Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the new consultation call. The service provider must fill this location with its handle for the new consultation call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error. When setting a call up for transfer, another call (a consultation call) is automatically allocated to enable the application (through TAPI.DLL) to dial the address (using **TSPI_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. This transfer procedure

may not be valid for some line devices. Instead of calling this procedure, TAPI.DLL may need to unhold an existing held call (using **TSPI_lineUnhold**) to identify the destination of the transfer. It may also be necessary to set up the consultation call as an entirely new call using **TSPI_lineMakeCall**, to the destination of the transfer. The **transferHeld** and **transferMake** flags in the **LINEADDRESSCAPS** data structure report what model the service provider uses.

`lpCallParams` Specifies a far 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 (the service provider uses defaults).

lineShutdown

This function shuts down the application's usage of the line abstraction of API.
If this function is called when the application has lines open or calls active, the call handles are deleted and TAPI.DLL 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 cancelled.

TAPI function:

`LONG` **`lineShutdown`**`(hLineApp)`

`hLineApp` Specifies the application's usage handle for the line API.

TSPI function:

The last application that calls lineShutdown causes TAPI.DLL and all service providers providers to be unloaded. This results in the function TSPI_providerShutdown being called in each service provider.

lineSwapHold

This function swaps the specified active call with the specified call on consultation hold.
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.

TAPI function:

`LONG` **`lineSwapHold`**`(hActiveCall, hHeldCall)`

`hActiveCall` Specifies the handle to the active call. The application must be an owner of the call.

`hHeldCall` Specifies the handle to the consultation call. The application must be an owner of the call.

TSPI function:

`LONG` **`TSPI_lineSwapHold`**`(dwRequestID, hdActiveCall, hdHeldCall)`

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdActiveCall` Specifies the handle to the call to be swapped with the call on consultation hold.

`hdHeldCall` Specifies the handle to the consultation call.

lineTranslateAddress

This operation translates the specified address into another format.

TAPI function:

```
LONG lineTranslateAddress(hLineApp, dwDeviceID, dwAPIVersion,
    lpszAddressIn, dwCard, dwTranslateOptions, lpTranslateOutput)
```

`hLineApp` Specifies the application handle returned by **lineInitialize**.

`dwDeviceID` Specifies 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 version of TAPI negotiated by **lineNegotiateAPIVersion**.

`lpszAddressIn` Specifies a far pointer to a NULL-terminated ASCII string containing the address from which the prefix information is to be extracted for processing. 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` Specifies 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 of TELEPHON.INI (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 TELEPHON.INI 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` Specifies the associated operations to be performed prior to the translation of the address into a dialable string. This parameter uses the following LINETRANSLATEOPTION_ constants: CARDOVERRIDE If this bit is set, *dwCard* specifies the permanent ID of a Card entry in the [Cards] section of TELEPHON.INI (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 TELEPHON.INI to be modified; the override applies only to the current translation operation. The *dwCard* field is ignored if the CARDOVERRIDE bit is not set.

`lpTranslateOutput` Specifies a far pointer to an application-allocated memory area to contain the output of the translation operation, of type **LINETRANSLATEOUTPUT**. Prior to calling **lineTranslateAdress**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI.DLL for returning information.

TSPI function:

None. The function is handled entirely within TAPI.DLL.

lineUncompleteCall

This function is used to cancel the specified call competion request on the specified line.

TAPI function:

`LONG` **`lineUncompleteCall`**`(hLine, dwCompletionID)`

`hLine` Specifies a handle to the line device on which a call completion is to be canceled.

`dwCompletionID` Specifies the completion ID for the request that is to be canceled.

TSPI function:

`LONG` **`TSPI_lineUncompleteCall`**`(dwRequestID, hdLine, dwCompletionID)`

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdLine` Specifies the handle to the line on which a call completion is to be canceled.

`dwCompletionID` Specifies the completion ID for the request that is to be canceled. This parameter is not validated by TAPI.DLL when this function is called.

lineUnhold

This function retrieves the specified held call.
This operation works only for calls on hard hold (calls placed on hold using **lineHold**).

TAPI function:

`LONG` **`lineUnhold`**`(hCall)`

`hCall` Specifies the handle to the call to be retrieved. The application must be an owner of this call.

TSPI function:

`LONG` **`TSPI_lineUnhold`**`(dwRequestID, hdCall)`

`dwRequestID` Specifies the identifier of the asynchronous request.

`hdCall` Specifies the handle to the call to be retrieved.

lineUnpark

This function retrieves the call parked at the specified address and returns a call handle for it.

TAPI function:

LONG **lineUnpark**(hLine, dwAddressID, lphCall, lpszDestAddress)

hLine Specifies a handle to the open line device on which a call is to unparked.

dwAddressID Specifies the address on *hLine* at which the unpark is to be originated.

lphCall Specifies a far 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 Specifies a far pointer to a NULL-terminated character buffer that contains the address where the call is parked. The address is in standard dialable address format.

TSPI function:

LONG **TSPI_lineUnpark**(dwRequestID, hdLine, dwAddressID, htCall, lphdCall, lpszDestAddress)

dwRequestID Specifies the identifier of the asynchronous request.

hdLine Specifies the handle to the line on which a call is to be unparked.

dwAddressID Specifies the address on *hdLine* at which the unpark is to be originated. This parameter is not validated by TAPI.DLL when this function is called.

htCall Specifies TAPI.DLL's handle to the new unparked call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call.

lphdCall Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the new unparked call. The service provider must fill this location with its handle for the call before this procedure returns. This handle is invalid if the function results in an error.

lpszDestAddress Specifies a far pointer to a NULL terminated ASCII string that contains the address where the call is parked. The address is in dialable address format.

Messages

All of the TAPI messages are sent to the application's callback function. When an application uses **lineInitialize**, it specifies this callback function by passing its entry point as a parameter.
The callback message always contains a handle to the relevant object (phone device, line device or call). The callback function can determine the type of the handle from the message that was passed to the callback. The actual parameters that are passed to the application's callback function are described for each of the messages.
Most TAPI messages sent to an application's callback function are a result of TSPI messages sent from a service provider to TAPI.DLL's callback function. Actually, TAPI.DLL contains two callback functions: one for asynchronous completion messages and one for line event messages.

Application's callback function profile (TAPI):

```
void CALLBACK CallbackFunc(dwDevice, dwMsg, dwCallbackInstance,
     dwParam1, dwParam2, dwParam3)
```

`CallbackFunc` is a placeholder for the application-supplied function name.

`dwDevice` Specified a handle to either a line device, phone device, or a call associated with the callback. The nature of this handle (line handle, phone handle, or call handle) can be determined by the context provided by *dwMsg*.

`dwMsg` Specifies a line, call, or phone device message.

`dwCallbackInstance` Specifies the user instance data specified at API initialization time.

`dwParam1`, `dwParam2` and `dwParam3` Specify parameters for the message's contents.

TAPI.DLL's callback functions profiles (TSPI):

```
void CALLBACK Completion_Proc(dwRequestID, lResult)
```

`dwRequestID` Specifies the ID that was passed in the original request that the service provider executed asynchronously.

`lResult` Specifies the outcome of the operation. This can be zero to indicate success or a negative number to indicate an error. The possible specific error values that may result from a function are the same for asynchronous or synchronous execution.

```
void CALLBACK Line_Event(htLine, htCall, dwMsg, dwParam1, dwParam2,
     dwParam3)
```

`htLine` Specifies TAPI.DLL's handle for the line on which the event occurred.

`htCall` Specifies TAPI.DLL's handle for the call on which the event occurred if this is a call-related event. For line-related events where there is no call, this parameter should be set to zero.

`dwMsg` Specifies what kind of event is being reported. Interpretation of the other parameters is done in different ways according to the context indicated by *dwMsg*.

`dwParam1`, `dwParam2` and `dwParam3` Specify parameters for the message's contents.

LINE_ADDRESSSTATE

This 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. This 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.

TAPI message parameters:

`dwDevice` Specifies a handle to the line device.

`dwParam1` Specifies the address ID of the address that changed status.

`dwParam2` Specifies the address state that changed. This parameter uses the LINEADDRESSSTATE_ constants.

`dwParam3` Unused.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Unused

`dwMsg` The value **LINE_ADDRESSSTATE**

`dwParam1` Specifies the address ID of the address that changed status.

`dwParam2` Specifies the address state that changed.

`dwParam3` Unused.

LINE_CALLINFO

This message is sent when the call information about the specified call has changed. The application can invoke **lineGetCallInfo** to determine the current call information.
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**.

TAPI message parameters:

`dwDevice` Specifies a handle to the call.

`dwParam1` Specifies the call information item that has changed. This parameter uses the LINECALL-INFOSTATE_ constants.

`dwParam2, dwParam3` Unused.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Specifies TAPI.DLL's opaque object handle to the call device.

`dwMsg` The value **LINE_CALLINFO**

`dwParam1` Specifies the call information item that has changed.

`dwParam2, dwParam3` Unused.

LINE_CALLSTATE

This message is sent whenever 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.
This message is sent to any application that has a handle for the call. Note that 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.

TAPI message parameters:

`dwDevice` Specifies a handle to the call.

`dwParam1` Specifies the new call state. This parameter uses the following LINECALLSTATE_ contstants.

`dwParam2` Specifies call-state-dependent information.

`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: MONITOR and OWNER

TSPI line event message parameters (1):

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Unused

`dwMsg` The value **LINE_NEWCALL**

`dwParam1` Specifies the service provider's opaque handle for the call, of type HDRVCALL. TAPI.DLL will pass this value as the *hdCall* parameter to identify the call in subsequent procedures it invokes to operate on the call.

`dwParam2` A far pointer pointing to a HTAPICALL. TAPI.DLL writes TAPI.DLL's opaque handle for the call to the indicated location. The service provider must save this value and pass it as the *htCall* parameter to identify the call in subsequent events it reports for the call.

`dwParam3` Unused.

TSPI line event message parameters (2):

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Specifies TAPI.DLL's opaque object handle to the call device.

`dwMsg` The value **LINE_CALLSTATE**

`dwParam1` Specifies the new call state.

`dwParam2` Specifies call-state-dependent information.

`dwParam3` The media mode of the call, as far as it is known. This is a combination of LINEMEDIA-MODE_ constants. If the service provider does not know the media mode, it should include the UNKNOWN bit together with all media modes currently being monitored for.

LINE_CLOSE

This 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.
This 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.

TAPI message parameters:

`dwDevice` Specifies a handle to the line device that was closed. This handle is no longer valid.
`dwParam1, dwParam2, dwParam3` Unused.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.
`htCall` Unused
`dwMsg` The value LINE_CLOSE
`dwParam1, dwParam2, dwParam3` Unused.

LINE_DEVSPECIFIC

This message is sent to notify the application about device-specific events occurring a line, address or call. The meaning of the message and the interpretation of the parameters is device specific.
This message is used by a service provider in conjunction with the **lineDevSpecific** function. Its meaning is device specific.

TAPI message parameters:

`dwDevice` Specifies a handle to either a line device or call. This is device specific.
`dwParam1, dwParam2,dwParam3` Device specific.

TSPI line event message parameters (1):

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.
`htCall` Specifies TAPI.DLL's opaque object handle to the call device.
`dwMsg` The value **LINE_CALLDEVSPECIFIC**
`dwParam1, dwParam2,dwParam3` Device specific.

TSPI line event message parameters (2):

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.
`htCall` Unused
`dwMsg` The value **LINE_DEVSPECIFIC**
`dwParam1, dwParam2,dwParam3` Device specific.

LINE_DEVSPECIFICFEATURE

This 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 is device specific. This message is used by a service provider in conjunction with the **lineDevSpecificFeature** function

TAPI message parameters:

`dwDevice` Specifies a handle to either a line device or call. This is device specific.

`dwParam1, dwParam2,dwParam3` Device specific.

TSPI line event message parameters (1):

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Specifies TAPI.DLL's opaque object handle to the call device.

`dwMsg` The value **LINE_CALLDEVSPECIFICFEATURE**

`dwParam1, dwParam2,dwParam3` Device specific.

TSPI line event message parameters (2):

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Unused

`dwMsg` The value **LINE_DEVSPECIFICFEATURE**

`dwParam1, dwParam2,dwParam3` Device specific.

LINE_GATHERDIGITS

This message is sent when the current buffered digit-gathering request has terminated or is canceled. It is only sent to the application that initiated the digit gathering on the call using **lineGatherDigits**.

TAPI message parameters:

`dwDevice` Specifies a handle to the call.

`dwParam1` Specifies the reason why digit gathering was terminated. This parameter uses the following LINEGATHERTERM_ constants: BUFFERFULL the requested number of digits has been gathered; TERMDIGIT one of the termination digits matched a received digit; FIRSTTIMEOUT the first digit timeout expired; INTERTIMEOUT the inter-digit timeout expired; CANCEL the request was canceled by this application, by another application, or because the call terminated.

`dwParam2, dwParam3` Unused.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Specifies TAPI.DLL's opaque object handle to the call device.

`dwMsg` The value **LINE_GATHERDIGITS**

`dwParam1` Specifies the reason why digit gathering was terminated.

`dwParam2` The *dwEndToEndID* that was specified in the original **TSPI_lineGatherDigits** request for which this is the final result.

`dwParam3` Unused.

LINE_GENERATE

This message is sent to notify the application that the current digit or tone generation has terminated. Note that 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.
This message is only sent to the application that requested the digit or tone generation.

TAPI message parameters:

`dwDevice` Specifies a handle to the call.

`dwParam1` Specifies the reason why digit or tone generation was terminated. This parameter uses the following LINEGENERATETERM_ constants: DONE the requested number of digits have been generated, or the requested tones have been generated for the requested duration; CANCEL the digit or tone generation request was canceled by this application, by another application, or because the call terminated.

`dwParam2, dwParam3` Unused.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Specifies TAPI.DLL's opaque object handle to the call device.

`dwMsg` The value **LINE_GENERATE**

`dwParam1` Specifies the reason why digit or tone generation was terminated.

`dwParam2` The *dwEndToEndID* parameter that was specified in the original **TSPI_lineGenerateDigits** or **TSPI_lineGenerateTone** request for which this is the final result.

`dwParam3` Unused.

LINE_LINEDEVSTATE

This 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.
The sending of this 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.

TAPI message parameters:

`dwDevice` Specifies a handle to the line device. This parameter is NULL when *dwParam1* is LINEDEVSTATE_REINIT.

`dwParam1` Specifies the line device status item that has changed. The parameter *dwParam1* can have multiple flags set, and it uses the LINEDEVSTATE_ constants.

`dwParam2` The interpretation of this parameter depends on the value of *dwParam1*. If *dwParam1* is LINEDEVSTATE_RINGING, *dwParam2* contains the ring mode with which the switch instructs the line to ring. Valid ring modes are numbers in the range one to **dwNumRingModes**, where **dwNumRingModes** is a line device capability.

`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.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Unused

`dwMsg` The value **LINE_LINEDEVSTATE**

`dwParam1` Specifies the line device status item that has changed. This parameter can have multiple flags set and uses the LINEDEVSTATE_ constants.

`dwParam2` The interpretation of this parameter depends on the value of *dwParam1*. If *dwParam1* is LINEDEVSTATE_RINGING, *dwParam2* contains the ring mode with which the switch instructs the line to ring. Valid ring modes are numbers in the range one to **dwNumRingModes**, where **dwNumRingModes** is a line device capability.

`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.

LINE_MONITORDIGITS

This message is sent whenever a digit is detected. The sending of this message is controlled with the **lineMonitorDigits** function.

TAPI messsage parameters:

`dwDevice` Specifies a handle to the call.

`dwParam1` The low-order byte contains the last digit received in ASCII.

`dwParam2` Specifies the digit mode that was detected. This parameter uses the following LINEDIGITMODE_ constants: PULSE detect digits as audible clicks that are the result of rotary pulse sequences. Valid digits for pulse are '0' through '9'; DTMF detect digits as DTMF tones. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'; DTMFEND detect and provide application notification of DTMF down edges.

`dwParam3` Unused.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Specifies TAPI.DLL's opaque object handle to the call device.

`dwMsg` The value **LINE_MONITORDIGITS**

`dwParam1` The low-order byte contains the last digit received in ASCII.

`dwParam2` Specifies the digit mode that was detected.

`dwParam3` Unused.

LINE_MONITORMEDIA

This message is sent whenever a change in the call's media mode is detected. The sending of this message is controlled with the **lineMonitorMedia** function.

TAPI message parameters:

`dwDevice` Specifies a handle to the call.

`dwParam1` Specifies the new media mode. This parameter uses the LINEMEDIAMODE_ constants.

`dwParam2, dwParam3` Unused.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Specifies TAPI.DLL's opaque object handle to the call device.

`dwMsg` The value **LINE_MONITORMEDIA**

`dwParam1` Specifies the new media mode.

`dwParam2, dwParam3` Unused.

LINE_MONITORTONE

This message is sent whenever a tone is detected. The sending of this message is controlled with the **lineMonitorTones** function.

TAPI message parameters:

`dwDevice` Specifies a handle to the call.

`dwParam1` Specifies the application-specific **dwAppSpecific** field of the **LINEMONITORTONE** structure for the tone that was detected.

`dwParam2, dwParam3` Unused.

TSPI line event message parameters:

`htLine` Specifies TAPI.DLL's opaque object handle to the line device.

`htCall` Specifies TAPI.DLL's opaque object handle to the call device.

`dwMsg` The value **LINE_MONITORTONE**

`dwParam1` Specifies the application-specific field **dwAppSpecific** of the **LINEMONITORTONE** structure for the tone that was detected.

`dwParam2` Specifies the **dwToneListID** of the tone list containing a matching tone description.

`dwParam3` Unused.

LINE_REPLY

This message is sent to an application's callback function to report the results of function calls that completed asynchronously.
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 this message carries the success or failure indication. Possible errors are the same as those defined by the corresponding function. This message cannot be disabled.

TAPI message parameters:

`dwDevice` Not used.

`dwParam1` Specifies the request ID for which this is the reply.

`dwParam2` Specifies 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.

TSPI line event message parameters:

None. The asynchronous completion callback function is used by the service provider.

LINE_REQUEST

This message is sent to an application's callback to report the arrival of a new Assisted Telephony request from another application.
This message is sent to the highest priority application that has registered for the corresponding request mode. This callback 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.

TAPI message parameters:

`dwDevice` Not used.

`dwCallbackInstance` Specifies the registration instance of the application specified on **lineRegister-RequestRecipient**.

`dwParam1` Specifies the request mode of the newly pending request. This parameter uses the following LINEREQUESTMODE_ constants: MAKECALL a **tapiRequestMakeCall** request; MEDIACALL a **tapiRequestMediaCall** request; DROP a **tapiRequestDrop** 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.

TSPI line event message parameters:

None. The TAPI message is not a result of a service provider message.

Constants

LINEADDRCAPFLAGS_ Constants

The LINEADDRCAPFLAGS_ bit-flag constants are used in the **dwAddrCapFlags** field of the **LINEADDRESSCAPS** data structure to describe various Boolean address capabilities.

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**.
PICKUPGROUPID	Specifies whether a group ID is required for call pickup.
SECURE	Specifies whether calls on this address can be made secure at call-setup time.
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.
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.
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").
ORIGOFFHOOK	Specifies whether the originating party's phone can automatically be taken offhook when making calls.
DESTOFFHOOK	Specifies whether the called party's phone can automatically be forced offhook when making calls.
FWDCONSULT	Specifies whether call forwarding involves the establishment of a consultation call.
SETUPCONFNULL	Specifies whether setting up a conference call starts out with an initial call (FALSE) or with no initial call (TRUE).
AUTORECONNECT	Specifies whether dropping a consultation call automatically reconnects to the call on consultation hold. TRUE if reconnect happens automatically; otherwise, FALSE.
COMPLETIONID	Specifies whether the completion IDs returned by **lineCompleteCall** are useful and unique. TRUE if useful; otherwise, FALSE.
TRANSFERHELD	Specifies whether a hard-held call can be transferred. Often, only calls on consultation hold can be transferred.
TRANSFERMAKE	Specifies whether an entirely new call can be established for use as a consultation call on transfer.
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.
CONFERENCEMAKE	Specifies whether an entirely new call can be established for use as a consultation call (to add) on conference.
PARTIALDIAL	Specifies whether partial dialing is available.
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.
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.
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.
ACCEPTTOALERT	TRUE if an offering call must be accepted using **lineAccept** in order to start alerting the users at both ends of the call; otherwise, FALSE. This is typically only used with ISDN.
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.
PICKUPCALLWAIT	TRUE if **linePickup** can be used to pickup a call detected by the user as a call waiting call; otherwise, FALSE.

LINEADDRESSSHARING_ Constants

The LINEADDRESSSHARING_ bit-flag constants describe various ways an address can be shared between lines.

PRIVATE	The address is private to the user's line; it is not assigned to any other station.
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.
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.
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.
MONITORED	This is an address whose idle/busy status is made available to this line.

LINEADDRESSSTATE_ Constants

The LINEADDRESSSTATE_ bit-flag constants describe various address status items.

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.
DEVSPECIFIC	The device-specific item of the address status has changed.
INUSEZERO	The address has changed to idle (it is not in use by any stations).

INUSEONE	The address has changed from idle or in use by many bridged stations to being in use by just one station.
INUSEMANY	The monitored or bridged address has changed from being in use by one station to being in use by more than one station.
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 **LINEADDRESS-STATUS** structure. The application should check all three of these fields when it receives a **LINE_ADDRESSSTATE** (numCalls) message.
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.
TERMINALS	The terminal settings for the address have changed.

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.

VOICE	This is a regular 3.1kHz analog voice grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.
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.
MULTIUSE	The multi-use mode defined by ISDN.
DATA	The unrestricted data transfer on the call. The data rate is specified separately.
ALTSPEECHDATA	The alternate transfer of speech or unrestricted data on the same call (ISDN).
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).

LINECALLINFOSTATE_ Constants

The LINECALLINFOSTATE_ bit-flag constants describe various call information items about which an application may be notified in the **LINE_CALLINFO** message.

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.
DEVSPECIFIC	The device-specific field of the call-information record.

BEARERMODE	The bearer mode field of the call-information record.
RATE	The rate field of the call-information record.
MEDIAMODE	The media-mode field of the call-information record.
APPSPECIFIC	The application-specific field of the call-information record.
CALLID	The call ID field of the call-information record.
RELATEDCALLID	The related call ID field of the call-information record.
ORIGIN	The origin field of the call-information record.
REASON	The reason field of the call-information record.
COMPLETIONID	The completion ID field of the call-information record.
NUMOWNERINCR	The number of owner field in the call-information record was increased.
NUMOWNERDECR	The number of owner field in the call-information record was decreased.
NUMMONITORS	The number of monitors field in the call-information record has changed.
TRUNK	The trunk field of the call-information record.
CALLERID	One of the callerID-related fields of the call-information record.
CALLEDID	One of the calledID-related fields of the call-information record.
CONNECTEDID	One of the cconnectedID-related fields of the call-information record.
REDIRECTIONID	One of the redirectionID-related fields of the call-information record.
REDIRECTINGID	One of the redirectingID-related fields of the call-information record.
DISPLAY	The display field of the call-information record.
USERUSERINFO	The user-to-user information of the call-information record.
HIGHLEVELCOMP	The high level compatibility field of the call-information record.
LOWLEVELCOMP	The low level compatibility field of the call-information record.
CHARGINGINFO	The charging information of the call-information record.
TERMINAL	The terminal mode information of the call-information record.
DIALPARAMS	The dial parameters of the call-information record.
MONITORMODES	One or more of the digit, tone, or media monitoring fields in the call-information record.

LINECALLPARTYID_ Constants

The LINECALLPARTYID_ bit-flag constants describe the nature of the information available about the parties involved in a call.

BLOCKED	Party ID information is not available because it has been blocked by the remote party.
OUTOFAREA	Caller ID information for the call is not available since it is not propagated all the way by the network.
NAME	Party ID information consists of the party's name (as, for example, from a directory kept inside the switch).
ADDRESS	Party ID information consists of the party's address in either canonical address format or dialable address format.
PARTIAL	Party ID information is valid but it is limited to partial information only.
UNKNOWN	Party ID information is currently unknown but may become known later.
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.

LINECALLSTATE_ Constants

The LINECALLSTATE_ bit-flag constants describe the call states a call can be in.

IDLE	The call is idle—no call exists.
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 since alerting is done by the switch instructing the line to ring. It does not affect any call states.
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.
DIALTONE	The call is receiving a dial tone from the switch, which means that the switch is ready to receive a dialed number.
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.
RINGBACK	The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.
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.
SPECIALINFO	Special information is sent by the network. Special information is typically sent when the destination cannot be reached.
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.
PROCEEDING	Dialing has completed and the call is proceeding through the switch or telephone network.
ONHOLD	The call is on hold by the switch.
CONFERENCED	The call is currently a member of a multi-party conference call.
ONHOLDPENDCONF	The call is currently on hold while it is being added to a conference.
ONHOLDPENDTRANSFER	The call is currently on hold awaiting transfer to another number.
DISCONNECTED	The remote party has disconnected from the call.

LINEDEVCAPFLAGS_ Constants

The LINEDEVCAPFLAGS_ bit-flag constants are a collection of Booleans describing various line device capabilities.

CROSSADDRCONF	Specifies whether calls on different addresses on this line can be conferenced.
HIGHLEVCOMP	Specifies whether high-level compatibility information elements are supported on this line.
LOWLEVCOMP	Specifies whether low-level compatibility information elements are supported on this line.
MEDIACONTROL	Specifies whether media-control operations are available for calls at this line.
MULTIPLEADDR	Specifies whether **lineMakeCall** or **lineDial** are able to deal with multiple addresses at once (as for inverse multiplexing).
CLOSEDROP	Specifies what happens when an open line is closed while the application has calls active on the line. If TRUE, a **lineClose** will drop (that is, clear) all calls on the line if the application is the sole owner of those calls. Knowing the setting of this flag ahead of time makes it possible for the application to display an [OK]/[Cancel] dialog box for the user, warning that the active call will be lost. If CLOSEDROP is FALSE, a **lineClose** will not automatically drop any calls still active on the line if the service provider knows that some other device 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 service provider should set the flag to FALSE, as the offhook phone will automatically keep the call active even after the computer powers down.
DIALBILLING DIALQUIET 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.

LINEDEVSTATE_ Constants

The LINEDEVSTATE_ bit-flag constants describe various line status events.

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.
RINGING	The switch tells the line to alert the user.
CONNECTED	The line was previously disconnected and is now connected to TAPI.
DISCONNECTED	This line was previously connected and is now disconnected from TAPI.
MSGWAITON	The "message waiting" indicator is turned on.
MSGWAITOFF	The "message waiting" indicator is turned off.
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.
OUTOFSERVICE	The line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.
MAINTENANCE	Maintenance is being performed on the line at the switch. TAPI cannot be used

	to operate on the line device.
OPEN	The line has been opened by another application.
CLOSE	The line has been closed by another application.
NUMCALLS	The number of calls on the line device has changed.
NUMCOMPLETIONS	The number of outstanding call completions on the line device has changed.
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).
ROAMMODE	The roam mode of the line device has changed.
BATTERY	The battery level has changed significantly (cellular).
SIGNAL	The signal level has changed significantly (cellular).
DEVSPECIFIC	The line's device-specific information has changed.
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.
LOCK	The locked status of the line device has changed. (For more information, refer to the description of the LINEDEVSTATUSFLAGS_LOCKED bit in the LINEDEVSTATUSFLAGS_ constants.)

LINEDEVSTATUSFLAGS_ Constants

The LINEDEVSTATUSFLAGS_ bit-flag constants describe a collection of Boolean line device status items.

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.
MSGWAIT	Indicates whether the line has a message waiting. If TRUE, a message is waiting; if FALSE, no message is waiting.
INSERVICE	Indicates whether the line is in service. If TRUE, the line is in service; if FALSE, the line is out of service.
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.

LINEFORWARDMODE_ Constants

The LINEFORWARDMODE_ bit-flag constants describe the conditions under which calls to an address can be forwarded.

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.

UNCONDINTERNAL	Forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.
UNCONDEXTERNAL	Forward all external calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.
UNCONDSPECIFIC	Unconditionally forward all calls that originated at a specified address (selective call forwarding).
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.
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.
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.
BUSYSPECIFIC	Forward on busy all calls that originated at a specified address (selective call forwarding).
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.
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.
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.
NOANSWSPECIFIC	Forward on no answer all calls that originated at a specified address (selective call forwarding).
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.
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.
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.
BUSYNASPECIFIC	Forward on busy/no answer all calls that originated at a specified address (selective call forwarding).

LINEMEDIAMODE_ Constants

The LINEMEDIAMODE_ constants describe media modes (the data type of a media stream) on calls.

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.
INTERACTIVEVOICE	The presence of voice energy on the call, and the call is treated as an interactive call with humans on both ends.
AUTOMATEDVOICE	The presence of voice energy on the call and the voice is locally handled by an automated application.
DATAMODEM	A data modem session on the call.
G3FAX	A group 3 fax is being sent or received over the call.
TDD	A TDD (Telephony Devices for the Deaf) session on the call.
G4FAX	A group 4 fax is being sent or received over the call.
DIGITALDATA	Digital data is being sent or received over the call.
TELETEX	A teletex session on the call. Teletex is one of the telematic services.
VIDEOTEX	A videotex session on the call. Videotex is one the telematic services.
TELEX	A telex session on the call. Telex is one the telematic services.
MIXED	A mixed session on the call. Mixed is one the ISDN telematic services.
ADSI	An ADSI (Analog Display Services Interface) session on the call.

LINETRANSLATERESULT_ Constants

The LINETRANSLATERESULT_ bit-flag constants describe various results of an address translation.

CANONICAL	Indicates that the input string was in valid canonical format.
INTERNATIONAL	Indicates that 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).
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).
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).
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.
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.
DIALBILLING	Indicates that the returned address contains a "$".
DIALQUIET	Indicates that the returned address contain a "@".
DIALDIALTONE	Indicates that the returned address contains a "W".
DIALPROMPT	Indicates that the returned address contains a "?".