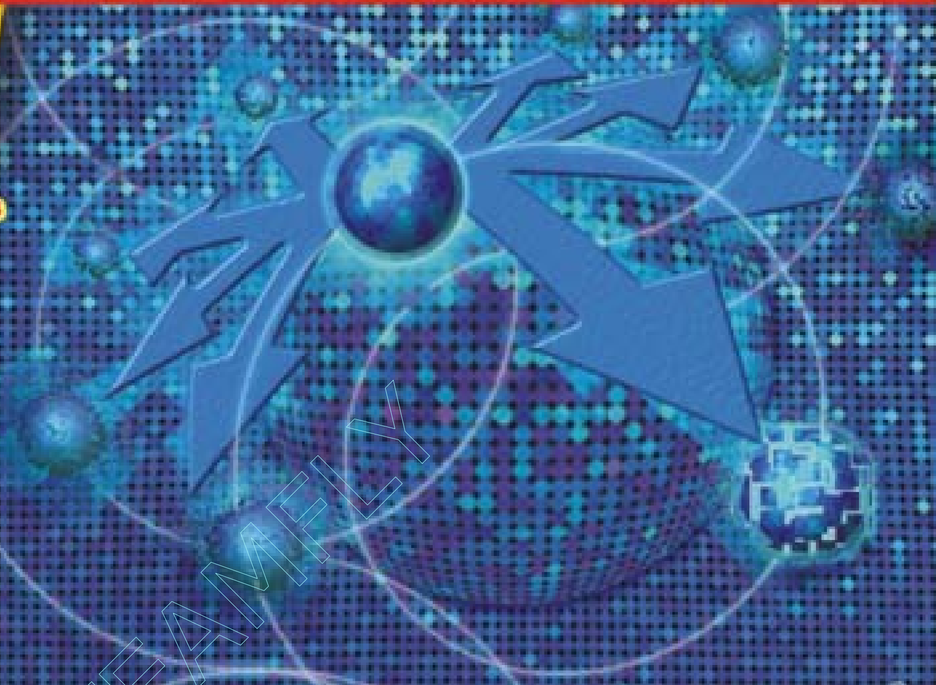
WORDWARE DELPHI DEVELOPER'S LIBRARY



Find out how to use Windows Sockets and the Telephony Application Programming Interface to add communications functionality to Windows applications built with Delphi.

Learn about basic Winsock and TAPI functions and how to write useful internet/intranet and telephony applications.

The Tomes of Delphi®

Basic 32-Bit Communications Programming

Companion CD-ROM Included

Alan C. Moore and John C. Penman

Technical review by Gary Frerking, president of TurboPower, and Chad Z. Hower

The Tomes of Delphi: Basic 32-Bit Communications Programming

Alan C. Moore
and
John C. Penman

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Moore, Alan C., 1943-.
The Tomes of Delphi : basic 32-bit communications programming / by Alan C. Moore and John C. Penman.
p. cm.
Includes bibliographical references and index.
ISBN 1-55622-752-3 (paperback)
1. Computer software—Development. 2. Delphi (Computer file). 3. Telecommunication systems. I. Penman, John C. II. Title.

QA76.76.D47 M665 2002
005.1--dc21 2002011
CIP

2320 Los Rios Boulevard
Plano, Texas 75074

Printed in the United States of America

ISBN 1-55622-752-3
10 9 8 7 6 5 4 3 2 1
0210

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

Dedications

To Ann, with all my love.

Alan C. Moore

To the memory of my dear mum, Marie Chisholm Penman, who passed away on March 11, 2001.

John C. Penman

Contents

Part 2: TAPI

Acknowledgments

Writing a book like this is a major endeavor. I want to take this opportunity to thank some of the many people who helped make it possible. First, let me thank my wife, Ann, and daughter, Treenah, for their support and patience during the many hours I spent in front of a computer screen coding and writing. My colleagues at Kentucky State University have also been very supportive, especially my new chairperson, Dr. Barbara Buck, who provided much encouragement for my writing.

There are several people and one organization that had a great deal to do with my getting involved with TAPI in the first place. The organization is Project JEDI, which produced the translation of the TAPI header file for use in Delphi. The pioneering work of the original translators, Alexander Staubo and Brad Choate, was followed by the excellent new translation by Marcel van Brakel, with contributions from Rudy Velthuis and myself.

The TAPI portion of this book is based to some extent on a series of articles I wrote in *Delphi Informant Magazine* beginning in the late 1990s. Thanks to my good friend Jerry Coffey, the editor of *Delphi Informant Magazine*, for his continued encouragement to explore and write about TAPI. Thanks also to Major Ken Kyler, with whom I wrote the first three articles. Ken provided me with my first introduction to the world of TAPI. I would be remiss if I did not acknowledge my current co-author, John Penman. In the process of writing this book, we have read each other's text in some detail. Working with John on this book has been delightful from the start. Finally, let me acknowledge my excellent technical editor, Gary Frerking, president of TurboPower. He was extremely helpful in identifying portions of the text that were not clear and code that needed further work.

Before closing, I want to acknowledge the importance of my guru and spiritual teacher, the late Chogyam Trungpa, Rinpoche. The meditative disciplines he introduced to me and so many others have helped make my life more full and productive.

Alan C. Moore

As with any programming project, there are team players, project leaders, and technical staff. In this context, Alan and I are project leaders who have written this book, but without the team players and technical staff, there wouldn't be a book for us to write and you to read and hopefully assimilate some useful knowledge. So, it is in this vein that I would like to thank the team players for their contribution to making this book a reality. First, a special thanks must go to Marcel van Brakel, a former JEDI knight of Project JEDI (www.jedi.org), who gave some of his valuable time to test and debug all of the Winsock examples, as well as provide constructive criticism and suggestions for the chapters. I would also like to thank Chad Z. Hower for undertaking the role of technical editor for the Winsock chapters, which he carried out so ably. To those two guys, thanks a million!

I would also like to thank Alan C. Moore for his encouragement and wit during the time we worked together on the book. You will be amazed to know that we have never met in person, but we forged an excellent friendship through our electronic collaboration on this book. Perhaps we will collaborate on another!

I would like to thank Jim Hill, Wes Beckwith, and the hard-working staff at Wordware Publishing for their unfailing patience in spite of numerous missed milestones.

To end on a personal note, I would like to express heartfelt thanks to my dad for his unstinting and uncomplaining support for me while I was on contract in Scotland during the last 18 months. Thanks for being a great dad.

Finally, I must thank my dear wife, Jocie, and my two children, David and Diana, for their loving support during the development of this tome.

John C. Penman

Introduction

Reliable communications using computers has been important for a long time, starting with DOS bulletin boards and the early days of the Internet. In this book, we will provide an introduction to two of the essential communications technologies, Windows Sockets (Winsock), the backbone of the Internet on the Windows platform, and the Telephony Application Programming Interface (TAPI).

We will provide a complete introduction to Winsock and basic TAPI. We had originally planned on covering many of the other Internet technologies and the entire TAPI, but discovered that the material was too extensive to do justice to any of these technologies. We plan to write another book dealing with advanced communications programming in which we will cover the more difficult and newer topics. Nevertheless, this work should provide all that you will need to write useful Internet/Intranet or telephony applications. The advanced book will build on this foundation and provide the means for going beyond basic functionality.

This book is organized into two parts. Part I, written by John C. Penman, is a complete introduction to Winsock programming. Chapter 1 provides an introduction to this technology and a description of the Winsock-related chapters that follow. Part II, written by Alan C. Moore, is a complete introduction to basic TAPI programming. Chapter 7 provides an introduction to this technology and a description of the TAPI-related chapters that follow. As in other volumes in Wordware Publishing's Tomes of Delphi series, most chapters include introductory sections on the various technologies, a complete reference to functions, structures, and constants, and Delphi code examples.

The book concludes with three appendices providing a glossary of essential communications terms, information about error handling in Winsock and TAPI, and printed and Internet resources that provide additional information and programming materials.

Let's begin the journey!

Part I

Internet/Intranet Programming with Winsock

by John C. Penman

Chapter 1

The Winsock API

Introduction

In this chapter, we'll outline the development of the Internet and the transport protocols that underpin it. We'll review the evolution of the Winsock Application Programming Interface (API) from its origins. We will also examine the Winsock 1.1 and 2 architectures, with particular emphasis on Winsock 2.

In the world of Windows, Winsock provides the crucial foundation upon which all Internet applications run. Without Winsock, there would be no web browsers, no file transfer, and none of the e-mail applications that we take so much for granted in today's Windows environment. Technologies like DCOM and n-tier database systems would be difficult to implement.

Winsock is an API that is an integral part of Microsoft's Windows Open Systems Architecture (WOSA), which we'll discuss later in this chapter, as well as in the second half of the book dealing with TAPI. Let's start with the history of the genesis of the Internet to the present.

In the Beginning

Nowadays, it's easy to forget that the genesis of the Internet arose as a need for a dependable and robust communication network for military and government computers in the United States of America. In response to this need, in 1969 the Defense Advanced Research Projects Agency (DARPA) sponsored an experimental network called Advanced Research Projects Agency Network (ARPANET).

Before the birth of ARPANET, for one computer to communicate with another on a network, both machines had to come from the same vendor. We call this arrangement a *homogeneous network*. In contrast, ARPANET, a collection of different computers linked together, was a *heterogeneous network*.

As ARPANET developed, it became popular for connected institutions to accomplish daily tasks such as e-mail and file transfer. In 1975, ARPANET became operational. However, as you might have already guessed, research into network protocols continued. Network protocols that developed early in the life of ARPANET evolved into a set of network protocols called the Transmission Control Protocol/Internet Protocol (TCP/IP) suite. The *TCP/IP protocol suite* became a Military Standard in 1983, which made it mandatory for all computers on ARPANET to use TCP/IP.

NOTE: For brevity, we use TCP/IP as a shorthand for TCP/IP suite.

In 1983, ARPANET split into two networks: MILNET for unclassified military use and ARPANET, which was the smaller of the two, for experimental research. These networks became known as the Internet.

NOTE: The meaning of the Internet is just a collection of smaller networks to form a large network. Therefore, we can use the generic term *internet* to refer to a network of smaller networks that is not the Internet.

The Internet expanded further when DARPA invited more universities to use the Internet for research and communications. In the early 1980s, however, university computer sites were using the Berkeley Software Distribution (BSD) UNIX, a variant of UNIX that did not have TCP/IP. DARPA funded Bolt Beranek and Newman, Inc. to implement TCP/IP in BSD UNIX. Thus, TCP/IP became an intimate part of UNIX.

Although TCP/IP became an important communications provider in BSD UNIX, it was still difficult to develop network applications. Programmers at the University of Berkeley created an abstract layer to sit on top of TCP/IP, which became known as the *Sockets layer*. The version of BSD UNIX that incorporated the Sockets layer for the first time was 4.2BSD, which was released in August 1983.

The Sockets layer made it easier and quicker to develop and maintain network applications. The Sockets layer became a catalyst for the creation of network applications, which further fueled the expansion of the Internet. With the expansion of the Internet, TCP/IP became the network protocol of choice.

The following properties of TCP/IP explain the rapid acceptance of TCP/IP:

- It is vendor independent, meaning an open standard.
- It is a standard implementation on every computer from PCs to supercomputers.
- It is used in local area networks (LANs) and wide area networks (WANs).
- It is used by commercial entities, government agencies, and universities.

The Internet's rapid growth (and its continued growth) owes much to the development of the Hypertext Transfer Protocol (HTTP) that has provided the underpinnings for the World Wide Web. Rightly or wrongly, the ordinary man and woman on the street now sees the World Wide Web as the Internet. Internet protocols like HTTP, FTP, SMTP, and POP3 are high-level protocols that operate seamlessly on top of the network protocols collectively known as the TCP/IP protocol suite, or just TCP/IP. We'll describe briefly the network protocols that constitute the TCP/IP protocol suite in the next section.

Network Protocols

TCP/IP is a suite of network protocols upon which higher-level protocols, such as FTP, HTTP, SMTP, and POP3, operate. This suite comprises the two major protocols (TCP and IP) and a family of other protocols. We enumerate these as follows:

- **Transmission Control Protocol** (TCP) is a connection-based protocol that provides a stable, full duplex byte stream for an application. Applications like FTP and HTTP use this protocol.
- **User Datagram Protocol** (UDP) is a connectionless protocol that provides unreliable delivery of datagrams. (Note: Do not confuse "unreliable" with quality in this context. Unreliable refers to the possibility that some datagrams may not arrive at their destination, become duplicated, or arrive out of sequence.) IP Multicast applications use this protocol.
- **Internet Control Message Protocol** (ICMP) is a protocol that handles error and control information between hosts. Applications like ping and traceroute use this protocol.
- **Internet Protocol** (IP) is a protocol that provides packet delivery for TCP, UDP, and ICMP.
- **Address Resolution Protocol** (ARP) is a protocol that maps an Internet address into a hardware address.
- **Reverse Address Resolution Protocol** (RARP) is a protocol that maps a hardware address into an Internet address.

Fortunately, the BSD Sockets layer insulated the programmer from these protocols and, with some exceptions, most network applications did not need to know the intimate details of TCP/IP.

The OSI Network Model 1

In 1977, the International Organization for Standardization (ISO) created a reference schema for networking computers together. This networking model is a guide, not a specification, for the construction of any network. This guide, Open System Interconnection (OSI), states that a network should provide seven layers, as explained in Figure 1-1.

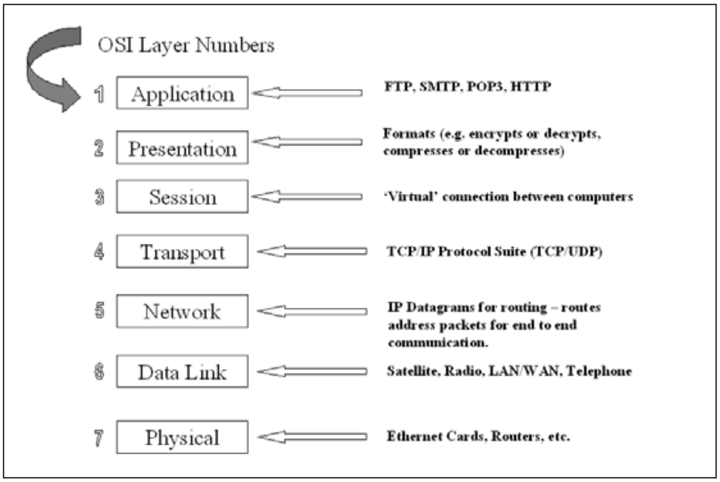


Figure 1-1

If we map TCP/IP using the OSI network model, we get the following simplified diagram in Figure 1-2.

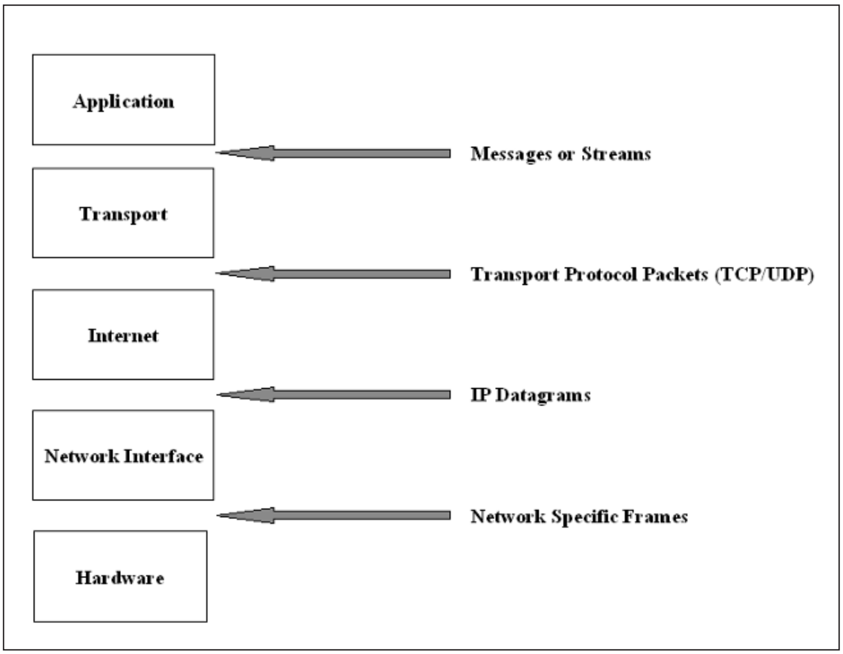


Figure 1-2

When a network application sends data to its peer across the network, the data is sent down through the layers to the data link and back up again through the layers on the peer's side. The following diagram makes this clear.

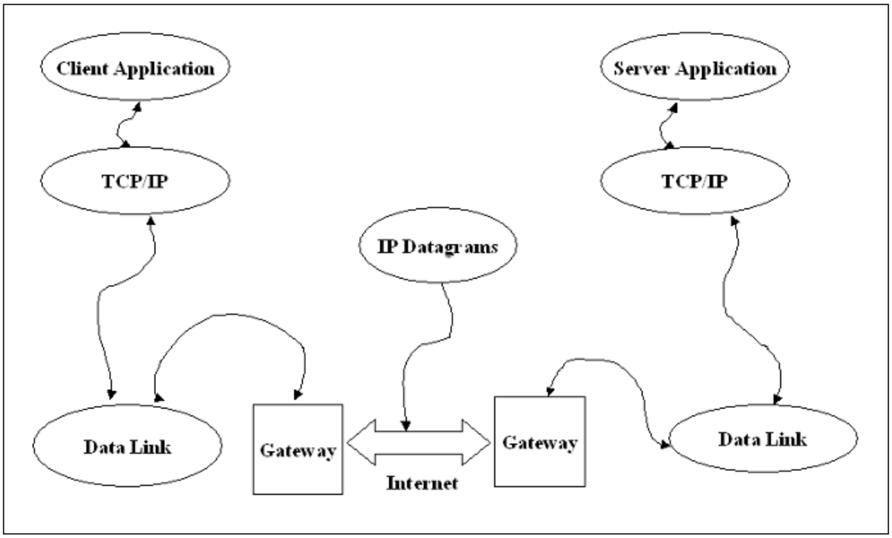


Figure 1-3

Before Winsock

In the early years of the Internet, computers that used the Internet were of the mainframe and minicomputer pedigree. Attached to these computers were dumb terminals that were used for e-mail, file transfer, and network news. It was natural, therefore, that when PCs appeared on the scene, there was some demand for PCs to connect as "superior dumb terminals" to the Internet. In response to this demand, developers ported BSD Sockets to the various DOS platforms, such as MS-DOS and DR-DOS. Unfortunately, vendors developed their own brand of TCP/IP stacks that were not completely compatible with each other. This meant, of course, that network application developers had to develop different versions of their applications to run on different stacks. This proliferation of applications to run on different stacks quickly became a maintenance nightmare. This problem continued in the Windows environment when Windows 3.0 and 3.1 appeared in the early 1990s.

Evolution of Winsock

Winsock to the rescue! Development of network-aware applications became so problematic that those leaders in the Windows communications industry organized a "Birds of a Feather" (BOF) conference at a prestigious trade show in 1991. At the end of the conference, delegates set up a committee to investigate the creation of a standard API for TCP/IP for Windows. This led to a specification that became Windows Sockets. The specification used much of BSD Sockets as its foundation. Windows 3.1 was a "cooperative" multitasking operating system, which relied on applications to yield quickly to avoid tying up resources such as the screen and mouse. Therefore, any Winsock application that blocked (for example, when waiting for data to arrive on a recv() function) would freeze the operating system, thus preventing any other application from running. To get around this major difficulty, the specification included modifications and enhancements that would permit a Winsock application to run asynchronously, avoiding a total freeze.

For example, the specification included functions such as WSAAsyncGetHostByAddr() and WSAAsyncGetHostByName(), which are asynchronous versions of the gethostbyaddr() and gethostbyname() functions, respectively. (We will examine the concept of blocking, non-blocking, and asynchronous operations later in the book.)

The first version of Winsock (1.0) appeared in June 1992. After some revision, another version (Winsock 1.1) appeared in January 1993. With the introduction of Winsock, network applications proliferated, making interfacing with the Internet easier than before.

One implementation of Winsock that soon became very common was Trumpet. Its developers took advantage of the Windows Dynamic Link Library (DLL) technology to house the Winsock API.

Some of the benefits of using Winsock include the following:

- Provides an open standard
- Provides application source code portability
- Supports dynamic linking

Since its inception, Winsock 1.1 has exceeded all expectations. However, the API focuses on TCP/IP to the exclusion of other protocol suites. This was a deliberate and strategic decision to encourage vendors of TCP/IP stacks to use the Windows Sockets specification in the early years. It worked!

Winsock is now the networking API of choice for all Windows platforms. Windows Sockets Version 2 addresses the need to use protocol suites other than TCP/IP. The Winsock 2 API can handle disparate protocol suites like DecNet, IPX/SPX, ATM, and many more. We call this capability multiple protocol support, or simply *protocol independence*. This degree of flexibility permits the development of generic network services. For example, an application could use a different protocol to perform one task and another for a different task. Although Winsock 2 adds new, flexible, and powerful features to the original API, the API is backward compatible with Version 1.1. This means that existing network applications developed for Winsock 1.1 can run without change under Winsock 2.

The Winsock Architecture

Winsock 1.1

As you have no doubt already deduced, the main difference between the two Winsock versions we're discussing is that Winsock 1.1 uses the TCP/IP protocol suite exclusively, whereas Winsock 2 supports other protocols, such as AppleTalk, IPX/SPX, and many others, as well as TCP/IP. Figure 1-4 shows the simple interaction between a Winsock 1.1 application with the WINSOCK.DLL or WSOCK.DLL. Because of its support for multiple protocols, the architecture of Winsock 2 is necessarily more complex.

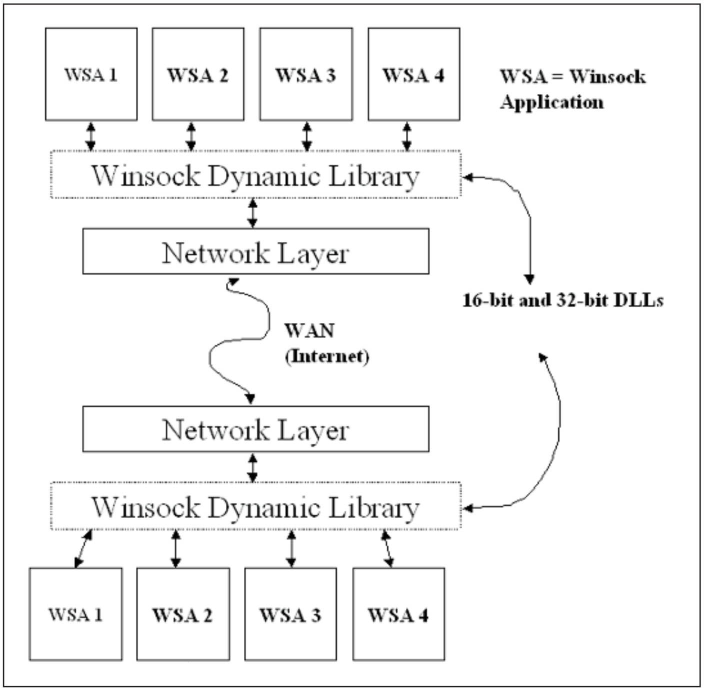


Figure 1-4

Winsock 2

Winsock 2 follows the Windows Open Systems Architecture (WOSA) model. In the WOSA model, the structure has two distinct parts; one is the API for application developers, and the other is the Service Provider Interface (SPI) for protocol stack and name space service providers. This two-tier arrangement allows Winsock to provide multiple protocol support, while using the same API. For example, an application that uses the ATM protocol will use the same API as the application that uses TCP/IP.

To make this clearer, take the scenario of a Winsock 2 server application that has several installed protocols (for example, ATM, TCP/IP, IPX/SPX, and AppleTalk). Because the server has access to each of these protocols, it can transparently service requests from different clients that are using any of the supported protocols.

The concept of the Service Provider Interface allows different vendors to install their own brand of transport protocol, such as NetWare's IPX/SPX. In addition to providing (transport) protocol independence, the Winsock 2 architecture also provides name space independence. Like the transport protocols, the SPI allows vendors to install their name space providers, which provides resolution of hosts, services, and protocols. The Domain Name System (DNS) that we use for resolving hosts on TCP/IP is an example of a name space provider. NetWare's Service Advertisement Protocol (SAP) is another. This capability enables any application to select the name space provider most suited to its needs. Figure 1-5 displays the Winsock 2 architecture. We'll discuss more details on protocol and name space independence in the next section.

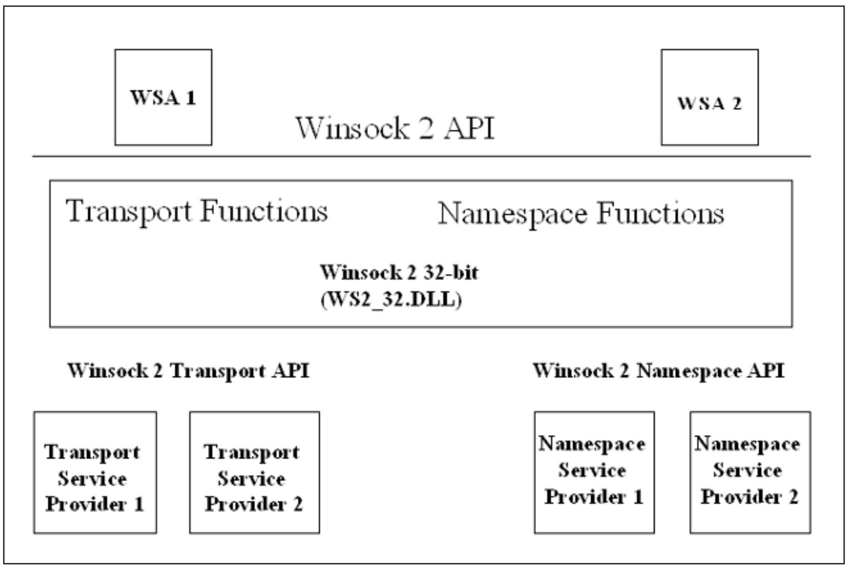


Figure 1-5

New Features of Winsock

Although we'll examine most of these new functions in detail in the chapters to come, we'll complete our overview of Winsock by enumerating these new features briefly.

Multiple Protocol Support

Like BSD Sockets before it, Winsock 2 provides simultaneous multiple protocol support. Winsock 2 has functions to allow an application to be protocol independent.

Name Space Independence

Name registration is a process that associates a protocol-specific address with a user-friendly name. For example, users find it easier to remember the address of Wordware Publishing, which is wordware.com, than a numeric address like 150.09.23.78. To make this possible, we use the Domain Name System (DNS) on TCP/IP to resolve host names to their IP addresses and vice versa.

There are three different types of name spaces: static, persistent, and dynamic. DNS is a static service, which is the least flexible. It is not possible to register a new name from Winsock using DNS. Dynamic name space, on the other hand, allows registration on the fly. An example of a persistent name space is Netware Directory Service (NDS).

Service Advertising Protocol (SAP) is a protocol for announcing dynamic name space changes on NDS.

Unlike Winsock 1.1, Winsock 2 can support multiple independent name space services in addition to the familiar DNS for TCP/IP.

Scatter and Gather

The support for "scatter and gather" is similar to the vectored I/O as supported by BSD Sockets.

Overlapped I/O

Winsock 2 uses the overlapped I/O model from the Win32 API. We will explain these functions in Chapter 5, "Communications."

Quality of Service

With the increasing use of multimedia applications that require a varying and often large bandwidth along with demanding timing requirements, the use of Quality of Service has become an important tool to manage network traffic. We will not discuss this tool, as it is beyond the scope of this book.

Multipoint and Multicast

Although Winsock 1.1 provides basic support for IP Multicast, Winsock 2 provides additional APIs that extend support for protocol-independent multipoint and multicast datagram transmission.

Conditional Acceptance

Winsock 2 provides the capability to examine the attributes of a connection request before accepting or rejecting the request. Using a callback function, WSAAccept() captures the attributes, such as caller's address, QOS information, and any connect data. After processing the data gleaned from the connection request, the application calls WSAAccept() again to accept, defer, or reject the request.

Connect and Disconnect Data

The new functions that support this feature are WSAAccept(), WSARecv-Disconnect(), WSASendDisconnect(), and WSAConnect().

Socket Sharing

Winsock 2 provides a means of sharing sockets between processes (but not between threads). The new function that provides this feature is WSADuplicate-Socket(). A process that requires sharing an existing socket does so through existing interprocess mechanisms like DDE, OLE, and others. However, the data itself is not shared, and each process needs to create its own event objects via calls to WSACreateEvent().

Protocol-specific Addition

Although Winsock 2 provides a consistent API, some protocols require additional data structures that are specific to a particular protocol. For example, ATM has extra data structures and special constants for its protocol. Although our focus is on the TCP/IP protocols, we have provided Delphi Interface units that translate the C headers containing the data structures for some of these protocols, such as AppleTalk, ATM, NETBIOS, ISO TP4, IPX/SPX, and BANYAN VINES.

Socket Groups

Winsock 2 introduces the concept of socket groups. An application can create a set of sockets with each socket dedicated to a specific task. However, in the current version (2.2), this feature is not yet supported, so we will not discuss it.

Summary

In this chapter, we covered the origins of the Internet, which led to the establishment of TCP/IP as the protocol suite of choice. We reviewed the evolution of Winsock from BSD Sockets and briefly covered the Winsock 2 architecture and its new features. To simplify coverage of the Winsock 2 API in the following chapters, the functions are grouped by the following topics:

Table 1-1: Function groups

Topic	Chapter
Starting and closing Winsock	Chapter 2
Winsock error handling	Chapter 2
Winsock 1.1 resolution	Chapter 3
Winsock 2 resolution	Chapter 4
Communications	Chapter 5
Network events	Chapter 5
Socket options	Chapter 6

For the majority of these functions, we'll demonstrate their usage with example code.

NOTE: These APIs are in the Winsock2.pas file on the companion CD that comes with this book. This file should be on a path visible to Delphi. By convention, you should put the Winsock2.pas file in the directory \Delphi 5\Lib.

Chapter 2

Winsock Fundamentals

In the last chapter, we provided a brief overview of the origins of the Internet and examined the evolution of BSD Sockets and the technology that gave birth to the Internet and provided the basis for Window's Internet technology, Windows Sockets.

In this chapter, we'll learn how to write a simple Winsock application that essentially does nothing useful. However, it does demonstrate how to load and unload Winsock correctly. We'll also learn how to detect Winsock errors properly.

Starting and Closing Winsock

In this chapter, we'll build a simple application that demonstrates the two most fundamental functions in the Winsock stable, WSAStartUp() and WSACleanup(). Without exception, your application must always call WSAStartUp() before calling any other Winsock API function. If you neglect this essential step, your application will fail, sometimes in spectacular fashion. Similarly, when your application ends, it should always call WSACleanup().

At invocation, WSAStartup() performs several essential tasks, as follows:

- Loads Winsock into memory
- Registers the calling application
- Allocates resources for the calling application
- Obtains the implementation details for Winsock

You can use the implementation details returned by WSAStartup() to determine if the version of Winsock is compatible with the version requested by the calling application. Ideally, any application should run using any version of Winsock. Winsock 1.1 applications can run unchanged using Winsock 2 because Winsock 2 seamlessly maps the Winsock 1.1 functions to their equivalents in Winsock 2.

To maintain this backward compatibility, WSAStartup() performs a negotiation phase with the calling application. In this phase, the Winsock DLL and the calling application negotiate the highest version that they both can support.

If Winsock supports the version requested by the application, the call succeeds and Winsock returns the highest version that it supports. In other words, if a Winsock 1.1 application makes a request to load Winsock 1.1, and if Winsock 2 is present, the application will work with Winsock 2 because it supports all versions up to 2, including 1.1.

This negotiation phase allows Winsock and the application to support a range of Winsock versions. Table 2-1 shows the range of Winsock versions that an application can use.

Table 2-1: Different versions of Winsock

App Version	DLL Version	Highest Version Expected	Expected Version	Highest Version Supported	End Result
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0, 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0, 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0, 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	Application fails
1.0	1.1	1.0	---	---	WSAVERNOTSUPPORTED
1.0, 1.1	1.0, 1.1	1.1	1.1	1.1	use 1.1
1.1, 2.0	1.1	2.0	1.1	1.1	use 1.1
2.0	2.0	2.0	2.0	2.0	use 2.0

It is only necessary for an application to call WSAStartup() and WSACleanup() once. Sometimes, though, an application may call WSAStartup() more than once. The golden rule is to make certain that the number of calls to WSAStartup() matches the number of calls to WSACleanup(). For example, if an application calls WSAStartup() three times, it must call WSACleanup() three times. That is, the first two calls to WSACleanup() do nothing except decrement an internal counter in Winsock; the final call to WSACleanup() for the task frees any resources.

Unlike Winsock 1.1 (which only supports one provider), the architecture of Winsock 2 supports multiple providers, which we will discuss in Chapter 4.

function WSAStartup Winsock2.pas

Syntax

WSAStartup(wVersionRequired: WORD; var lpWSAData: TWSAData): Integer; stdcall;

Description

This function initializes the Winsock DLL, registers the calling application, and allocates resources. It allows the application to specify the minimum version of Winsock it requires. The function also returns implementation information that the calling application should examine for version compatibility. After successful invocation of WSAStartup(), the application can call other Winsock functions.

Parameters

wVersionRequired: The highest version that the calling application requires. The high-order byte specifies the minor version and the low-order byte the major version. Under Windows 95, the highest version that is supported is 1.1. At the time of publication, the current version is 2.2. Table 2-2 presents which version of Winsock is available for all Windows operating systems.

Table 2-2: Winsock versions for all Windows platforms

Operating System	Winsock Version
Windows 3.1	1.1
Windows 95	1.1 (2.2) See Tip
Windows 98	2.2
Windows Millennium	2.2
Windows NT 4.0	2.2
Windows XP	2.2

TIP: If you belong to that unique tribe of developers that still uses Win95 as a development platform, and you want to develop Winsock 2 applications for Windows 95, you will have to upgrade Winsock 1.1. The upgrade is available from the Microsoft web site (www.microsoft.com).

wsData: This is a placeholder for the WSAData record that contains implementation details for Winsock. When we call WSAStartUp(), the function populates the WSAData record, which is defined in Winsock2.pas as follows:

```
WSAData = record
    wVersion: WORD;
    wHighVersion: WORD;
    szDescription: array [0..WSADESCRIPTION_LEN] of Char;
    szSystemStatus: array [0..WSASYS_STATUS_LEN] of Char;
    iMaxSockets: Word;
    iMaxUdpDg: Word;
    lpVendorInfo: PChar;
    end;

LPWSADATA = ^WSAData;
TWsaData = WSAData;
PWsaData = LPWSADATA;
```

Table 2-3 describes these fields of the WSAData data structure.

Table 2-3: Values for the main members of the WSAData structure

Member	Meaning
wVersion	The version of the Windows Sockets specification that the Windows Sockets DLL expects the calling application to use
wHighVersion	The highest version of the Windows Sockets specification that this DLL can support (also encoded as above). Normally this will be the same as wVersion.
szDescription	A NULL-terminated ASCII string into which the Windows Sockets DLL copies a description of the Windows Sockets implementation. The text may be up to 256 characters in length and contain any characters except control and formatting characters. The information in this field is often used by an application to provide a status message.
szSystemStatus	A NULL-terminated ASCII string into which the Windows Sockets DLL copies relevant status or configuration information. The Windows Sockets DLL should use this field only if the information might be useful to the user or support staff; it should not be considered as an extension of the szDescription field.
iMaxSockets	This field is retained for backward compatibility but should be ignored for version 2 and later, as no single value can be appropriate for all underlying service providers.
iMaxUdpDg	This value should be ignored for version 2 and onward. It is retained for backward compatibility with Windows Sockets specification 1.1 but should not be used when developing new applications. For the actual maximum message size specific to a particular Windows Sockets service provider and socket type, applications should use getsockopt() to retrieve the value of option SO_MAX_MSG_SIZE after a socket has been created.
lpVendorInfo	This value should be ignored for version 2 and onward. It is retained for backward compatibility with Windows Sockets specification 1.1. Applications needing to access vendor-specific configuration information should use getsockopt() to retrieve the value of option PVD_CONFIG. The definition of this value (if utilized) is beyond the scope of this specification.

Return Value

If successful, WSAStartup() will return zero. As we'll see when we cover other Winsock functions, WSAStartup() is the exception to the rule in that it does not return a Winsock error that we can use to determine the cause of that error. Since WSAStartup() is a function that initializes the Winsock DLL, which includes the WSAGetLastError() function to report Winsock-specific errors, it cannot call WSAGetLastError() because the DLL is not loaded. It is a conundrum like the proverbial chicken and egg problem. Therefore, to test for the success or failure to initialize Winsock, we just check for the return value of zero. Listing 2-1 demonstrates how to check the return value from WSAStartup().

Returning to the WSAData data structure, as far as programming Winsock applications goes, the most important fields that you should always read or check are *wVersion* and *wHighVersion*.

The WSAData structure in Winsock 2 no longer necessarily applies to a single vendor's stack. This means that Winsock 2 applications should ignore *iMaxSockets*, *iMaxUdpDg*, and *lpVendorInfo*, as these are irrelevant. However, you can retrieve provider-specific information by calling the getsockopt() function. We'll discuss this function briefly in Chapter 6, "Socket Options."

See Also

getsockopt, send, sendto, WSACleanup

Example

Listing 2-1 (program EX21 on the companion CD) shows how to load Winsock using WSAStartup() and how to verify version compatibility. It also shows how to close a Winsock application properly using WSACleanup().

function WSACleanup   Winsock2.pas

Syntax

WSACleanup: Integer; stdcall;

Description

This function unloads the Winsock DLL. A call to WSACleanup() will cancel the following operations: blocking and asynchronous calls, overlapped send and receive operations, and close and free any open sockets. Please note that any data pending may be lost.

Parameters

None

Return Value

If successful, the function will return a value of zero. Otherwise, the function returns a value of SOCKET_ERROR. To retrieve information about the error, call the WSAGetLastError() function. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, and WSAEINPROGRESS.

See Appendix B for a detailed description of the error codes.

See Also

closesocket, shutdown, WSAStartup

Example

Listing 2-1 shows how to load and unload the Winsock DLL by calling WSAStartup() and WSACleanup(), respectively.

Listing 2-l: Loading and unloading Winsock

```
{
  Example EX21 demonstrates how to load and unload Winsock correctly.
  It also demonstrates how to call different versions of Winsock. In this
  example,  the program expects an input of 1 for Winsock 1.1 or 2 for
  Winsock 2.2. Failing that, the program displays a warning and halts.
  To run this program from the IDE, Select Run|Parameters from the Run option
  in the IDE toolbar and enter 1 or 2 in the Parameters edit box. To run the
  application from the command line, type in the following:
```

```
  ex21 1
  or

  ex21 2

  for WinSock 1.1 or Winsock 2.2, respectively.
}
program EX21;

{$APPTYPE CONSOLE}

uses
  WinSock2,
  SysUtils;


const
 Version1 : Word = $101; // Version 1.1
 Version2 : word = $202; // Version 2.2

var
  WSAData : TWSAData;
  Version : Word;
begin
 Version := 0;
 if ParamStr(1) = '1' then
  Version := Version1
 else
 if ParamStr(1) = '2' then
  Version := Version2
 else
 begin
  WriteLn('Missing version. Please input 1 for Version 1.1 or 2 for Version 2.2');
  Halt;
 end;
 if WSAStartUp(Word(Version), WSAData) = 0 then // yes, Winsock does exist ...
 try
  WriteLn(Format('Version = %d.%d',[Hi(WSAData.wVersion),Lo(WSAData.wVersion)]));
  WriteLn(Format('High Version =  %d.%d',[Hi(WSAData.wHighVersion),
                Lo(WSAData.wHighVersion)]));
  WriteLn(Format('Description = %s',[WSAData.szDescription]));
  WriteLn(Format('System Status = %s',[WSAData.szSystemStatus]));
  WriteLn(Format('Maximum Number of Sockets = %d',[WSAData.iMaxSockets]));
  WriteLn(Format('Maximum Size of a Datagram = %d bytes',[WSAData.iMaxUdpDg]));
  if WSAData.lpVendorInfo <> NIL then
    WriteLn(Format('Vendor Information = %s',[WSAData.lpVendorInfo]));
 finally
  WSACleanUp;
 end
 else WriteLn('Failed to initialize Winsock.');
end.
```

Figure 2-1 shows output from EX21 calling Winsock 1.1. Compare this output to that produced by the same program but calling 2.2 in Figure 2-2.

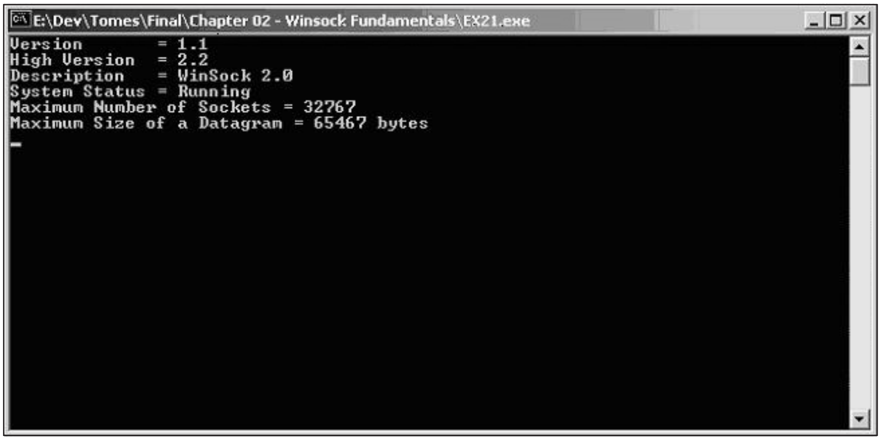


Figure 2-1

Notice that the fields *iMaxSockets* and *iMaxUdpDg*, used to return the maximum number of sockets and the maximum size of the message, respectively, give us no useful information.

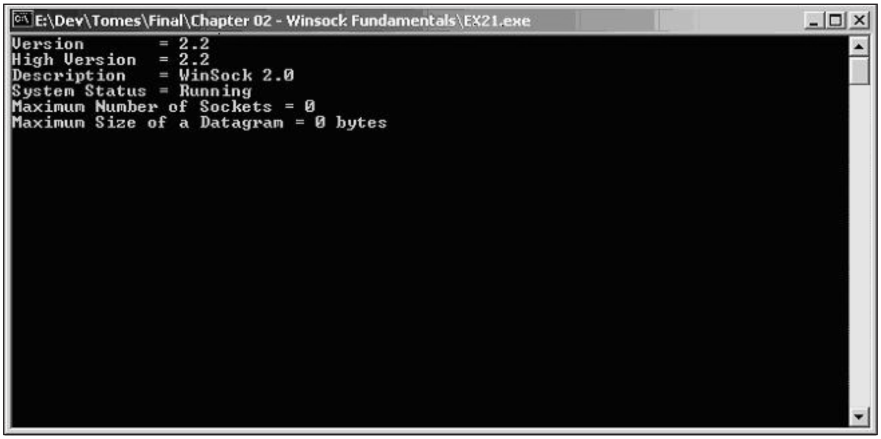


Figure 2-2

Handling Winsock Errors

Like any application, a Winsock application can fail. You cannot always prevent an application error, but you can at least detect and handle any Winsock error. There are two classes of Winsock errors. One is an error caused by inappropriate calls to the Winsock function. A classic example of this is calling any other Winsock function without first calling the WSAStartup() function. The other is a network error, which is completely unpredictable, hence the importance of trapping this type of error.

To help you detect and handle errors, Winsock provides two functions, WSAGetLastError() and WSASetLastError(). When an error occurs, your application should determine the error by calling WSAGetLastError() and take appropriate action, depending on the context of the error. For example, when an application makes an inappropriate call to an API, it should report the error and retire gracefully. For a network error, the application should handle it in context. For example, if a connection breaks, the application should report the error and perform another task or retire altogether.

WSAGetLastError() is a wrapper for GetLastError(), which is a standard function for reporting errors in Windows, and because GetLastError() uses a TLS (thread local storage) entry in the calling threads context, WSAGetLastError() is thread safe. (For more information on threads, consult *The Tomes of Delphi: Win32 Core API—Windows 2000 Edition* by John Ayres (ISBN 1-55622-750-7) from Wordware Publishing, Inc.).

For a robust Winsock application, the strategy to employ is as follows: After each call to a Winsock API, you must check the result of the function (which is usually SOCKET_ERROR, though INVALID_SOCKET is used for certain function calls such as socket()). If there is an error, you call WSAGetLastError() to determine the cause of the error. The application code should always provide a means of handling the error gracefully and retiring, if necessary. You can use the WSASetLastError() function to set an error code that your application can use in certain situations. This function is similar to SetLastError(), which, like GetLastError(), is also a member of the Win32 API.

WSAGetLastError() is not the only function to return a Winsock error code. The other reporting functions are getsockopt(), WSAGetAsyncError(), and WSAGetSelectError(). WSAGetAsyncError() and WSAGetSelectrror() are functions that extract additional error information whenever an error occurs. You should use WSAGetAsyncError() and WSAGetSelectError() rather than WSAGetLastError() when you use Microsoft's family of asynchronous functions, which we will cover in Chapters 3 and 5.

The WSASetLastError() function is a useful function to deploy, provided you are aware of the *caveat emptor* of using this function inappropriately. You use WSASetLastError() to set a virtual error that your application can retrieve with a call to WSAGetLastError(). However, any subsequent call to WSAGetLast-Error() will wipe out the artificial error, which is where the *caveat emptor* comes in if your program logic is incorrect. To explain the use of WSASetLastError(), I have developed a rather contrived example in Listing 2-3.

Errors and errors

As you would expect, error codes, like socket functions, have a UNIX pedigree. The list of errors and their brief descriptions are in Appendix B. As well as that pedigree, we have Winsock-specific error codes resulting in a hybrid. If you examine Winsock2.pas, you will see two blocks of error codes that begin with WSA and E prefixes. These refer to Winsock and Berkeley error codes, respectively. The Berkeley error codes are mapped to their Winsock equivalents. This mapping is rather useful for UNIX developers porting their socket applications to Windows. Thankfully, this detail is irrelevant to Delphi developers.

Rather than listing what's common to Winsock and UNIX socket error codes, the following list shows Winsock-specific error codes not found in UNIX. We will describe some of these errors in detail when we discuss the Winsock functions in the chapters to follow. Note that we will not discuss Quality of Service (error codes from WSA_QOS_RECEIVERS to and including WSA_QOS_RESERVED_PETYPE), as this is a topic for another tome.

WSASYSNOTREADY
WSAVERNOTSUPPORTED
WSANOTINITIALISED
WSAEDISCON
WSAENOMORE
WSAECANCELLED
WSAEINVALIDPROCTABLE
WSAEINVALIDPROVIDER
WSAEPROVIDERFAILEDINIT
WSASYSCALLFAILURE
WSASERVICE_NOT_FOUND
WSATYPE_NOT_FOUND
WSA_E_NO_MORE
WSA_E_CANCELLED
WSAEREFUSED
WSA_QOS_RECEIVERS
WSA_QOS_SENDERS
WSA_QOS_NO_SENDERS
WSA_QOS_NO_RECEIVERS
WSA_QOS_REQUEST_CONFIRMED
WSA_QOS_ADMISSION_FAILURE
WSA_QOS_POLICY_FAILURE
WSA_QOS_BAD_STYLE
WSA_QOS_BAD_OBJECT
WSA_QOS_TRAFFIC_CTRL_ERROR
WSA_QOS_GENERIC_ERROR
WSA_QOS_ESERVICETYPE
WSA_QOS_EFLOWSPEC
WSA_QOS_EPROVSPECBUF
WSA_QOS_EFILTERSTYLE
WSA_QOS_EFILTERTYPE
WSA_QOS_EFILTERCOUNT

WSA_QOS_EOBJLENGTH	WSA_QOS_EPSFLOWSPEC
WSA_QOS_EFLOWCOUNT	WSA_QOS_EPSFILTERSPEC
WSA_QOS_EUNKOWNPSOBJ	WSA_QOS_ESDMODEOBJ
WSA_QOS_EPOLICYOBJ	WSA_QOS_ESHAPERATEOBJ
WSA_QOS_EFLOWDESC	WSA_QOS_RESERVED_PETYPE

Before concluding this section, here is a final word to the wise about error codes: It is all very well for your application to handle Winsock exceptions and report error codes as they arise. Your Winsock application should also present exceptions in plain language as well as the actual error code for ease of error reporting for the user. In the examples in this book, we use SysErrorMessage(), a function that translates error codes into plain language that your user will hopefully understand. The sting in the tail with this function is that it doesn't work across all Windows platforms. The SysErrorMessage() function works fine on Windows 2000 but reports an empty string on Windows NT 4.0.

> **TIP:** Use SysErrorMessage() to present a meaningful explanation of Winsock errors to your users.

Listing 2-3 demonstrates how to use SysErrorMessage().

function WSAGetLastError *Winsock2.pas*

Syntax

WSAGetLastError: Integer; stdcall;

Description

This function retrieves the error status for the last network operation that failed.

Parameters

None

Return Value

The return value indicates the error code for the last operation that failed.

See Also

getsockopt, WSASetLastError

Example

Listing 2-2 (program EX22) shows how to use WSAGetLastError().

Listing 2-2: Using WSAGetLastError()

```
{This example demonstrates how to use WSAGetLastError function. To create an artificial
error, we set the size of the Name array to zero before calling the function gethostname(),
which will cause Winsock to report a bad address due to an insufficient allocation to store
the name. We will examine the gethostname() function later in the book.
No inputs are required for this console application.

}
program EX22;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Winsock2;

var
  WSAData : TWSAData;
  Res     : Integer;
begin
 if WSAStartUp($101, WSAData) = 0 then
 try
  Res := gethostname('127.0.0.1',0); // this will always fail ...
  if Res = Integer(SOCKET_ERROR) then
  begin
   WriteLn(Format('Call to gethostname() failed with error: %d',[WSAGetLastError]));
   WriteLn(Format('Reason for the error is: %s',[SysErrorMessage(WSAGetLastError)]));
  end;
 finally
  WSACleanUp;
 end
 else
  WriteLn('Failed to load Winsock.');
end.
```


procedure WSASetLastError Winsock2.pas

Syntax

WSASetLastError (iError: Integer); stdcall;

Description

The function sets the error code that can be retrieved through the WSAGetLastError() function.

Parameters

iError: Integer that specifies the error code to be returned by a subsequent WSAGetLastError() call

Return Value

There is no return value.

See Also

getsockopt, WSAGetLastError

Example

Listing 2-3 (program EX23) shows how to use WSASetLastError() and WSAGetLastError().

Listing 2-3: Using WSASetLastError() and WSAGetLastError()

```
{This contrived example demonstrates how to use the WSASetLastError() function}
program EX23;

{$APPTYPE CONSOLE}

uses
  Dialogs,
  SysUtils,
  Winsock2;

var
  WSAData : TWSAData;
  Res,
  OldError : Integer;
begin
 if WSAStartUp($101, WSAData) = 0 then
 try
// Create a virtual error, any old error code will do nicely ...
  OldError := 10061;
  WSASetLastError(OldError);
  WriteLn(Format('Virtual error is: %d',[WSAGetLastError]));
  WriteLn(Format('Reason for the virtual error is: %s',[SysErrorMessage(WSAGetLastError)]));
// Now create an artificial error ...
  Res := gethostname('127.0.0.1',0); // This will always fail as length of the name is
                                        zero...
  if Res = Integer(SOCKET_ERROR) then
  begin
   WriteLn('An Artificial Error:');
   WriteLn(Format('Call to gethostname() failed with error: %d',[WSAGetLastError]));
   WriteLn(Format('Reason for the error is: %s',[SysErrorMessage(WSAGetLastError)]));
   WriteLn;
   WriteLn(Format('The virtual error is %d', [OldError]));
   WSASetlastError(OldError);
   WriteLn(Format('Reason for the virtual error is: %s',[SysErrorMessage(WSAGetLastError)]));
  end;
 finally
  WSACleanUp;
 end
 else
  WriteLn('Failed to load Winsock.');
end.
```

The Many Faces of the Winsock DLL

By this stage, you might have the impression that Winsock 2 is a monolithic API wrapped in a DLL. Not so! At least, it is no longer true for Winsock 2. Unlike Winsock 1.1, which had only one transport protocol to contend with, namely TCP/IP, Winsock 2 is designed to handle transport protocols other than TCP/IP. (If you cast your mind back to Chapter 1, Winsock is an integral component of WOSA.) Complicating matters, Winsock 2 also has to handle different name spaces for the resolution of names, services, and ports. (Don't worry; we will cover these topics in Chapter 4.) This complexity, which permits Winsock 2 to be *multilingual*, is reflected in how Winsock 2 is structured across DLLs. This sharing of tasks by DLLs becomes clear if you take a look at Table 2-4. As split up as Winsock 2 is, the main DLL for the Winsock 2 API resides in the Ws2_32 DLL. Those applications that require Winsock 1.1 are handled by the Winsock and WSock32 DLLs, which are 16-bit and 32-bit, respectively. When an application calls the Winsock 1.1 API, Winsock 2 intercepts these calls and passes them to the Winsock and Wsock32 DLLs as appropriate. This is known as *thunking*. Winsock 2 delegates tasks to the appropriately called *helper* DLLs. For example, Wshatm handles functions specific to the ATM transport protocol.

Table 2-4: How Winsock 2 is shared across DLLs

Winsock Files	Function
Winsock.dll	16-bit Winsock 1.1
Wsock32.dll	32-bit Winsock 1.1
Ws2_32.dll	Main Winsock 2.0
Mswsock.dll	Microsoft extensions to Winsock. Mswsock.dll is an API that supplies services that are not part of Winsock.
Ws2help.dll	Platform-specific utilities. Ws2help.dll supplies operating system-specific code that is not part of Winsock.
Wshtcpip.dll	Helper for TCP
Wshnetbs.dll	Helper for NetBT
Wshirda.dll	Helper for IrDA (infrared sockets)
Wshatm.dll	Helper for ATM
Wshisn.dll	Helper for Netware
Wshisotp.dll	Helper for OSI transports
Sfmwshat.dll	Helper for Macintosh
Nwprovau.dll	Name resolution provider for IPX
Rnr20.dll	Main name resolution
Winrnr.dll	LDAP name resolution
Msafd.dll	Winsock interface to kernel
Afd.sys	Winsock kernel interface to TDI transport protocols

Summary

We have learned how to load and unload Winsock. We also learned how to detect Winsock and handle errors. In the next chapter, we'll learn how to use the various functions for resolving hosts and services. Resolution of hosts, ports, and services is an essential step to perform before communication can occur between peer applications.

Chapter 3

Winsock 1.1 Resolution

With the introduction of Winsock 2, Microsoft provided developers with a protocol-independent API that resolves hosts, protocols, and services in a more flexible and powerful way than the services that came with Winsock 1.1. The use of these new functions, though, comes at a price in terms of increased complexity. As with most other Microsoft APIs, the original functions are still valid and simpler to understand. However, it is worthwhile to pick up this technology of protocol-independent functions for the resolving of hosts and services because by using the concept of protocol independence, we can simplify the whole process of resolving host names and services.

However, before we begin to explore the new functions, we must lay the foundation by understanding the rudiments of resolving hosts, protocols, and services. With that background, you will be prepared to master the more complex Winsock 2 resolution functions. Therefore, we'll concentrate on Winsock 1.1 resolution functions in this chapter and leave the Winsock 2 protocol-independent functions to the next chapter. Before dealing with the Winsock 1.1 resolution functions in detail, we'll examine the translation functions that handle byte ordering.

Before dipping our toes in the unknown waters of Winsock resolution, let's consider this question: What is Winsock 1.1 resolution? We'll use a simple analogy to discover an answer to this question. You use a telephone directory to look up a telephone number to call your friend. The telephone directory enables you to quickly retrieve your friend's telephone number without having to remember the number. When it comes to host name resolution, the same principle applies. When you want to connect with a host on the Internet, you need to know its IP address, which, if you like, is the equivalent of the telephone number. Hosts on every TCP/IP network, including the Internet, use IP addresses to identify themselves to each other. Unfortunately, the majority of humans (and that includes Delphi developers) cannot remember IP addresses in their raw form. In the early days of the Internet, IP addresses were routinely used but became impossible when the Internet expanded. To resolve (no pun intended) this problem, the mechanism of mapping names (essentially aliases) to IP addresses came into being. The mapping of these aliases to their IP addresses is called *host name resolution*. Because of mapping names that are user friendly, that is, easy to remember, you don't need to know the host's IP address, provided you know its friendly name.

Establishing a mapping of a host name with an IP address is not the end of the equation. Before you can communicate with a TCP/IP host, you need to know the port upon which the host operates the desired service, like FTP and HTTP. Extending the telephone directory analogy, you would either know your friend's extension or speak with the operator to put you through to your friend. Perhaps in your case, you would speak to the operator to get through. This is analogous to what we call *service resolution*. Added to this equation, we must also resolve service names to their port numbers. Services such as FTP and HTTP are well known to surfers on the Net, but hosts deal in numbers when it comes to providing a service like FTP. Again, service names were invented to make life easier for users. Like host name mapping, it is necessary to map human understandable service names to their ports, which are numbers that hosts can understand.

And that's not all. We also need to resolve transport protocols to their protocol numbers. Hosts require knowing which transport protocols are needed to operate a service. For example, FTP requires the TCP protocol, which hosts translate as 6. We will continue to use the telephone directory analogy as we examine the Winsock 1.1 resolution functions.

Translation Functions

Computers store numbers through byte ordering. There are two ways to represent numbers, *little endian* and *big endian*. Intel processors store numbers in little endian format—from the least significant byte to the most significant byte (right to left). On other processors (such as those that run some UNIX systems), numbers are in big-endian format—from the most significant byte to the least significant byte—left to right. Since the Internet is a heterogeneous network of different computers, incompatible byte ordering poses a significant obstacle to communication. To overcome this barrier, current network standards specify that ports used for communicating between computers should be in network byte order (otherwise known as big endian format), irrespective of their native byte ordering. That is, network byte order is big endian for use on the TCP/IP network. You mustn't forget that network addresses, datagram length, and TCP/IP window sizes must also be in network byte order (big endian).

Figure 3-1 on the following page shows how little endian and big endian numbers are stored in memory.

So, before using resolution functions and starting communications, your application needs to translate the native host byte (little endian) ordered number (for example, port number of the host on the PC) to network byte ordered

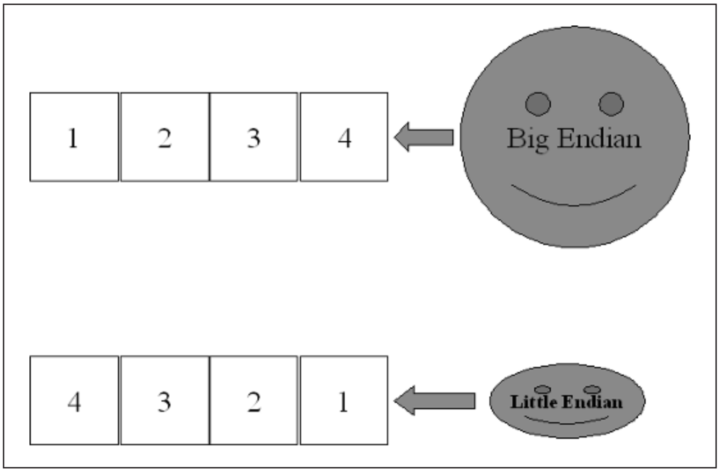


Figure 3-1

number first. That is, you must translate the port number into network byte order. If this translation is not done, it is very likely that connecting with the service on the host will never occur, even if the host name resolution works. Another problem that can cause you to scratch your head is using the port in host byte order instead of network byte order, which is a common lapse. However, it is not necessary to convert numerical data into network byte order; only port numbers, services, and IP addresses need to be in network byte order.

The following trivial example shows graphically the effect of converting a number in host byte order to network byte order and from network byte order to host byte order:

```
host: 100d ➔ 00000064h
network: 64000000h = 1677721600d
```

The following functions are used to convert from host byte order to network byte order, or network byte order to host byte order.

function htonl Winsock2.pas

Syntax

htonl(hostlong: u_long): u_long; stdcall;

Description

This function translates a 32-bit integer from host byte order to network byte order. In other words, it translates an integer in little endian format to big endian format.

Parameters

hostlong: A 32-bit integer in host byte order

Return Value

The function will return a value in network byte order.

See Also

htons, ntohl, ntohs, WSAHtonl, WSAHtons, WSANtohl, WSANtohs

Example

See Listing 3-1 (program EX31).

function htons Winsock2.pas

Syntax

```
htons(hostshort: u_short): u_short; stdcall;
```

Description

This function translates a 16-bit integer from host byte order to network byte order. In other words, it translates an integer in little endian format to big endian format.

Parameters

hostshort: A 16-bit number in host byte order

Return Value

The function will return a value in network byte order.

See Also

htonl, ntohl, ntohs, WSAHtonl, WSAHtons, WSANtohl, WSANtohs

Example

See Listing 3-1 (program EX31).

function ntohl Winsock2.pas

Syntax

```
ntohl(netlong: u_long): u_long; stdcall;
```

Description

This function converts a 32-bit integer from network byte order to host byte order. In other words, it translates an integer in big endian format to little endian format.

Parameters

netlong: A 32-bit integer in network byte order

Return Value

The function will return a value in host byte order.

See Also

htonl, htons, ntohs, WSAHtonl, WSAHtons, WSANtohl, WSANtohs

Example

See Listing 3-1 (program EX31).

function ntohs Winsock2.pas

Syntax

ntohs(netshort: u_short): u_short; stdcall;

Description

This function converts a 16-bit integer from network byte order to host byte order. In other words, it translates an integer in big endian format to little endian format.

Parameters

netshort: A 16-bit integer in network byte order

Return Value

The function will return a value in host byte order.

See Also

htonl, htons, ntohl, WSAHtonl, WSAHtons, WSANtohl, WSANtohs

Example

Listing 3-1 demonstrates how to use these functions: htonl(), htons(), ntohl(), and ntohs(). This example requires a number on the command line. For example, you would type the following:

```
EX31 n
```

where *n* is the number to convert.

Listing 3-1: Using htonl(), htons(), ntohl(), and ntohs()

```
{Example EX31 demonstrates how to convert numbers from network to host order and vice versa.
 The following functions are used: htons(), htonl(), ntohs() and ntohl().}

program EX31;

{$APPTYPE CONSOLE}

uses
  Dialogs,
  SysUtils,
  Winsock2;

const
  WSVersion: Word = $101;

var
  WSAData: TWSAData;
```

```
  Value: Cardinal;
  Code: Integer;
begin
  if ParamCount < 1 then
  begin
    WriteLn('Missing value. Please input a numerical value.');
    Halt;
  end;
  // Convert input to a numerical value ...
  Val(ParamStr(1), Value, Code);
  // Check for bad conversion
  if Code <> 0 then
  begin
    WriteLn(Format('Error at position: %d', [Code]));
    Halt;
  end;
  if WSAStartUp(Word(WSVersion), WSAData) = 0 then // yes, Winsock does exist ...
  try
    WriteLn(Format('Using htonl() the value %d converted from host order to network order
                  (long format)  = %d', [Value, htonl(Value)]));
    WriteLn(Format('Using htons() the value %d converted from host order to network order
                  (short format) = %d', [Value, htons(Value)]));
    WriteLn(Format('Using ntohl() the value %d converted from network order to host order
                  (long format)  = %d', [Value, ntohl(Value)]));
    WriteLn(Format('Using ntohs() the value %d converted from network order to host order
                  (short format) = %d', [Value, ntohs(Value)]));
  finally
    WSACleanUp;
  end
  else WriteLn('Failed to initialize Winsock.');
end.
```

Miscellaneous Conversion Functions

The functions we have just examined relate to translating numbers between different endian formats. What about IP addresses and their matching host names? In this section, we will look at functions that convert an IP dotted address into a network address and vice versa. Be aware, however, that these functions only translate between different formats and don't actually resolve names and IP addresses; we will examine those functions that do later in this chapter.

function inet_addr   Winsock2.pas

Syntax

inet_addr(cp: PChar): u_long; stdcall;

Description

This function converts a NULL-terminated string containing an Internet Protocol (IP) address in dotted decimal format into an Internet network address (in_addr) in network byte order.

Parameters

cp: A pointer to a NULL-terminated string containing an Internet Protocol address in dotted decimal format (e.g., 192.168.0.1)

Return Value

If successful, the function will return an unsigned long integer that contains a binary representation of the Internet address. Otherwise, the function returns the value INADDR_NONE. An invalid Internet Protocol address in dotted decimal format will cause a failure. For example, if any number in the IP address exceeds 255, the conversion will fail.

See Also

inet_ntoa

Example

See Listing 3-2 (program EX32).

function inet_ntoa Winsock2.pas

Syntax

inet_ntoa(inaddr: TInAddr): PChar; stdcall;

Description

This function translates an Internet network address into a NULL-terminated string containing an IP address in dotted decimal format.

TIP: Since the string returned by inet_ntoa() resides in a buffer in memory, there is no guarantee that the contents of this buffer will not be overwritten when your application makes another Winsock call. It is safer to store the contents of the buffer returned by inet_ntoa() should your application require it later.

Parameters

inaddr: A record that represents an IP address. The record, which is defined in Winsock2.pas, looks like this:

```
in_addr = record
  case Integer of
    0: (S_un_b: SunB);
    1: (S_un_c: SunC);
    2: (S_un_w: SunW);
    3: (S_addr: u_long);
end;
TInAddr = in_addr;
PInAddr = ^in_addr;

SunB = packed record
                s_b1,
```

```
                s_b2,
                s_b3,
                s_b4: u_char;
       end;

SunC = packed record
                s_c1,
                s_c2,
                s_c3,
                s_c4: Char;
             end;

SunW = packed record
                 s_w1,
                 s_w2: u_short;
              end;
```

where SunB and SunC are the addresses of the host formatted as four u_chars and SunW is the address of the host formatted as two u_shorts.

Finally, S_addr is the address of the host formatted as a u_long.

Return Value

If successful, the function will return a pointer to a NULL-terminated string containing the address in standard Internet dotted notation. Otherwise, it will return NIL.

See Also

inet_addr

Example

Listing 3-2 shows how to use the inet_ntoa() and inet_addr() functions. The example also shows that inet_ntoa() and inet_addr() are inverses of each other.

Listing 3-2: Using inet_ntoa() and inet_addr()

```
{This example demonstrates two functions inet_addr() and inet_ntoa().
 The inet_addr function converts a null-terminated string containing
 an Internet Protocol (IP) dotted address into an Internet network address
 (in_addr) in network byte order.
 The inet_ntoa function translates an Internet network address into a
 null-terminated string containing a dotted IP address.}

program EX32;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Winsock2;

const
  WSVersion: Word = $101;

var
  WSAData: TWSAData;
  Address: TInAddr;  // socket address structure
  Addr: Integer;
```

```
  AddrStr: String;
begin
  if WSAStartUp(WSVersion, WSAData) = 0 then // yes, Winsock does exist ...
  try
    Addr := inet_addr('127.0.0.1');
    if Addr = INADDR_NONE then
      WriteLn('Error converting 127.0.0.1')
    else
      WriteLn('inet_addr(127.0.0.1) returned: ' + IntToStr(Addr));
      Address.S_addr :=16777343; // This is the address for 127.0.0.1 ...
    AddrStr := String(inet_ntoa(Address));
    if AddrStr = '' then
      WriteLn('Error converting 16777343')
    else
      WriteLn('inet_ntoa(16777343) returned: ' + AddrStr);
  finally
    WSACleanUp;
  end
  else WriteLn('Failed to initialize Winsock.');
end.
```


Resolution

When you want to communicate with a host on the Internet, you must ascertain that host's network address. Each host on the Internet has a unique IP address that has a name associated with it, usually a mnemonic or something that matches the company's name or product that has a corresponding network address. A host can have many names assigned to the same host. For example, Wordware Publishing, the publishers of this fine tome and many other excellent publications, has a host name of www.wordware.com. The host name or alias can be a mixture of alphabetic and numeric characters up to 255 characters long. Host names can take various forms. The two most common forms are a nickname and a domain name. A *nickname* is an alias to an IP address that individual people can assign and use. A *domain name* is a structured name that follows Internet conventions.

TIP: With Windows-based servers, the host name does not have to match the Windows computer name.

In short, the "www" component is the service for the World Wide Web and "wordware.com" is the domain. The domain has a registered DNS server (a host that is running Domain Name System) that resolves the service (www) in this domain to a specific host (or even hosts) that provide that service (which may in fact exist outside of wordware.com). To put it in another way, wordware.com is the DNS domain name and www is a "protocol entry" (CNAME record) in the DNS zone database that will be mapped to a host name by DNS. For your client application to communicate with the host, it has to look up the network address for that host name. Think of this like a postal system;

you cannot send mail to anyone unless you have his or her street address. Occasionally, you may want to connect to a host that has no name at all but is reachable through an IP address in decimal dotted format. Fortunately, this is a rare beast nowadays. (The exception is the router, of which there are many. It is a host that specializes in managing, or *routing*, traffic between networks. It doesn't offer any services, such as FTP and HTTP, and therefore has no name.)

There are three ways to resolve a host name, which are:

- Hosts file
- DNS server
- Local database file with DNS

Resolving Using a hosts File

The simplest way to resolve a host name to an IP address is to use a locally stored database file. This database file (the name of which is a misnomer) is nothing more than a text file that contains a list of IP addresses and their host names. On Windows NT, Windows 2000, and Windows XP systems, this database file is the hosts file (it has no extension), which resides in the \system32\drivers\etc directory. For those of you who are planning to develop sockets applications using Kylix for Linux, the database file is in the /etc directory. The following shows a typical hosts file.

```
# Copyright (c) 1993-1999 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.

127.0.0.1         localhost
192.168.1.1       newton.craiglockhart.com newton
192.168.1.2       laser.craiglockhart.com laser
192.168.1.3       galileo.craiglockhart.com galileo
192.168.1.4       hugyens.craiglockhart.com  hugyens
```

The host name, such as newton.craiglockhart.com, is known as a fully qualified domain name (FQDN). Rather than type out the FQDN of the host when you need to connect every time, you can simply use another alias, which is in the third column in the hosts file. In the case of newton.craiglockhart.com, it is newton.

In the hosts file on both Windows and Linux systems, there is always a special entry, which is 127.0.0.1, called *localhost*. This is a special IP address known as the *loopback address*. What is so special about this address? Simply put, instead of having a server on another machine, you can have the server on the same machine as the client. In other words, the server and client share this address, which is very convenient for testing client-server systems on the same machine that has no network connection.

Although testing client server systems (I am referring to applications that use Winsock) on the same machine is not an ideal way to test, it is a good way to test the logic of such systems. For proper testing of such systems, it is preferable to locate the server on a separate machine and the client on a separate machine on a different network from that of the server. By this arrangement, you can test the robustness of such a system under varying network loads, a factor that is obviously missing from a stand-alone machine. Taking the telephone book analogy further, the hosts file is like your personal numbers book. Using the hosts file like a telephone directory is not the answer, as it is not scaleable because it becomes unmanageable to maintain an expanding hosts file when adding new hosts and deleting hosts. The solution to this management problem comes in the form of DNS.

Resolving Using DNS

The Domain Name System (DNS) was designed to make host name resolution scaleable and centrally manageable. A DNS server maintains a special database that contains IP address mappings for fully qualified domain names (FQDNs).

When your Winsock application requires a connection with a host, it passes an FQDN of the destination host. The application calls a Winsock function to resolve the name to an IP address. The function passes the request to the DNS resolver in the TCP/IP protocol stack, which is packaged as a *DNS Name Query* packet. The resolver sends the packet to the DNS server. If the DNS server resolves the name to an IP address, it sends back the IP address to the application, which then uses the address to communicate with the host. However, this is not the whole story, as we shall soon discover later in the chapter when we discuss these functions.

Before concluding this section concerning DNS, let's explore how DNS servers work. Every DNS host does not store all of its hosts' IP addresses and their FQDNs for the entire Internet; that would be an impossible mission to keep all hosts' DNS databases synchronized. Instead, each DNS host is responsible for a region or zone of Internet hosts. When your client application wishes to connect with a host, the first DNS host, which is the local DNS host to which your ISP has configured your TCP/IP settings by default, attempts to resolve the FQDN that your application sent. If no matching IP address is found, the DNS host passes the request to an *authoritative* DNS host, which in turn attempts to resolve the FQDN. This "passing the buck" approach is achieved by having the database on each DNS host point to each other.

Resolving Using a Local Database File with DNS

In many ways, this is the best solution because it's flexible enough to resolve a host using the database file (the hosts file) locally. If the host is not found, DNS is invoked to resolve the host. This combined approach to resolving host names operates like this:

1. Check the local database file (the host's file) for a matching name.
2. If a matching name is not found in the local database file, the host name is packaged as a DNS Name Query and sent to the configured DNS server somewhere on another network.

However, resolution does not end with hosts. To make use of services such as FTP and SMTP, you also need to resolve services that hosts provide, such as the web (www) service for Wordware (www.wordware.com). It would need to be resolved before you can surf that site. To complicate matters a little more, resolving the underlying protocol for the required service is also necessary.

Before examining the Winsock 1.1 resolution functions in detail, we must compare the pros and cons of using blocking and asynchronous functions.

Blocking and Asynchronous Resolution

Winsock provides two sets of functions to resolve hosts, protocols, and services. The first set uses the concept of a blocking operation, and the second set uses asynchronous mode.

Using a blocking function in the main thread of the application causes the user interface to "freeze" during resolution. That is, the operation blocks until it gets a result, preventing any input from the keyboard or mouse. Freezing the user interface can be inconvenient and possibly not user friendly. However, the time it takes to use a blocking function may be short if we are resolving over a fast LAN. To overcome this freezing problem, you should use threads in your application, a technique we will discuss later in Chapter 5. By putting a blocking function on a background thread, the user is allowed to continue with other tasks in the application.

TIP: Freezing of the application or Windows may occur when using a blocking function. To prevent this, place such functions in their own thread. This will not work if you use the same thread for both the user interface and the blocking functions.

As you'll see then, you can resolve (pun not intended) this problem by placing the blocking function on a background thread that will allow the user to interact with the application interface.

When you use a blocking function, such as gethostbyaddr(), to resolve a host, the process is a complex one (which we covered when we explored DNS) that involves several steps like this:

```
Hosts file ➔ DNS ➔ WINS ➔ broadcasts ➔ LMHOSTS ➔ DNS
```

The function queries the local database first. This database is just a text file called "hosts" that contains the names of hosts and their corresponding IP addresses. You will find this file in \Winnt\System32\drivers\etc on NT 4.0 and Windows 2000 and in \Windows\System on Windows 95/98 systems. If there is no entry that matches the query, the function contacts the local name server (via a dial-up line or over a permanent connection) to use DNS (Domain Name System) to search for a match. If there is no match, the Windows Internet Naming Service (WINS) broadcasts a request. If this fails, DNS is called again. If DNS cannot find a match, the function returns a NIL result. Looking at that sequence of events, it is no wonder that a search for a match can take some time because the calling thread or application is waiting for it to return, hence the term "blocking." However, if the host name is in the hosts file, then the function will return quickly.

TIP: To speed up lookups, you can store your favorite web site with its IP address in the hosts file. A word of warning: This can fail if the owners of the web site change the IP address without prior warning.

Resolving a service or protocol is no different from resolving a host. When there is no corresponding service, the function uses the DNS service to search the CNAME records in the database. These functions, such as getservbyname(), query the local database, which is located in the *services* file. If the function cannot find a match, then it calls DNS. If there is no match, the function returns an error. This is also true for resolving protocols, and the local database to use is in the *protocol* file.

To overcome the problem of blocking, Winsock provides an additional set of resolution functions that operate asynchronously. Using this set of asynchronous functions, which is essentially a mapping of the set of blocking functions, enables the user to interact with your application while resolution proceeds in the background. These asynchronous functions take advantage of the Windows messaging system.

When your application calls an asynchronous function, Winsock initiates the operation and returns to the application immediately, passing back an asynchronous task handle that your application uses to identify the operation. When the operation is complete, Winsock copies the data returned into a buffer that is provided by the application and sends a message to the application's window.

When the asynchronous operation is complete, your application's message window *hWnd* receives the message in the *wMsg* parameter. The *wParam* parameter of this message contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain an error code, which may be any error as defined in Winsock2.pas. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer passed to the asynchronous function contains a record. To access this record, you should cast the original buffer address as a record pointer. It is important to parse each message that your application receives. Your application should call the WSAGetAsyncError() function to check the *lParam* argument.

Note that if the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it may reissue the asynchronous function call with a buffer large enough to receive all the desired information (i.e., no smaller than the low 16 bits of *lParam*).

If Winsock could not start the asynchronous operation, the function will return a zero value, and you should call WSAGetLastError() to determine the cause of the error. However, the price to pay for this is an increase in program complexity and some overhead. Applications that use blocking functions are simpler and cleaner.

Now we will return to the set of blocking functions.

Host Resolution

The blocking functions that resolve hosts are gethostbyaddr() and gethostbyname(); their asynchronous equivalents are WSAAsyncGetHostByAddr() and WSAAsyncGetHostByName(), respectively. To resolve the host name of the machine that you are using, call the gethostname() function.

function gethostbyaddr   Winsock2.pas

Syntax

```
gethostbyaddr(addr: PChar; len, type_: Integer): PHostEnt; stdcall;
```

Description

The function returns a pointer to the THostEnt record containing one or more "names" and addresses that correspond to the given address. All strings are NULL terminated.

The Hostent record is defined in Winsock2.pas as follows:

```
Hostent = record
  h_name: PChar;              // official name of host
  h_aliases: PPChar;          // alias list
  h_addrtype: Smallint;       // host address type
  h_length: Smallint;         // length of address
  case Integer of
    0: (h_addr_list: PPChar);  // list of addresses
    1: (h_addr: PPChar);       // address, for backward compatibility
end;
THostEnt = hostent;
PHostEnt = ^hostent;
```

The members of this data structure are defined as:

h_name: Official name of the host

h_aliases: An array of NULL-terminated alternate names

h_addrtype: The type of address, which is usually AF_INET for TCP/IP on the Internet. Other address types include AF_IPX for Netware, AF_ATM for ATM, and AF_UNIX for UNIX.

h_length: The length, in bytes, of each address

h_addr_list: A list of NULL-terminated addresses for the host. Addresses are in network byte order.

h_addr: An address

The pointer that you get back points to a record allocated by Winsock. As the data is transient, your application should copy any information that it needs before issuing any other Winsock function calls.

The field *h_name* is the official name of the host. If you're using the DNS or similar resolution system on the Internet, the name server will return a fully qualified domain name (FQDN). If you're using a local "hosts" database file, it will return the first entry that matches the query.

Parameters

addr: A pointer to an address in network byte order

len: The length of the address in bytes

*type_*: The type of address, such as AF_INET for TCP/IP

Return Value

If successful, the function will return a pointer to the THostEnt record. Otherwise, it will return NIL. To retrieve information about the error, call the WSAGetLastError() function. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, WSANO_DATA, WSAEINPROGRESS, WSAEFAULT, and WSAEINTR.

See Appendix B for a detailed description of the error codes.

See Also

gethostbyname, WSAAsyncGetHostByAddr

Example

Listing 3-3 (program EX33) shows how to use the gethostbyaddr() function.

Listing 3-3: Using gethostbyaddr()

```
{ The EX33 example demonstrates the gethostbyaddr() function.
  The command line parameter to use is the IP address to resolve. For example,
  to execute the program to resolve the IP address 127.0.0.1, you would
  type the following:

  EX33 127.0.0.1

  The gethostbyaddr() function returns a pointer to the THostent record
  containing one or more name(s) and addresses that correspond to the given
  address.  All strings are NULL terminated.}

program EX33;

{$APPTYPE CONSOLE}

uses
  Dialogs,
  SysUtils,
  Winsock2;

const
  WSVersion: Word = $101;

var
  WSAData: TWSAData;
  Address: Integer;
  Hostent: PHostent;
  HostName: string;
  Len,
  AddrType: Integer;
begin
  if ParamCount < 1 then
  begin
    WriteLn('Error - missing IP address. Please supply an IP address in' + #10#13 + 'dotted
            IP notation (e.g. 127.0.0.1).');
    Halt;
  end;
  HostName := ParamStr(1);
  if WSAStartUp(WSVersion, WSAData) = 0 then // yes, Winsock does exist ...
  try
    Address := inet_addr(PChar(HostName));
    if Address <> INADDR_NONE then // Yes, this is a dotted IP address ...
    begin
      AddrType := AF_INET; // Address Family type, usually AF_INET for TCP/IP ...
      Len := SizeOf(AddrType);
      Hostent := gethostbyaddr(PChar(@Address), Len, AddrType);
      if Hostent <> nil then // success! ...
        WriteLn(Format('IP address %s successfully resolved to %s', [HostName,
                       Hostent^.h_name]))
      else // failure, cannot resolve ...
```

```
        WriteLn(Format('Call to gethostbyaddr() to resolve %s failed with error: %s',
               [HostName, SysErrorMessage(WSAGetLastError)]));
    end else WriteLn('IP address is required.');
  finally
    WSACleanUp;
  end
  else WriteLn('Failed to initialize Winsock.');
end.
end.
```

function gethostbyname Winsock2.pas

Syntax

gethostbyname(name: PChar): PHostEnt; stdcall;

Description

The function retrieves information for the host and returns a pointer to the THostEnt record allocated by Winsock (see gethostbyaddr() for details of THostEnt record). Your application must not modify this record or free any of its components.

TIP: As this data is transient, your application should copy any information that it needs before issuing any other Winsock function calls.

Parameters

name: A pointer to the NULL-terminated name (FQDN) of the host or domain

Return Value

If successful, the function will return a pointer to the THostEnt record. Otherwise, it will return NIL. To retrieve information about the error, call the WSAGetLastError() function. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, WSANO_DATA, WSAEINPROGRESS, WSAEFAULT, and WSAEINTR.

See Appendix B for a detailed description of the error codes.

See Also

WSAAsyncGetHostByName, gethostbyaddr

Example

Listing 3-4 (program EX34) demonstrates how to use the gethostbyname() function.

Listing 3-4: Using gethostbyname()

```
{  Example EX34 demonstrates the gethostbyname() function.

  The command line parameter to use is the host name to resolve. For example,
```

3 Chapter

```
  to execute the program to resolve the host name localhost you would type the following:

  EX34 localhost

  The gethostbyname() function gets host information corresponding to a hostname.
  All strings are NULL terminated.
  The function returns a pointer to the THostent record.
}
program EX34;

{$APPTYPE CONSOLE}
uses
  Dialogs,
  SysUtils,
  Winsock2;

const
 WSVersion : Word = $101;

var
 WSAData : TWSAData;
 Hostent : PHostent;
 HostName : String;
 h_addr  : PChar;
 HostAddress : TSockAddrIn;
begin
 if ParamCount < 1 then
 begin
  WriteLn('Error - missing hostname! Please supply a hostname.');
  Halt;
 end;
 HostName := ParamStr(1);
 if WSAStartUp(WSVersion, WSAData) = 0 then // yes, Winsock does exist ...
 try
// Check if this string contains an Internet dotted address. Reject it if it is ...
  if inet_addr(PChar(HostName)) = INADDR_NONE then
  begin
    Hostent := gethostbyname(PChar(HostName));
    if Hostent <> NIL then
    begin
      Move(Hostent^.h_addr_list^, h_addr, SizeOf(Hostent^.h_addr_list^));
      HostAddress.sin_addr.S_un_b.s_b1 := Byte(h_addr[0]);
      HostAddress.sin_addr.S_un_b.s_b2 := Byte(h_addr[1]);
      HostAddress.sin_addr.S_un_b.s_b3 := Byte(h_addr[2]);
      HostAddress.sin_addr.S_un_b.s_b4 := Byte(h_addr[3]);
      WriteLn(Format('Hostname %s successfully resolved to %s',[Hostname,
              inet_ntoa(HostAddress.sin_addr)]));
    end else WriteLn(Format('Call to gethostbyname() to resolve %s failed with error: %s',
                    [HostName, SysErrorMessage(WSAGetLastError)]));
   end else WriteLn('This is not a valid host name!');
 finally
  WSACleanUp;
 end else
 WriteLn('Failed to load Winsock.');
end.
```

function gethostname Winsock2.pas

Syntax

gethostname(name: PChar; len: Integer): Integer; stdcall;

Description

This function determines the host name of the local machine. Some applications need to be aware of the name of the machine on which they are running; using gethostname() provides this name. The name returned by gethostname() may be a simple name or an FQDN.

Parameters

name: A pointer to a buffer containing a NULL-terminated string for the host name

len: The length of the buffer

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

gethostbyname, WSAAsyncGetHostByName

Example

Listing 3-5 (program EX35) shows how to use the gethostname() function.

Listing 3-5: Using gethostname()

```
{Example EX35 demonstrates the gethostname function.

 The gethostname function returns the standard host name for the
 local machine. }

program EX35;

{$APPTYPE CONSOLE}
uses
  Dialogs,
  SysUtils,
  Winsock2;

const
 WSVersion : Word = $101;

var
 WSAData : TWSAData;
 HostName : PChar;
begin
 if WSAStartUp(WSVersion, WSAData) = 0 then // yes, Winsock does exist ...
```

```
  try
   HostName := AllocMem(MAXGETHOSTSTRUCT);
   try
    if gethostname(HostName, MAXGETHOSTSTRUCT) <> Integer(SOCKET_ERROR) then
     WriteLn(Format('Host name for the local machine is %s',[Hostname]))
    else
     WriteLn(Format('Call to gethostname() failed with error: %s',
                   [SysErrorMessage(WSAGetLastError)]));
   finally
    Freemem(HostName);
   end;
  finally
   WSACleanUp;
  end
  else
   WriteLn('Failed to load Winsock.');
 end.
```

function WSAAsyncGetHostByName Winsock2.pas

Syntax

WSAAsyncGetHostByName(hWnd: HWND; wMsg: u_int; name, buf: PChar; buflen: Integer): HANDLE; stdcall;

Description

This function asynchronously retrieves information corresponding to a host name.

Parameters

hWnd: The handle of the window that should receive a message when the asynchronous request completes

wMsg: The message to receive when the asynchronous request completes

name: A pointer to the NULL-terminated name of the host

buf: A pointer to the data area to receive the THostEnt data. The size of the buffer must be larger than the size of THostEnt record.

buflen: The size of *buf* in bytes

Return Value

The return value will only specify if the operation started successfully; it will not indicate success or failure of the operation itself.

If the operation starts successfully, the function will return a nonzero value of type THandle. Otherwise, the function will return zero. To retrieve the specific error code, call the function WSAGetLastError(). Possible errors are WSAENETDOWN, WSAENOBUFS, WSAEFAULT, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, and WSANO_DATA.

The following errors may occur at the time of the function call, which indicate that the asynchronous operation could not start: WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

gethostbyname, WSACancelAsyncRequest

Example

Listing 3-6 (program EX36) shows how to perform asynchronous lookup calls.

Listing 3-6: Performing asynchronous lookup calls

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls,
  Winsock2;

const
 ASYNC_EVENT = WM_USER+100;

type
 TypeOfLookUp = (HostName, HostAddr, ServiceName, ServicePort, ProtocolName, ProtocolNumber);

  TfrmMain = class(TForm)
    gbService: TGroupBox;
    btnServiceLookUp: TButton;
    edService: TEdit;
    Label1: TLabel;
    rgbProtocols: TRadioGroup;
    gbHost: TGroupBox;
    edHost: TEdit;
    gbProtocol: TGroupBox;
    btnProtocolLookUp: TButton;
    edProtocol: TEdit;
    GroupBox4: TGroupBox;
    Memo1: TMemo;
    btnHost: TButton;
    btnClose: TButton;
    btnCancel: TButton;
    Label2: TLabel;
    edWinsVer: TEdit;
    btnStart: TButton;
    btnStop: TButton;
    procedure Form1Destroy(Sender: TObject);
    procedure btnServiceLookUpClick(Sender: TObject);
    procedure btnProtocolLookUpClick(Sender: TObject);
    procedure btnHostClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure btnCancelClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnStartClick(Sender: TObject);
    procedure btnStopClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
   Host       : PHostent;
```

```
    Service    : PServent;
    Protocol   : PProtoent;
    LookUpType : TypeOfLookUp;
    AsyncBuff  : array[0..MAXGETHOSTSTRUCT-1] of char;
    TaskHandle : Integer;
    TaskWnd    : THandle;
    WSRunning  : Boolean;
    WSAData    : TWSADATA;
    procedure AsyncOp(var Mess : TMessage);
    procedure AbortAsyncOp;
  end;

var
  frmMain: TfrmMain;

implementation

{$R *.DFM}
 procedure TfrmMain.AsyncOp(var Mess : TMessage);
 var
  MsgErr : Word;
  h_addr : PChar;
  SockAddress : TSockAddrIn;
 begin
  if Mess.Msg = ASYNC_EVENT then
  begin
   MsgErr := WSAGetAsyncError(Mess.lparam);
   if MsgErr <> 0 then
    Exception.Create('Error : ' + IntToStr(MsgErr));
   case LookUpType of
    HostName      : begin
                     Host := PHostent(@AsyncBuff);
                     if Host = NIL then
                     begin
                     Memo1.Lines.Add('Unknown host');
                     Exit;
                     end;
                     if Host^.h_name = NIL then
                     begin
                     Memo1.Lines.Add('Host Lookup failed...');
                     Exit;
                     nd;
                     move(Host^.h_addr_list^, h_addr, SizeOf(Host^.h_addr_list^));
                     with SockAddress.sin_addr do
                     begin
                     S_un_b.s_b1 := byte(h_addr[0]);
                     S_un_b.s_b2 := byte(h_addr[1]);
                     S_un_b.s_b3 := byte(h_addr[2]);
                     S_un_b.s_b4 := byte(h_addr[3]);
                     memo1.Lines.Add('IP Address = ' + String(inet_ntoa
                                    (SockAddress.sin_addr)));
                     end;
                     end;
    HostAddr      : begin
                     Host := PHostent(@AsyncBuff);
                     if Host = NIL then
                     begin
                     Memo1.Lines.Add('Unknown Host');
                     Exit;
                     end;
```

```
                   move(Host^.h_addr_list^, h_addr, SizeOf(Host^.h_addr_list^));
                   Memo1.Lines.Add('Host Name = ' + String(Host^.h_name));
                   end;
    ServiceName    : begin
                   Service := PServent(@AsyncBuff);
                   if Service = NIL then
                   begin
                   Memo1.Lines.Add('Unknown Service');
                   Exit;
                   end;
                   Memo1.Lines.Add('Service Port = ' + IntToStr(ntohs(Service^.s_port)));
                   end;
    ServicePort    : begin
                   Service := PServent(@AsyncBuff);
                   if Service = NIL then
                   begin
                   Memo1.Lines.Add('Unknown Service');
                   Exit;
                   end;
                   Memo1.Lines.Add('Service Name = ' + StrPas(Service^.s_name));
                   end;
    ProtocolName   : begin
                   Protocol := PProtoent(@AsyncBuff);
                   if Protocol = NIL then
                   begin
                   Memo1.Lines.Add('Unknown Protocol');
                   Exit;
                   nd;
                   Memo1.Lines.Add('Protocol Number = ' + IntToStr(Protocol^.p_proto));
                   end;
    ProtocolNumber : begin
                   Protocol := PProtoent(@AsyncBuff);
                   if Protocol = NIL then
                   begin
                   Memo1.Lines.Add('Unknown Protocol');
                   Exit;
                   end;
                   Memo1.Lines.Add('Protocol Name = ' + String(Protocol^.p_name));
                   end;
   end;// case
  end // if
 end;

 procedure TfrmMain.AbortAsyncOp;
 begin
  if WSACancelAsyncRequest(THandle(TaskHandle)) = Integer(SOCKET_ERROR) then
   Exception.Create('Error ' + SysErrorMessage(WSAGetLastError))
  else
   Memo1.Lines.Add('Asynchronous Lookup Operation cancelled...');
 end;

procedure TfrmMain.Form1Destroy(Sender: TObject);
begin
 if WSRunning then
 begin
  WSACleanUp;
  DeAllocateHWND(TaskWND);
 end;
end;
```

```
procedure TfrmMain.btnServiceLookUpClick(Sender: TObject);
var
 ProtocolName : String;
 DummyValue, Code : integer;
begin
  if (length(edService.Text) = 0) or (edService.Text = '') then
   Exception.Create('You must enter a service name or port number!');
  val(edService.Text, Dummyvalue, Code);
  if Code <> 0 then // this is not a numerical value ...it is a service name
   LookUpType := ServiceName
  else
   LookUpType := ServicePort;
  FillChar(AsyncBuff, SizeOf(AsyncBuff), #0);
  if rgbProtocols.ItemIndex = 0 then
   ProtocolName := 'tcp'
  else
   ProtocolName := 'udp';
  if LookupType = ServiceName then
   TaskHandle := WSAAsyncGetServByName(TaskWnd, ASYNC_EVENT, PChar(edService.Text),
                 PChar(ProtocolName), @AsyncBuff[0], MAXGETHOSTSTRUCT)
  else
   Taskhandle := WSAAsyncGetServByPort(TaskWnd, ASYNC_EVENT, htons(StrToInt(edService.Text)),
                 PChar(ProtocolName),@AsyncBuff[0], MAXGETHOSTSTRUCT);
  if TaskHandle = 0 then
  begin
   if LookUpType = ServiceName then
    Exception.Create('Call to WSAAsyncGetServByName failed...')
   else
    Exception.Create('Call to WSAAsyncGetServByPort failed...');
  end;
end;

procedure TfrmMain.btnProtocolLookUpClick(Sender: TObject);
var
 DummyValue, Code : integer;
begin
 if (length(edProtocol.Text) = 0) or (edProtocol.Text = '') then
  Exception.Create('You must enter a protocol name or protocol number!');
 val(edProtocol.Text, Dummyvalue, Code);
 if Code <> 0 then // this is not a numerical value ...it is a service name
  LookUpType := ProtocolName
 else
  LookUpType := ProtocolNumber;
  FillChar(AsyncBuff, SizeOf(AsyncBuff), #0);
  if LookUpType = ProtocolName then
   TaskHandle := WSAAsyncGetProtoByName(TaskWnd,
ASYNC_EVENT,PChar(edProtocol.Text),@AsyncBuff[0], MAXGETHOSTSTRUCT)
  else
   TaskHandle := WSAAsyncGetProtoByNumber(TaskWnd,ASYNC_EVENT,StrToInt(edProtocol.Text),
                 @AsyncBuff[0], MAXGETHOSTSTRUCT);
  if TaskHandle = 0 then
  begin
   if LookUpType = ProtocolName then
    Exception.Create('Call to WSAAsyncGetProtoByName failed...')
   else
    Exception.Create('Call to WSAAsyncGetProtoByNumber failed...');
  end;
end;
```

```
procedure TfrmMain.btnHostClick(Sender: TObject);
var
 Count, Len : integer;
 IPAddr : TInAddr;
begin
  if (length(edHost.Text) = 0) or (edHost.Text = '') then
   Exception.Create('You must enter a host name or IP Address!');
  Len := length(edHost.text);
  LookUpType := HostAddr;
  for Count := 1 to Len do
   if edHost.Text[Count] in ['a'..'z','A'..'Z'] then
   begin
    LookUpType := HostName;
    Break;
   end;
  FillChar(AsyncBuff, SizeOf(AsyncBuff), #0);
  if LookUpType = HostName then
   TaskHandle := WSAAsyncGetHostByName(TaskWnd, ASYNC_EVENT,
                 PChar(edHost.Text),@AsyncBuff[0], MAXGETHOSTSTRUCT)
  else
  begin
   IPAddr.S_addr := inet_addr(PChar(edHost.Text));
   TaskHandle := WSAAsyncGetHostByAddr(TaskWnd, ASYNC_EVENT, PChar(@IPAddr), 4, AF_INET,
                 @AsyncBuff[0], MAXGETHOSTSTRUCT);
  end;
  if TaskHandle = 0 then
  begin
   if LookUpType = HostName then
    Exception.Create('Call to WSAAsyncGetHostByName failed...')
   else
    Exception.Create('Call to WSAAsyncGetHostByAddr failed...');
  end;
end;

procedure TfrmMain.btnCloseClick(Sender: TObject);
begin
 Close;
end;

procedure TfrmMain.btnCancelClick(Sender: TObject);
begin
 if WSACancelAsyncRequest(THandle(TaskHandle)) = Integer(SOCKET_ERROR) then
  Exception.Create('Error ' + SysErrorMessage(WSAGetLastError))
 else
  Memo1.Lines.Add('Asynchronous Lookup Operation cancelled...');
end;

procedure TfrmMain.FormCreate(Sender: TObject);
begin
 gbHost.Enabled     := FALSE;
 gbService.Enabled  := FALSE;
 gbProtocol.Enabled := FALSE;
 btnCancel.Enabled  := FALSE;
 btnStop.Enabled    := FALSE;
end;

procedure TfrmMain.btnStartClick(Sender: TObject);
begin
 WSRunning := WSAStartUp($101, WSAData) = 0;
 if WSRunning then
```

```
  begin
   Memo1.Lines.Add('Winsock is running');
   TaskWnd := AllocateHWND(AsyncOp);
   gbHost.Enabled     := TRUE;
   gbService.Enabled  := TRUE;
   gbProtocol.Enabled := TRUE;
   btnCancel.Enabled  := TRUE;
   btnStart.Enabled   := FALSE;
   btnStop.Enabled    := TRUE;
  end
  else
   Memo1.Lines.Add('Winsock is not running');
end;

procedure TfrmMain.btnStopClick(Sender: TObject);
begin
 if WSRunning then
 begin
  WSACleanUp;
  DeAllocateHWND(TaskWND);
  gbHost.Enabled := FALSE;
  gbService.Enabled := FALSE;
  gbProtocol.Enabled := FALSE;
  btnCancel.Enabled := FALSE;
  btnStop.Enabled := FALSE;
  btnStart.Enabled := TRUE;
  WSRunning := FALSE;
 end;
end;

end.
```

function WSAAsyncGetHostByAddr Winsock2.pas

Syntax

WSAAsyncGetHostByAddr(hWnd: HWND; wMsg: u_int; addr: PChar; len, type_: Integer; buf: PChar; buflen: Integer): HANDLE; stdcall;

Description

This asynchronous function retrieves host information corresponding to an address.

Parameters

hWnd: The handle of the window that should receive a message when the asynchronous request completes

wMsg: The message to be received when the asynchronous request completes

addr: A pointer to the network address for the host. Host addresses are stored in network byte order.

len: The length of the address

*type_*: The type of the address (for example, AF_INET for an IP address)

buf: A pointer to the data area to receive the THostEnt data *buflen*: The size of data area in *buf*

Return Value

The return value will only specify if the operation started successfully; it will not indicate success or failure of the operation itself.

If the operation starts successfully, the function will return a nonzero value of type THandle. Otherwise, the function will return a zero. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAENETDOWN, WSAENOBUFS, WSAEFAULT, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, and WSANO_DATA.

The following errors may occur at the time of the function call, which indicate that the asynchronous operation could not start: WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

gethostbyaddr, WSACancelAsyncRequest

Example

See Listing 3-6 (program EX36).

Service Resolution

The blocking functions that resolve services are getservbyname() and getservbyport(), and their asynchronous equivalents are WSAAsyncGetServByName() and WSAAsyncGetServByPort(), respectively.

function getservbyname Winsock2.pas

Syntax

getservbyname(name, proto: PChar): PServEnt; stdcall;

Description

The function returns information for the requested service and retrieves a pointer to the TServEnt data structure that contains information corresponding to a service name and protocol. The TServEnt record is defined as follows in Winsock2.pas:

```
servent = record
  s_name: PChar;         // official service name
  s_aliases: PPChar;     // alias list
  s_port: Smallint;      // port number
  s_proto: PChar;        // protocol to use
end;
TServEnt = servent;
PServEnt = ^servent;
```

The pointer that you receive points to a record allocated by Winsock. Your application must not attempt to modify this record or free any of its parameters. This data is transient, so your application should copy any information that it needs before issuing any other Winsock function calls.

> **TIP:** To reinforce the previous point, remember the pointer you receive points to a record allocated by Winsock. Your application must never attempt to modify this record or free any of its parameters.

The members of this data structure are defined as:

s_name: The name of the service

s_aliases: An array of NULL-terminated strings populated with alternative names

s_port: Port number for the service. Port numbers are always in network byte order.

s_proto: The name of the protocol to use for the service

Parameters

name: A pointer to a NULL-terminated string representing the service name

proto: An optional pointer to a NULL-terminated string. If this argument is NIL, the function returns a pointer to the TServEnt record.

Return Value

If successful, the function will return a pointer to the TServEnt record. Otherwise, it will return an invalid pointer. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, WSANO_DATA, WSAEINPROGRESS, WSAEFAULT, and WSAEINTR.

See Appendix B for a detailed description of the error codes.

See Also

getservbyport, WSAAsyncGetServByName

Example

Listing 3-7 (program EX37) shows how to use the getservbyname() function.

Listing 3-7: Using getservbyname()

```
{ Example EX37 demonstrates the getservbyname() function.
  To execute this example you need to supply the service and protocol.
  For example, supply smtp and tcp for the service and protocol, respectively.

  EX37 smtp tcp
```

```
  The getservbyname() function gets service information corresponding
  to a service name and protocol.
  The function returns a pointer to the TServent which contains
  the name(s) and service number which correspond to the given service name.
  All strings are NULL terminated.}

program EX37;

{$APPTYPE CONSOLE}
uses
  Dialogs,
  SysUtils,
  Winsock2;

const
 WSVersion : Word = $101;

var
 WSAData : TWSAData;
 Servent : PServent;
 ProtocolName,
 ServiceName : String;
 Alias : PPChar;
 ServiceCount : Integer;
begin
  if ParamCount < 2 then
  begin
    WriteLn('Error - missing parameter(s)! '+#10#13+'Please supply a service name and
             protocol (e.g. ftp tcp');
    Halt;
  end;
  ServiceName  := ParamStr(1);
  ProtocolName := ParamStr(2);
  if WSAStartUp(Word(WSVersion), WSAData) = 0 then // yes, Winsock does exist ...
  try
    Servent := getservbyname(PChar(ServiceName),PChar(ProtocolName));
    if Servent <> NIL then
    begin
      WriteLn(Format('Official Service Name is %s',[Servent^.s_name]));
      WriteLn(Format('Service Port is %d in network order',[Servent^.s_port]));
      WriteLn(Format('Service Port is %d in host order',[ntohs(Servent^.s_port)]));
      WriteLn(Format('Protocol is %s',[Servent^.s_proto]));
      WriteLn('List of Aliases');
      ServiceCount := 0;
      Alias := Servent^.s_aliases;
      while Alias^ <> nil do
      begin
        Inc(ServiceCount);
        WriteLn(Format('Service Name [%d] is %s',[ServiceCount, Alias^]));
        Inc(Alias);
      end;
      if ServiceCount = 0 then WriteLn('None');
    end else
     WriteLn(Format('Call to getservbyname() failed with error: %s',
            [SysErrorMessage(WSAGetLastError)]));
  finally
    WSACleanUp;
  end else
   WriteLn('Failed to load Winsock.');
end.
```

function getservbyport   Winsock2.pas

Syntax

getservbyport(port: Integer; proto: PChar): PServEnt; stdcall;

Description

This function retrieves information about a service based on the port number and protocol. Your application must not attempt to modify this record or free any of its components. This data is transient, so your application should copy any information that it needs before calling any other Winsock function calls.

Parameters

port: The port for a service, which must be in network byte order

proto: An optional pointer to a protocol name. If the argument is NIL, the function returns the first service entry that matches the *port* argument with the *s_port* field of the TServEnt record. Otherwise, getservbyport() matches both the *port* and the *proto* fields.

Return Value

If successful, getservbyport() will return a pointer to the TServEnt record that is allocated by Winsock. Otherwise, it will return an invalid pointer. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, WSANO_DATA, WSAEINPROGRESS, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

getservbyname, WSAAsyncGetServByPort

Example

Listing 3-8 (program EX38) shows how to use the getservbyport() function.

Listing 3-8: Using getservbyport()

```
{ Example EX38 demonstrates the getservbyport() function.

  To execute this example, you need to supply the service and protocol.
  For example, supply smtp and tcp for the service and protocol, respectively.

  EX38 21 tcp


  The getservbyport() function gets service information corresponding
  to a service name and protocol.
  The function returns a pointer to the Tservent, which contains
  the name(s) and service number that correspond to the given service name.
  All strings are NULL terminated.}
```

```
program EX38;

{$APPTYPE CONSOLE}
uses
  Dialogs,
  SysUtils,
  Winsock2;

const
 WSVersion : Word = $101;

var
 WSAData : TWSAData;
 Servent : PServent;
 ProtocolName: String;
 Alias : PPChar;
 Port,
 ServiceCount    : Integer;
begin
 Port := 0;
 if ParamCount < 2 then
 begin
  WriteLn('Error - missing service port or protocol! '+#10#13+'Please supply a service name
           and protocol (e.g. 21 tcp)');
  Halt;
 end;
 try
  Port := StrToInt(ParamStr(1));
 except on EConvertError do
  begin
   WriteLn(Format('Invalid Port %d',[Port]));
   Halt;
  end;
 end;
 ProtocolName := ParamStr(2);
 if WSAStartUp(WSVersion, WSAData) = 0 then // yes, Winsock does exist ...
 try
  Servent := getservbyport(htons(Port),PChar(ProtocolName));
  if Servent <> NIL then
  begin
    WriteLn(Format('Official Service Name is %s',[Servent^.s_name]));
    WriteLn(Format('Service Port is %d in network order',[Servent^.s_port]));
    WriteLn(Format('Service Port is %d in host order',[ntohs(Servent^.s_port)]));
    WriteLn(Format('Protocol is %s',[Servent^.s_proto]));
    ServiceCount := 0;
    Alias := Servent^.s_aliases;
    while Alias^ <> nil do
    begin
      Inc(ServiceCount);
      WriteLn(Format('Service Name [%d] is %s',[ServiceCount, Alias^]));
      Inc(Alias);
    end;
    if ServiceCount = 0 then WriteLn('None');
  end else
   WriteLn(Format('Call to getservbyport() failed with error: %s',
[SysErrorMessage(WSAGetLastError)]));
 finally
  WSACleanUp;
```

3 Chapter

```
  end else
   WriteLn('Failed to load Winsock.');
 end.
```

function WSAAsyncGetServByName   Winsock2.pas

Syntax

WSAAsyncGetServByName(hWnd: HWND; wMsg: u_int; name, proto, buf: PChar; buflen: Integer): HANDLE; stdcall;

Description

This asynchronous function retrieves service information corresponding to a service name and port.

Parameters

hWnd: The handle of the window that should receive a message when the asynchronous request completes

wMsg: The message to be received when the asynchronous request completes

name: A pointer to a NULL-terminated string containing the service name

proto: A pointer to a protocol name, which may be NIL. If the argument is NIL, the function searches for the first service entry for which *s_name* or one of the *s_aliases* matches the given name above. Otherwise, WSAAsyncGetServByName() matches both *name* and *proto*.

buf: A pointer to the buffer to receive the PServEnt record

buflen: The length of the buffer, *buf*

Return Value

The return value will only specify if the operation started successfully; it will not indicate success or failure of the operation itself. If the operation starts successfully, the function will return a nonzero value of type THandle. Otherwise, the function will return a zero to indicate a failure. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAENETDOWN, WSAENOBUFS, WSAEFAULT, WSAHOST_NOT_FOUND, WSATRY_ AGAIN, WSANO_RECOVERY, and WSANO_DATA.

The following errors may occur at the time of the function call, which indicate that the asynchronous operation could not start: WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

getservbyname, WSACancelAsyncRequest

Example

See Listing 3-6 (program EX36).

function WSAAsyncGetServByPort   Winsock2.pas

Syntax

WSAAsyncGetServByPort(hWnd: HWND; wMsg: u_int; port: Integer; proto, buf: PChar; buflen: Integer): HANDLE; stdcall;

Description

This asynchronous function retrieves service information corresponding to a port and protocol.

Parameters

hWnd: The handle of the window that should receive a message when the asynchronous request completes

wMsg: The message to be received when the asynchronous request completes

port: The port for the service in network byte order

proto: A pointer to a protocol name. This may be NIL, in which case WSAAsyncGetServByPort() will search for the first service entry for which *s_port* matches the given *port*. Otherwise, WSAAsyncGetServByPort(), matches both *port* and *proto*.

buf: A pointer to the data area to receive the TServEnt data

buflen: The size of data area *buf*

Return Value

The return value will only specify if the operation started successfully; it will not indicate success or failure of the operation itself.

If the operation starts successfully, the function will return a nonzero value of type THandle. Otherwise, the function will return a zero. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAENETDOWN, WSAENOBUFS, WSAEFAULT, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, and WSANO_DATA.

The following errors may occur at the time of the function call, which indicate that the asynchronous operation could not start: WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

getservbyport, WSACancelAsyncRequest

Example

See Listing 3-6 (program EX36).

Protocol Resolution

Before using a service, it is necessary to resolve the underlying protocol first. Services such as FTP, SMTP, POP3, HTTP, and many others use TCP as their transport protocol, which should be present. Other services, like TFTP (Trivial File Transfer Protocol), use UDP instead. Some services, such as DNS, are agnostic in that they can use either UDP or TCP.

The blocking functions that resolve services are getprotobyname() and getprotobynumber(), and their asynchronous equivalents are WSAAsyncGetProtoByName() and WSAAsyncGetProtoByNumber(), respectively.

function getprotobyname Winsock2.pas

Syntax

getprotobyname(name: PChar): PProtoEnt; stdcall;

Description

This function retrieves protocol information corresponding to a protocol name. The protoent record is defined in Winsock2.pas as follows:

```
protoent= record
  p_name: PChar;          // official protocol name
  p_aliases: PPChar;      // alias list
  p_proto: Smallint;      // protocol #
end;
TProtoEnt = protoent;
PProtoEnt = ^protoent;
```

The members of this data structure are defined as:

p_name: Official name of the protocol

p_aliases: An array of NULL-terminated strings that can hold alternative names

p_proto: The protocol number in host byte order

The PProtoEnt value that is returned points to a record that is allocated by Winsock. The data is transient, so the application should copy any information that it needs before calling any other Winsock function calls.

TIP: The application must never attempt to modify this record or to free any of its components.

Parameters

name: A pointer to a NULL-terminated protocol name

Return Value

If successful, the function will return a pointer to the PProtoEnt record. Otherwise, it will return NIL. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED,

WSAENETDOWN, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, WSANO_DATA, WSAEINPROGRESS, WSAEFAULT, and WSAEINTR.

See Appendix B for a detailed description of the error codes.

See Also

getprotobynumber, WSAAsyncGetProtoByName

Example

Listing 3-9 (program EX39) shows how to use the getprotobyname() function.

Listing 3-9: Using getprotobyname()

```
{ Example EX39 demonstrates the getprotobyname() function.

  To execute this example, you need to supply the protocol.
  For example, supply tcp for the protocol.

  EX39 tcp


  The getprotobyname() function gets protocol information corresponding to a
  protocol name.
  The getprotobyname() function returns a pointer to the TProtoEnt record,which
  contains the name(s) and protocol number that correspond to the given
  protocol name. All strings are NULL terminated.}

program EX39;

{$APPTYPE CONSOLE}
uses
  Dialogs,
  SysUtils,
  Winsock2;

const
 WSVersion : Word = $101;

var
 WSAData: TWSAData;
 Protocol: PProtoEnt;
 ProtocolName: String;
 Alias: PPChar;
 ProtocolCount: Integer;
begin
 if ParamCount < 1 then
 begin
  WriteLn('Error - missing protocol name! '+#10#13+'Please supply a protocol name(e.g. tcp');
  Halt;
 end;
 ProtocolName := ParamStr(1);
 if WSAStartUp(WSVersion, WSAData) = 0 then // yes, Winsock does exist ...
 try
  Protocol := getprotobyname(PChar(ProtocolName));
  if Protocol <> NIL then
  begin
   with Protocol^ do
   begin
```

```
     WriteLn(Format('Protocol Name is %s',[Protocol.p_name]));
     WriteLn(Format('Protocol Number is %d',[Protocol.p_proto]));
    end;
    ProtocolCount := 0;
    WriteLn(Format('The %s Protocol has the following aliases',[Protocol.p_name]));
    Alias := Protocol^.p_aliases;
    while Alias^ <> nil do
    begin
      Inc(ProtocolCount);
      WriteLn(Format('Service Name [%d] is %s',[ProtocolCount, Alias^]));
      Inc(Alias);
    end;
    if ProtocolCount = 0 then WriteLn('None');
   end
   else
    WriteLn(Format('Call to getprotobyname() failed with error: %s',
[SysErrorMessage(WSAGetLastError)]));
 finally
  WSACleanUp;
 end else
  WriteLn('Failed to load Winsock.');
end.
```

function getprotobynumber Winsock2.pas

Syntax

getprotobynumber(proto: Integer): PProtoEnt; stdcall;

Description

This function retrieves information for a protocol corresponding to a protocol number and returns a pointer to a protoent record, as described previously in getprotobyname(). As in previous examples, your application must not attempt to modify this record or free any of its components. Since the data is transient, your application should copy any information that it needs before calling any other Winsock function calls.

Parameters

proto: A protocol number in host byte order

Return Value

If successful, the function will return a pointer to the PProtoEnt record. Otherwise, it will return NIL. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, WSANO_DATA, WSAEINPROGRESS, and WSAEINTR.

See Appendix B for a detailed description of the error codes.

See Also

getprotobyname, WSAAsyncGetProtoByNumber

Example

Listing 3-10 (program EX310) shows how to use the getprotobynumber() function.

Listing 3-10: Using getprotobynumber()

```
{ Example EX310 demonstrates the getprotobynumber() function.

  To execute this example, you need to supply the protocol number.
  For example, supply 6 for the tcp protocol.

  EX310 6


  The getprotobynumber() function gets protocol information
  corresponding to a protocol number.
  This function returns a pointer to a ProtoEnt record.
  The contents of the structure correspond to the given protocol number.}

program EX310;

{$APPTYPE CONSOLE}
uses
  Dialogs,
  SysUtils,
  Winsock2;

const
 WSVersion : Word = $101;

var
 WSAData : TWSAData;
 Protocol : PProtoEnt;
 Alias  : PPChar;
 ProtoNumber,
 ProtocolCount : Integer;
begin
 if ParamCount < 1 then
 begin
   WriteLn('Error - missing protocol number! '+#10#13+'Please supply a protocol number.');
   Halt;
 end;
 ProtoNumber := 0;
 try
  ProtoNumber := StrToInt(ParamStr(1));
 except on EConvertError do
  begin
   ShowMessage(Format('Invalid input %s',[ParamStr(1)]));
   Halt;
  end;
 end;
 if WSAStartUp(Word(WSVersion), WSAData) = 0 then // yes, Winsock does exist ...
 try
  Protocol := getprotobynumber(ProtoNumber);
  if Protocol <> NIL then
  begin
   with Protocol^ do
   begin
    WriteLn(Format('Protocol is %s',[Protocol.p_name]));
```

```
     WriteLn(Format('Protocol number is %d',[Protocol.p_proto]));
    end;
    ProtocolCount := 0;
    WriteLn(Format('The %s Protocol has the following aliases',[Protocol.p_name]));
    Alias := Protocol^.p_aliases;
    while Alias^ <> nil do
    begin
      Inc(ProtocolCount);
      WriteLn(Format('Protocol Name [%d] is %s',[ProtocolCount, Alias^]));
      Inc(Alias);
    end;
   end
   else
    WriteLn(Format('Call to getprotobynumber() failed with error: %s',
                  [SysErrorMessage(WSAGetLastError)]));
  finally
   WSACleanUp;
  end else
   WriteLn('Failed to load Winsock.');
 end.
```

function WSAAsyncGetProtoByName *Winsock2.pas*

Syntax

WSAAsyncGetProtoByName(hWnd: HWND; wMsg: u_int; name, buf: PChar; buflen: Integer): HANDLE; stdcall;

Description

This asynchronous function retrieves protocol information corresponding to a protocol name.

Parameters

hWnd: The handle of the window that should receive a message when the asynchronous request completes

wMsg: The message to receive when the asynchronous request completes

name: A pointer to the NULL-terminated protocol name to resolve

buf: A pointer to the data area to receive the protoent data

buflen: The size of data area *buf*

Return Value

The return value will only indicate if the operation started successfully; it will not indicate success or failure of the operation itself. If the operation starts successfully, the function will return a nonzero value of type THandle. Otherwise, the function will return a value of zero. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAENETDOWN, WSAENOBUFS, WSAEFAULT, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, and WSANO_DATA.

The following errors may occur at the time of the function call, which indicate that the asynchronous operation could not start: WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

getprotobyname, WSACancelAsyncRequest

Example

See Listing 3-6 (program EX36).

function WSAAsyncGetProtoByNumber Winsock2.pas

Syntax

WSAAsyncGetProtoByNumber (hWnd: HWND; wMsg: u_int; number: Integer; buf: PChar; buflen: Integer): HANDLE; stdcall;

Description

This asynchronous function retrieves protocol information corresponding to a protocol number.

Parameters

hWnd: The handle of the window that should receive a message when the asynchronous request completes

wMsg: The message to receive when the asynchronous request completes

number: The protocol number to be resolved, in host byte order

buf: A pointer to the data area to receive the TProtoEnt data

buflen: The size of data area *buf*

Return Value

The return value will only indicate if the operation started successfully; it will not indicate success or failure of the operation itself.

If the operation starts successfully, the function will return a nonzero value of type THandle. Otherwise, the function will return a value of zero. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAENETDOWN, WSAENOBUFS, WSAEFAULT, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, and WSANO_DATA.

The following errors may occur at the time of the function call, which indicate that the asynchronous operation could not start: WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

getprotobynumber, WSACancelAsyncRequest

Example

See Listing 3-6 (program EX36).

Canceling an Outstanding Asynchronous Call

It is sometimes necessary to cancel an outstanding asynchronous call. You might want to abort the call for any reason. For example, the asynchronous call was taking too long to complete, or the user of your application might want to cancel the call before closing down the application. A call to WSACancelAsync-Request() cancels any asynchronous call that is still being serviced.

function WSACancelAsyncRequest Winsock2.pas

Syntax

WSACancelAsyncRequest(hAsyncTaskHandle: THandle): Integer; stdcall;

Description

This function cancels an incomplete asynchronous operation.

Parameters

hAsyncTaskHandle: A handle to identify the asynchronous operation to cancel, which is the handle previously assigned for the asynchronous operation

Return Value

If successful, the function will return a value of zero. Otherwise, it will return the value SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINVAL, WSAEINPROGRESS, and WSAEALREADY.

An attempt to cancel an existing asynchronous operation can fail with an error code of WSAEALREADY for two reasons. First, the original operation has already completed and the application has dealt with the resultant message. Second, the original operation has already completed but the resultant message is still waiting in the application message queue.

See Appendix B for a detailed description of the error codes.

See Also

WSAAsyncGetHostByAddr, WSAAsyncGetHostByName, WSAAsyncGetProto-ByName, WSAAsyncGetProtoByNumber, WSAAsyncGetServByName, WSAAsyncGetServByPort

Example

See Listing 3-6 (program EX36).

Summary

In this chapter, you have learned how to perform translation from host order to network order and vice versa, and resolution of a host, service, and protocol using the blocking and asynchronous functions. The next chapter focuses on Winsock 2 style resolution that is protocol independent and is more flexible and powerful than the old style Winsock 1.1 resolution functions.

Chapter 4

Winsock 2 Resolution

In the last chapter, we discussed Winsock 1.1 style resolution. Because of its simplicity and proven technology, the majority of existing applications still use Winsock 1.1 resolution functions. Indeed, the Winsock 2 extensions do not replace the original functions, but rather, they enhance the existing repertoire by providing the means to register a service on the server side and perform queries on the client side without the need to resolve ports, host names, and services. As part of its armory, Winsock 2 provides tools to enumerate transport protocols and name spaces that are required to register and query a service.

Although the Winsock 2 resolution and registration functions are more complex than we have seen so far, mastering the implementation details of these functions is a worthwhile investment on your part. One important reason is that their inclusion will help make your application user friendly. These resolution APIs also perform protocol-independent name registration.

First, however, we are going to skim through the new translation functions that Winsock 2 introduced to extend the scope of the existing Winsock 1.1 translation tools. Then we will explore how to install a service, advertise a service on the server side, and generate service queries from the client.

Translation Functions

Like their Winsock 1.1 peers, the following functions (which Winsock 2 designates with a WSA prefix to distinguish them from their Winsock 1.1 cousins) perform operations that transcend the byte ordering incompatibility that exists on the Internet. We have already covered this topic in some depth in the previous chapter; let's briefly examine these functions.

function WSAHtonl   Winsock2.pas

Syntax

WSAHtonl(s: TSocket; hostlong: u_long; lpnetlong: pu_long): u_int; stdcall;

Description

This function takes a 32-bit number in host byte order and returns a 32-bit number pointed to by the *lpnetlong* parameter in network byte order for the socket descriptor *s*.

Parameters

s: A socket descriptor

hostlong: A 32-bit number in host byte order

lpnetlong: A pointer to a 32-bit number in network byte order

Return Value

If the function succeeds, it will return zero. If the function fails, the return value will be SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAENOTSOCK, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

htonl, htons, ntohl, ntohs, WSAHtons, WSANtohl, WSANtohs

Example

Listing 4-1 (program EX41) shows how to use the WSAHtons(), WSAHtonl(), WSANtohs(), and WSANtohl() functions.

Listing 4-1: Converting numbers from network order to host order

```
{ Example EX41 demonstrates how to convert numbers from network to host order and vice versa.

To execute this example, you need to supply a number. For example, to translate a number, say
21, type the following and press ENTER on the command line:

  EX41 21


  The following functions are used: WSAhtons, WSAhtonl,
  WSAntohs, and WSAntohl.
}
program EX41;

{$APPTYPE CONSOLE}

uses
  Dialogs,
  SysUtils,
  Winsock2,
  Windows;

const
 WSVersion = $0202;

var
 WSAData : TWSAData;
```

```
  Netlong : DWORD;
  Netshort: WORD;
  Value   : Cardinal;
  Code    : Integer;
  skt     : TSocket;
  Res     : Integer;
begin
  if ParamCount < 1 then
  begin
   WriteLn('Missing value. Please input a numerical value.');
   Halt;
  end;
// Convert input to a numerical value ...
  Val(ParamStr(1), Value, Code);
// Check for bad conversion
  if Code <> 0 then
  begin
   MessageDlg(Format('Error at position: %d',[Code]), mtError, [mbOk], 0);
   Halt;
  end;
  if WSAStartUp(Word(WSVersion), WSAData) = 0 then // yes, Winsock does exist ...
  try
   skt := socket(AF_INET, SOCK_STREAM,0);
   if skt = SOCKET_ERROR then
   begin
    WriteLn(Format('Call to socket() failed with error: %s',
[SysErrorMessage(WSAGetLastError)]));
   end else
   begin
   {mvb you're not checking the result of the WSA functions ?!}
    Res := WSAhtonl(skt, Value, Netlong);
    if Res = SOCKET_ERROR then
     WriteLn(Format('Call to WSAhtonl() failed with error: %s',
[SysErrorMessage(WSAGetLastError)]))
    else
     WriteLn(Format('Using WSAhtonl() the value %d converted from host order to network order
              (long format)  = %d',[Value, Netlong]));
    Res := WSAhtons(skt, Value, Netshort);
    if Res = SOCKET_ERROR then
     WriteLn(Format('Call to WSAhtons() failed with error: %s',
          [SysErrorMessage(WSAGetLastError)]))
    else
     WriteLn(Format('Using WSAhtons() the value %d converted from host order to network order
              (short format) = %d',[Value, Netshort]));
    Res := WSAntohl(skt, Value, Netlong);
    if Res = SOCKET_ERROR then
     WriteLn(Format('Call to WSAntohl() failed with error: %s',
          [SysErrorMessage(WSAGetLastError)]))
    else
     WriteLn(Format('Using WSAntohl() the value %d converted from network order to host order
              (long format)  = %d',[Value, Netlong]));
    Res := WSAntohs(skt, Value, Netshort);
    if Res = SOCKET_ERROR then
     WriteLn(Format('Call to WSAntohs() failed with error: %s',
          [SysErrorMessage(WSAGetLastError)]))
    else
     WriteLn(Format('Using WSAntohs() the value %d converted from network order to host order
          (short format) = %d',[Value, Netshort]));
    closesocket(skt);
   end;
```

```
  finally
   WSACleanUp;
  end
  else WriteLn('Failed to initialize Winsock.');
end.
```

function WSAHtons Winsock2.pas

Syntax

WSAHtons(s: TSocket; hostshort: u_short; lpnetshort: pu_short): u_int; stdcall;

Description

This function converts a 16-bit number in host byte order and returns a 16-bit number pointed to by the *lpnetshort* parameter in network byte order for the socket descriptor *s*.

Parameters

s: A socket descriptor

hostshort: A 16-bit number in host byte order

lpnetshort: A pointer to a 16-bit number in network byte order

Return Value

If the function succeeds, it will return zero. If the function fails, the return value will be SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAE-NETDOWN, WSAENOTSOCK, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

htonl, htons, ntohl, ntohs, WSAHtonl, WSANtohl, WSANtohs

Example

See Listing 4-1 (program EX41).

function WSANtohl Winsock2.pas

Syntax

WSANtohl(s: TSocket; netlong: u_long; lphostlong: pu_long): u_int; stdcall;

Description

This routine takes a 32-bit number in network byte order for the socket *s* and returns a 32-bit number pointed to by the *lphostlong* parameter in host byte order.

Parameters

s: A descriptor identifying a socket

netlong: A 32-bit number in network byte order

lphostlong: A pointer to a 32-bit number in host byte order

Return Value

If the function succeeds, it will return zero. If the function fails, it will return a value of SOCKET_ERROR(). To retrieve the specific error code, call the function WSAGetLastError. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAENOTSOCK, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

htonl, htons, ntohl, ntohs, WSAHtonl, WSAHtons, WSANtohs

Example

See Listing 4-1 (program EX41).

function WSANtohs Winsock2.pas

Syntax

WSANtohs(s: TSocket; netshort: u_short; lphostshort: pu_short): u_int; stdcall;

Description

This routine takes a 16-bit number in network byte order for the socket *s* and returns a 16-bit number pointed to by the *lphostshort* parameter in host byte order.

Parameters

s: A socket descriptor

netshort: A 16-bit number in network byte order

lphostshort: A pointer to a 16-bit number in host byte order

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAENOTSOCK, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

htonl, htons, ntohl, ntohs, WSAHtonl, WSAHtons, WSANtohl

Example

See Listing 4-1 (program EX41).

Address and String Conversion Functions

In this section, we will examine briefly the WSAAddressToString() and WSA-StringToAddress() functions. Neither function exists in Winsock 1.1. These functions convert a TSockAddr data structure into a string and vice versa using the specified transport protocol. In a later section, we'll discuss protocol independence, which is a basic feature of the Winsock 2 architecture.

As well as specifying a transport protocol, such as TCP and UDP, we need to specify the address family that supports the transport protocol. At the time of publication, Winsock 2 supports only AF_INET and AF_ATM address families with these conversion functions. These functions are defined below.

function WSAAddressToString Winsock2.pas

Syntax

WSAAddressToString (lpsaAddress: PSockAddr; dwAddressLength: DWORD; lpProtocolInfo: PWSAPROTOCOL_INFO; lpszAddressString: PChar; lpdwAddress-StringLength: PDWORD): u_int; stdcall;

Description

This function converts all components of a TSockAddr record into a readable numeric string representation of the address. To translate the structure on the specified transport protocol, you must supply the corresponding WSA-PROTOCOL_INFO record in the *lpProtocolInfo* parameter. The TSockAddr data structure is defined in Winsock2.pas as follows:

```
  sockaddr = record
  sa_family: u_short;              // address family
  sa_data: array [0..13] of Char;    // up to 14 bytes of direct address
end;

TSockAddr = sockaddr;
PSockAddr = ^sockaddr;
```

Parameters

lpsaAddress: A pointer to a TSockAddr record to translate into a string

dwAddressLength: The length of the address, which may vary in size with different protocols

lpProtocolInfo: An optional pointer to the WSAPROTOCOL_INFO record for the transport protocol. If this is NIL, the function uses the first available provider of the protocol supporting the address family pointed to in *lpsa-Address*. In the case of TCP/IP, the address family would be AF_INET.

lpszAddressString: A buffer that receives the human-readable address string

lpdwAddressStringLength: On input, the length of the *lpszAddressString* buffer. On output, returns the length of the string actually copied into the buffer. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT, and this parameter is updated with the required size in bytes.

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR(). To retrieve the specific error code, call the function WSA-GetLastError(). Possible error codes are WSAEFAULT, WSAEINVAL, WSANOTINITIALISED, and_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSAStringToAddress

Example

Listing 4-2 (program EX42) shows how to use the address and string conversion functions.

Listing 4-2: Using WSAStringToAddress() and WSAAddressToString()

```
{
  Example EX42 demonstrates how to use WSAStringToAddress and
  WSAAddressToString.
  No command line parameters are required.

}
program EX42;

{$APPTYPE CONSOLE}

uses
  Windows,
  WinSock2,
  ComObj,
  SysUtils;

const
 HostAddress = '127.0.0.1';


var
  WSAData: TWSAData;

  AddrStr: array[0..MAXGETHOSTSTRUCT – 1] of char;
  AddrSize: Integer;
  Res: DWORD;
  LocalAddr: TSockAddrIn;

begin
  if WSAStartUp($0202, WSAData) = 0 then
```

```
  try
    LocalAddr.sin_family := AF_INET;
    AddrSize := SizeOf(TSockAddrIn);
    Res := WSAStringToAddress(PChar(HostAddress), AF_INET, NIL, @LocalAddr, AddrSize);
    if Res = SOCKET_ERROR then
      WriteLn('Call to WSAStringToAddress() failed with error: ' +
              SysErrorMessage(WSAGetLastError))
    else
    begin
      WriteLn('Address = ' + String(inet_ntoa(LocalAddr.sin_addr)));
      Res := WSAAddressToString(@LocalAddr, SizeOf(TSockAddrIn), NIL, @AddrStr,
                                Cardinal(AddrSize));
      if Res = SOCKET_ERROR then
        WriteLn('Call to WSAAddressToString() failed with error: ' +
                SysErrorMessage(WSAGetLastError))
      else
        WriteLn('Host = ' + String(AddrStr));
     end;
   finally
    WSACleanUp;
  end
  else WriteLn('Windows Sockets initialization failed.');
end.
```

function WSAStringToAddress   Winsock2.pas

Syntax

WSAStringToAddress(AddressString: PChar; AddressFamily: u_int; lpProtocolInfo: PWSAPROTOCOL_INFO; lpAddress: PSockAddr; lpAddressLength: PInt): u_int; stdcall;

Description

This function converts an address in a numeric string to a socket address record. Such a record is required by Winsock functions that use the TSockAddr data structure. The function will set default values in place of missing fields of the address. For example, a missing port number will have the default value of zero. To use a particular transport provider, such as TCP/IP, to do the conversion, you should supply the corresponding pointer to the WSAPROTOCOL_INFO record in the *lpProtocolInfo* parameter.

Parameters

AddressString: Pointer to the NULL-terminated string to convert

AddressFamily: The address family to which the string belongs (for example, AF_INET for TCP/IP)

lpProtocolInfo: An optional pointer to the WSAPROTOCOL_INFO record associated with the provider to be used. If this is NIL, the function will use the first available provider of the first protocol that supports the supplied *AddressFamily* parameter.

lpAddress: A buffer filled with a single TSockAddr record.

lpAddressLength: The length of the *lpAddress* buffer to hold the TSockAddr record. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it returns SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAEFAULT, WSAEINVAL, WSANOTINITIALISED, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSAAddressToString

Example

See Listing 4-2 (program EX42).

Enumerating Network Protocols

In some cases, it is necessary to determine what network protocols are available before your application can run. Of those installed protocols that are available on the machine, you will need to determine which protocol has the desired properties that match the application's requirements.

Occasionally, the protocol that your application requires may not be present, in which case you will have to install the required protocol. We'll discuss how to install a new protocol later in this chapter.

To determine what protocols are available on your machine, you use the WSAEnumProtocols() function to enumerate these protocols. The WSAEnumProtocols() function returns an array of WSAPROTOCOL_INFO records, each of which corresponds to a description for an installed protocol. On Windows machines, TCP/IP is the default network protocol, and it will usually have two IP entries, TCP and UDP.

The WSAPROTOCOL_INFO record, which is defined in Winsock2.pas, is as follows:

```
_WSAPROTOCOL_INFO = record
 dwServiceFlags1: DWORD;
 dwServiceFlags2: DWORD;
 dwServiceFlags3: DWORD;
 dwServiceFlags4: DWORD;
 dwProviderFlags: DWORD;
 ProviderId: TGUID;
 dwCatalogEntryId: DWORD;
 ProtocolChain: WSAPROTOCOLCHAIN;
 iVersion: Integer;
 iAddressFamily: Integer;
```

```
    iMaxSockAddr: Integer;
    iMinSockAddr: Integer;
    iSocketType: Integer;
    iProtocol: Integer;
    iProtocolMaxOffset: Integer;
    iNetworkByteOrder: Integer;
    iSecurityScheme: Integer;
    dwMessageSize: DWORD;
    dwProviderReserved: DWORD;
    szProtocol: array [0..WSAPROTOCOL_LEN] of WideChar;
  end;
  WSAPROTOCOL_INFOW = _WSAPROTOCOL_INFOW;
  LPWSAPROTOCOL_INFOW = ^WSAPROTOCOL_INFOW;
  TWsaProtocolInfoW = WSAPROTOCOL_INFOW;
  PWsaProtocolInfoW = LPWSAPROTOCOL_INFOW;
```

Often, you do not know the exact number of available transport protocols that are installed on your machine, and therefore the size of the buffer in which to store the array of WSAPROTOCOL_INFO records is unknown. A call to WSAEnumProtocols() will fail with the error of WSAENOBUFFS. To rectify this defect, you must call WSAEnumProtocols() twice. The first call is to discover the size of the buffer to hold the array of WSAPROTOCOL_INFO entries. To get this magic value, first set the buffer, *lpProtocolBuffer*, to NIL, and then set the length of the buffer, *lpdwBufferLength*, to zero. With these values set, the function will always fail with an error of WSAENOBUFS, but the *lpdwBufferLength* parameter will contain the correct buffer size. You use this buffer size in the second call to WSAEnumProtocols(). On successful completion of the second call, WSAEnumProtocols() returns an array of network protocols installed on your machine.

Examining the WSAPROTOCOL_INFO record above, it does look overwhelming with so much detail. From our perspective, the most useful fields to use are *dwServiceFlags1*, *iProtocol*, *iSocketType*, and *iAddressFamily*. Later in the chapter, we'll demonstrate how to use these fields (program EX43 in Listing 4-3). To determine if an installed protocol supports a property that your application requires, you should perform an AND bitwise operation on that property. Table 4-1 (following Listing 4-3) shows a list of properties for all protocols. For example, if your application requires a connectionless service, you would select XPI_CONNECTIONLESS from Table 4-1 and perform an AND operation on this property with the *dwServiceFlags1* field. If the AND operation yields a nonzero value, the protocol does support a connectionless service; otherwise, it does not.

The WSAEnumProtocols() function also enumerates protocol chains that may be present on the machine. Protocol chains link layered protocol entries together. Like a chain in real life, the protocol chain has an anchor, which is like the base layer protocol. In Windows, the TCP/IP protocol is usually the anchor to which other protocols can attach to form a chain of layered protocols. (We say that the protocols are layered because they lie on top of each other.) However, we will not discuss the layered protocol chains, as these are in the realm of the Service Provider Interface (SPI) (see Chapter 1 for the architecture of Winsock 2), which is beyond the scope of this book.

Before we leave the topic of protocol chains, let's discuss a hypothetical application that uses protocol chains. Say, for example, that you want to add a simple security scheme to your company's web site. Unfortunately, there isn't a product on the market that matches your requirement. So, you design and add your own security protocol to scan packets of data sent by the browser clients. To achieve a scanning scheme that is transparent to the clients, you add your simple security protocol via the SPI to link with the TCP/IP service provider, which in this scenario is the anchor or base of the protocol chain.

Listing 4-3 shows how to use WSAEnumProtocols(), and there is a working example in program EX43.

Listing 4-3: Using WSAEnumProtocols()

```
{
  Example EX43 demonstrates how to use WSAEnumProtocols.

  No command line parameters are required.
}
program EX43;

{$APPTYPE CONSOLE}

uses
  Windows,
  WinSock2,
  ComObj,
  SysUtils;

var
  WSAData: TWSAData;
  BufferLength: DWORD;
  Buffer, Info: PWSAProtocolInfo;
  I, Count: Integer;
  ExtendedInfo: Boolean;

function ByteOrderToString(O: DWORD): string;
begin
  case O of
    BIGENDIAN: Result := 'Big Endian';
    LITTLEENDIAN: Result := 'Little Endian';
  else
    Result := 'Unknown';
  end;
end;

function SocketTypeToString(T: DWORD): string;
begin
  case T of
    SOCK_STREAM: Result := 'Stream';
    SOCK_DGRAM: Result := 'Datagram';
  else
    Result := 'Unknown';
```

```
  end;
end;

function AddressFamilyToString(F: DWORD): string;
begin
  case F of
    AF_UNIX: Result := 'local to host (pipes, portals';
    AF_INET: Result := 'internetwork: UDP, TCP, etc.';
    AF_IMPLINK: Result := 'arpanet imp addresses';
    AF_PUP: Result := 'pup protocols: e.g. BSP';
    AF_CHAOS: Result := 'mit CHAOS protocols';
    AF_NS: Result := 'XEROX NS protocols';
 // AF_IPX: Result := 'IPX protocols: IPX, SPX, etc.';
    AF_ISO: Result := 'ISO protocols';
 // AF_OSI: Result := 'OSI is ISO';
    AF_ECMA: Result := 'european computer manufacturers';
    AF_DATAKIT: Result := 'datakit protocols';
    AF_CCITT: Result := 'CCITT protocols, X.25 etc';
    AF_SNA: Result := 'IBM SNA';
    AF_DECnet: Result := 'DECnet';
    AF_DLI: Result := 'Direct data link interface';
    AF_LAT: Result := 'LAT';
    AF_HYLINK: Result := 'NSC Hyperchannel';
    AF_APPLETALK: Result := 'AppleTalk';
    AF_NETBIOS: Result := 'NetBios-style addresses';
    AF_VOICEVIEW: Result := 'VoiceView';
    AF_FIREFOX: Result := 'Protocols from Firefox';
    AF_UNKNOWN1: Result := 'Somebody is using this!';
    AF_BAN: Result := 'Banyan';
    AF_ATM: Result := 'Native ATM Services';
    AF_INET6: Result := 'Internetwork Version 6';
    AF_CLUSTER: Result := 'Microsoft Wolfpack';
    AF_12844: Result := 'IEEE 1284.4 WG AF';
    AF_IRDA: Result := 'IrDA';
    AF_NETDES: Result := 'Network Designers OSI & gateway enabled protocols';
  else
    Result := 'Unknown';
  end;
end;

procedure DisplayProtocolInfo(const Info: PWSAProtocolInfo);
var
  I: Integer;
begin
  WriteLn(Info^.szProtocol);
  WriteLn('Protocol Version:     ' + IntToStr(Info^.iVersion));
  WriteLn('Address Family:       ' + AddressFamilyToString(Info^.iAddressFamily));
  WriteLn('Provider:             ' + GUIDToString(Info^.ProviderId));
  if not ExtendedInfo then Exit;
  WriteLn('Service Flags1:       ' + IntToHex(Info^.dwServiceFlags1, 8)); // TODO ToString
  WriteLn('Service Flags2:       ' + IntToHex(Info^.dwServiceFlags2, 8));
  WriteLn('Service Flags3:       ' + IntToHex(Info^.dwServiceFlags3, 8));
  WriteLn('Service Flags4:       ' + IntToHex(Info^.dwServiceFlags4, 8));
  WriteLn('Provider Flags:       ' + IntToHex(Info^.dwProviderFlags, 8));
  if Info^.dwProviderFlags and PFL_MULTIPLE_PROTO_ENTRIES <> 0 then WriteLn('
           PFL_MULTIPLE_PROTO_ENTRIES');
  if Info^.dwProviderFlags and PFL_RECOMMENDED_PROTO_ENTRY <> 0 then WriteLn('
           PFL_RECOMMENDED_PROTO_ENTRY');
  if Info^.dwProviderFlags and PFL_HIDDEN <> 0 then WriteLn('  PFL_HIDDEN');
```

```
  if Info^.dwProviderFlags and PFL_MATCHES_PROTOCOL_ZERO <> 0 then WriteLn('
           PFL_MATCHES_PROTOCOL_ZERO');
  WriteLn('Catalog Entry:       ' + IntToStr(Info^.dwCatalogEntryId));
  WriteLn('Maximum Message Size: ' + IntToHex(Info^.dwMessageSize, 8));
  WriteLn('Security Scheme:     ' + IntToStr(Info^.iSecurityScheme));
  WriteLn('Byte Order:          ' + ByteOrderToString(Info^.iNetworkByteOrder));
  WriteLn('Protocol:            ' + IntToStr(Info^.iProtocol));
  WriteLn('Protocol MaxOffset:  ' + IntToStr(Info^.iProtocolMaxOffset));
  WriteLn('Min Socket Address:  ' + IntToStr(Info^.iMinSockAddr));
  WriteLn('Max Socket Address:  ' + IntToStr(Info^.iMaxSockAddr));
  WriteLn('Socket Type:         ' + SocketTypeToString(Info^.iSocketType));
  Write('Protocol Chain:        ');
  for I := 0 to Info^.ProtocolChain.ChainLen - 1 do
           Write(IntToStr(Info^.ProtocolChain.ChainEntries[I]) + ' ');
  WriteLn;
end;

begin

  ExtendedInfo := FindCmdLineSwitch('e', ['-', '/'], True);

  if WSAStartUp($0202, WSAData) = 0 then
  try
    Assert(WSAData.wHighVersion >= 2);
    BufferLength := 0;
    if (WSAEnumProtocols(nil, nil, BufferLength) = Integer(SOCKET_ERROR)) and
       (WSAGetLastError = WSAENOBUFS) then
    begin
      Buffer := AllocMem(BufferLength);
      try
        Count := WSAEnumProtocols(nil, Buffer, BufferLength);
        if Count <> Integer(SOCKET_ERROR) then
        begin
          Info := Buffer;
          for I := 0 to Count - 1 do
          begin
            Assert(not IsBadReadPtr(Info, SizeOf(TWSAProtocolInfo)));
            DisplayProtocolInfo(Info);
            WriteLn;
            Inc(Info);
          end;
        end
        else WriteLn('Failed to retrieve protocol information.');
      finally
        FreeMem(Buffer);
      end;
    end
    else
    begin
      WriteLn('Unable to enumerate protocols.');
      WriteLn('Error code: ' + IntToStr(WSAGetLastError));
      WriteLn('Error message: ' + SysErrorMessage(WSAGetLastError));
    end;
  finally
    WSACleanUp;
  end
  else WriteLn('Windows Sockets initialization failed.');

end.
```

Table 4-1: Available properties for the dwServiceFlags1 field

Property	Meaning
XP1_CONNECTIONLESS	A protocol that provides connectionless (datagram) service. If not set, the protocol supports connection-oriented data transfer.
XP1_GUARANTEED_DELIVERY	A protocol that guarantees that all data sent will reach the intended destination
XP1_GUARANTEED_ORDER	A protocol that guarantees that data will only arrive in the order in which it was sent and that it will not be duplicated. This characteristic does not necessarily mean that the data will always be delivered, but any data that is delivered is delivered in the order in which it was sent.
XP1_MESSAGE_ORIENTED	A protocol that honors message boundaries, as opposed to a stream-oriented protocol where there is no concept of message boundaries
XP1_PSEUDO_STREAM	This is a message-oriented protocol, but message boundaries will be ignored for all receives. This is convenient when an application does not desire message framing to be done by the protocol.
XP1_GRACEFUL_CLOSE	The protocol supports two-phase (graceful) close. If not set, only abortive closes are performed.
XP1_EXPEDITED_DATA	A protocol that supports expedited (urgent) data
XP1_CONNECT_DATA	A protocol that supports connect data
XP1_DISCONNECT_DATA	A protocol that supports disconnect data
XP1_SUPPORT_BROADCAST	A protocol that supports a broadcast mechanism
XP1_SUPPORT_MULTIPOINT	A protocol that supports a multipoint or multicast mechanism. Control and data plane attributes follow immediately.
XP1_MULTIPOINT_CONTROL_PLANE	Indicates whether the control plane is rooted (value = 1) or non-rooted (value = 0)
XP1_MULTIPOINT_DATA_PLANE	Indicates whether the data plane is rooted (value = 1) or non-rooted (value = 0)
XP1_QOS_SUPPORTED	A protocol that supports quality of service requests
XP1_RESERVED	This bit is reserved.
XP1_UNI_SEND	A protocol that is unidirectional in the send direction
XP1_UNI_RECV	A protocol that is unidirectional in the recv direction
XP1_IFS_HANDLES	The socket descriptors returned by the provider are operating system Installable File System (IFS) handles.

Table 4-2: The remaining fields of the WSAPROTOCOL_INFO record

Field	Meaning
dwServiceFlags2	Reserved for additional protocol attribute definitions
dwServiceFlags3	Reserved for additional protocol attribute definitions
dwServiceFlags4	Reserved for additional protocol attribute definitions
dwProviderFlags	Provides information about how this protocol is represented in the protocol catalog
ProviderId	A globally unique identifier assigned to the provider by the service provider vendor. This value is useful for instances where more than one service provider is able to implement a particular protocol. An application may use the ProviderId value to distinguish between providers that might otherwise be indistinguishable.

Field	Meaning
ProtocolChain	A data structure representing a protocol chain consisting of one or more layered protocols on top of a base protocol
dwCatalogEntryId	A unique identifier assigned by the WinSock 2 DLL for each WSAPROTOCOL_INFO structure
iVersion	A protocol version identifier
iAddressFamily	A value to pass as the address family parameter to the socket or WSASocket function to open a socket for this protocol. This value also uniquely defines the record of protocol addresses (TSockAddr) used by the protocol.
iMaxSockAddr	The maximum address length in bytes (e.g., 16 for IP version 4. We get the value by calling the standard function, SizeOf, to compute the size of the TSockAddr data structure.)
iMinSockAddr	The minimum address length (same as iMaxSockAddr, unless protocol supports variable length addressing)
iSocketType	The value to pass as the socket type parameter to the socket function in order to open a socket for this protocol
iProtocol	The value to pass as the protocol parameter to the socket function in order to open a socket for this protocol
iProtocolMaxOffset	The maximum value that may be added to iProtocol when supplying a value for the protocol parameter to socket and WSASocket. Not all protocols allow a range of values. When this is the case, iProtocolMaxOffset will be zero.
iNetworkByteOrder	Currently these values are manifest constants (BIGENDIAN and LITTLEENDIAN) that indicate either "big endian" or "little endian" with the values 0 and 1, respectively.
iSecurityScheme	Indicates the type of security scheme employed (if any). A value of SECURITY_PROTOCOL_NONE is used for protocols that do not incorporate security provisions.
dwMessageSize	The maximum message size supported by the protocol. This is the maximum size that can be sent from any of the host's local interfaces. For protocols that do not support message framing, the actual maximum that can be sent to a given address may be less. There is no standard provision to determine the maximum inbound message size. The following special values are defined: ■ 0: The protocol is stream-oriented and hence the concept of message size is not relevant. ■ $1: The maximum outbound (send) message size is dependent on the underlying network MTU (maximum sized transmission unit) and hence cannot be known until after a socket is bound. Applications should use getsockopt to retrieve the value of SO_MAX_MSG_SIZE after the socket has been bound to a local address. ■ $FFFFFFFF: The protocol is message-oriented, but there is no maximum limit to the size of messages that may be transmitted.
dwProviderReserved	Reserved for use by service providers
szProtocol	An array of characters that contains a human-readable name identifying the protocol (for example, "SPX"). The maximum number of characters allowed is WSAPROTOCOL_LEN, which is defined to be 255.

Before we explore the topic of name space resolution and registration, we will give a formal definition of WSAEnumProtocols(), which is defined in Winsock2.pas.

function WSAEnumProtocols Winsock2.pas

Syntax

```
WSAEnumProtocols(lpiProtocols: PInt; lpProtocolBuffer:
PWSAPROTOCOL_INFO; lpdwBufferLength: PDWORD): u_int; stdcall;
```

Description

This function enumerates all available transport protocols and protocol chains installed on the local machine. You may use the *lpiProtocols* parameter as a filter to constrain the amount of information provided. Normally you set this parameter to NIL, which will cause the function to return information on all available transport protocols and protocol chains.

A TWSAProtocolInfo record is provided in the buffer pointed to by *lpProtocolBuffer* for each requested protocol. If the supplied buffer is not large enough (as indicated by the input value of *lpdwBufferLength*), the value pointed to by *lpdwBufferLength* will be updated to indicate the required buffer size. The application should then obtain a large enough buffer and call this function again.

The ordering of the TWSAProtocolInfo records that appear in the buffer coincides with the order of the protocol entries that the service provider registered with the WinSock DLL. For more detailed information on protocol chains, please refer to the WinSock 2 Service Provider Interface specification in the MSDN Library Platform SDK in Appendix C.

Parameters

lpiProtocols: An optional array of *iProtocol* values. When this parameter is NIL, information on all of the available protocols is returned. Otherwise, information is retrieved only for those protocols listed in the array.

lpProtocolBuffer: A buffer of WSAPROTOCOL_INFO records. See Tables 4-1 and 4-2 for a detailed description of the contents of the WSAPROTOCOL_INFO record.

lpdwBufferLength: On input, the size of the *lpProtocolBuffer* buffer passed to WSAEnumProtocols(). On output, the minimum buffer size required to retrieve all the requested information. The supplied buffer must be large enough to hold all entries for the routine to succeed. The number of protocols loaded on a machine is usually small.

Return Value

If the function succeeds, it will return the number of protocols. If the function fails, it will return the value of SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, WSAEINVAL, WSAENOBUFS, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

getsockopt, socket, WSASend, WSASendTo

Example

See Listing 4-3 for an example of how to use WSAEnumProtocols(). The full code for this example is in EX43.

Name Space Resolution and Registration

In Chapter 3, we learned how to use the Winsock 1.1 functions to perform resolution of hosts, protocols, and services. Winsock 2 extends this repertoire considerably with its flexible and powerful functions to determine the name spaces.

A *name space* is a collection of hosts, protocols, and services to which a computer has access. All networked machines will have at least one name space installed. You may have more than one name space on your machine. In addition, by using the functions that Winsock 2 provides, you can register a service on the server side and generate service queries on the client side.

Suppose you want to advertise a new service. You do this by registering and advertising the new service on the server. (We'll describe these steps in detail in the section "Registering a Service.") On the client side, equipped with Winsock 2, the application finds the service in a single step. Your client doesn't even have to know the port required for communicating with the service. With the old style resolution, your client application has to be very knowledgeable about the service it is trying to locate. That is, your client has to resolve the server hosting the service and then resolve the service if it has an entry in the Services file. If there is no entry for the service, which is usually the case with private services, you need to supply the port to your client, which, of course, requires prior knowledge.

Before communicating with a server, you need to enumerate the available name spaces on your workstation first. After discovering the available name spaces on your machine, you can use the appropriate name space provider to find the service you want. Every registered service has a name space associated with it, as we'll see in the next section. To perform this enumeration, we use the WSAEnumNameSpaceProviders() function to list the available name space providers that your client may have.

Enumerating Name Spaces

On a given machine, you may choose from a collection of name space models in order to resolve hosts, protocols, and services. One of these is DNS, which is the most common name space provider for TCP/IP. This is a common setup on machines equipped with Winsock 1.1. Others exist for other protocols, such as NDS (NetWare Directory Services) for Novell's IPX networks.

There are three types of name spaces: static, dynamic, and persistent. DNS is a static name space, which simply means that it cannot update its database unless the DNS server goes offline for updating. This is not very flexible. On the other hand, a dynamic name space can update on the fly. An example of a dynamic name space is SAP (Service Advertising Protocol) for Novell's IPX networks. A persistent name space, which is also dynamic, maintains registration information on disk. NDS is a persistent name space.

The WSAEnumNameSpaceProviders() function lists all available name space providers installed on the machine. The function returns an array of TWSANameSpaceInfo records. Each record contains all of the registration information for a name space provider. The TWSANameSpaceInfo record, which is defined in Winsock2.pas, is as follows:

```
_WSANAMESPACE_INFO = record
  NSProviderId: TGUID;
  dwNameSpace: DWORD;
  fActive: BOOL;
  dwVersion: DWORD;
  lpszIdentifier: LPWSTR;
end;
WSANAMESPACE_INFO = _WSANAMESPACE_INFO;
TWsaNameSpaceInfo = WSANAMESPACE_INFO;
PWsaNameSpaceInfo = PWSANAMESPACE_INFO;
```

Table 4-3 lists in detail the fields of the WSANAMESPACE_INFO.

Table 4-3: Fields of the WSANAMESPACE_INFO record

Field	Description
NSProviderId	The unique identifier for this name space provider
dwNameSpace	Specifies the name space supported by this implementation of the provider
fActive	If TRUE, indicates that this provider is active. If FALSE, the provider is inactive and is not accessible for queries, even if the query specifically references this provider.
dwVersion	Name space version identifier
lpszIdentifier	Display string for the provider

function WSAEnumNameSpaceProviders Winsock2.pas

Syntax

WSAEnumNameSpaceProviders(var lpdwBufferLength: DWORD; lpnspBuffer: LPWSANAMESPACE_INFOW): Integer; stdcall;

Description

This function retrieves information about available name spaces on the local machine.

Parameters

lpdwBufferLength: On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the API fails and the error is WSAEFAULT), the minimum number of bytes to pass for the *lpnspBuffer* to retrieve all the requested information. On input, the buffer must be large enough to hold all of the name spaces.

lpnspBuffer: On success, this is a buffer containing WSANAMESPACE_INFO records. The returned records are located consecutively at the head of the buffer. The return value of WSAEnumNameSpaceProviders() is the number of WSANAMESPACE_INFO records.

Return Value

This function will return the number of WSANAMESPACE_INFO records copied into *lpnspBuffer*. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAEFAULT, WSANOTINITIALISED, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSAGetLastError, WSAStartup

Example

Listing 4-4 (program EX44) shows how to use the WSAEnumNameSpaceProviders() function.

Listing 4-4: Using WSAEnumNameSpaceProviders()

```
{No command line parameters are required.

  Example EX44 demonstrates how to use WSAEnumNameSpaceProviders.
}
program EX44;

{$APPTYPE CONSOLE}

uses
  Windows,
  ComObj,
```

```
  WinSock2,
  SysUtils;

var
  WSAData: TWSAData;
  Buffer, Info: PWSANameSpaceInfo;
  BufferLength: DWORD;
  I, Count: Integer;

function BoolToStr(B: Boolean): string;
begin
  if B then Result := 'True' else Result := 'False';
end;

begin

  if WSAStartUp($0202, WSAData) = 0 then
  try
    Assert(WSAData.wHighVersion >= 2);
    BufferLength := 0;
    if (WSAEnumNameSpaceProviders(BufferLength, nil) = Integer(SOCKET_ERROR)) and
       (WSAGetLastError = WSAEFAULT) then
    begin
      Buffer := AllocMem(BufferLength);
      try
        Count := WSAEnumNameSpaceProviders(BufferLength, Buffer);
        if Count <> Integer(SOCKET_ERROR) then
        begin
          Info := Buffer;
          for I := 0 to Count - 1 do
          begin
            WriteLn(Info^.lpszIdentifier);
            WriteLn('Provider GUID: ' + GUIDToString(Info^.NSProviderId));
            WriteLn('Namespace:     ' + IntToStr(Info^.dwNameSpace));
            WriteLn('Active:        ' + BoolToStr(Info^.fActive));
            WriteLn('Version:       ' + IntToStr(Info^.dwVersion));
            WriteLn;
            Inc(Info);
          end;
        end
        else WriteLn('Failed to retrieve name space provider information.');
      finally
        FreeMem(Buffer);
      end;
    end
    else WriteLn('Failed to retrieve name space provider information.');
  finally
    WSACleanUp;
  end
  else WriteLn('Windows Sockets initialization failed.');

end.
```

Registering a Service

Before any potential clients can communicate with your service, you need to advertise it. This is analogous to advertising a product or service in the business world. To advertise a new service to your potential clients on the network, you need to call two functions, WSAInstallServiceClass() to install your new service class and WSASetService() to register an instance of your service. You must call these functions in that order.

The WSAInstallServiceClass() function creates a service class for the new service, associating that service class with one or more name space providers. In addition, the function defines essential properties of the new service, such as whether the service is connection oriented or connectionless. It makes the determination if the service will use a SOCK_STREAM or SOCK_DGRAM type of socket for a TCP connection or UDP connection, respectively. However, the function does not define how a client can establish a connection with the service.

The single parameter that WSAInstallServiceClass() uses is a pointer to the following data structure, which is defined in Winsock2.pas:

```
WSASERVICECLASSINFO = record
  lpServiceClassId: PGUID;
  lpszServiceClassName: LPSTR;
  dwCount: DWORD;
  lpClassInfos: LPWSANSCLASSINFOA;
end;
```

The first field, *lpServiceClassId*, is a pointer to the GUID that uniquely identifies the service class. Creating a GUID is a straightforward step in which you call a function defined in SVCGUID.PAS. For example, to create a service class for the DNS name space provider, you call the SVCID_DNS function to create the GUID. The second field, *lpszServiceClassName*, is a name of the service class. The third field, *dwCount*, is the number of WSASERVICECLASSINFO records passed in the last field, *lpClassInfos*, which is a pointer to the WSASERVICE-CLASSINFO record that defines the name spaces and protocol characteristics applicable to the service class. For example, if you want to register the service class with two name space providers, say with SAP and the Windows NT domain name spaces, then you must set dwCount to 4 because you set two attributes for each name space.

The WSANSCLASSINFO record is as follows:

```
WSANSCLASSINFO = record
  lpszName: LPWSTR;
  dwNameSpace: DWORD;
  dwValueType: DWORD;
  dwValueSize: DWORD;
  lpValue: LPVOID;
end;
```

The first field, *lpszName*, defines the attribute that the service class possesses. You should use one of the predefined values in Table 4-4 to define the attributes for the class. The second field, *dwNameSpace*, is the name space that applies to the service. The last three fields, *dwValueType*, *dwValueSize*, and *lpValue*, describe the type of data associated with the service. For example, if the value is a DWORD, *dwValueType* is set to REG_DWORD, and *dwValueSize* is the size of *lpValue*, which is a pointer to the data.

Table 4-4: Service types

String Value	Constant Define	Name Space	Description
SAPID	SERVICE_TYPE_VALUE_SAPID	NS_SAP	SAP ID
ConnectionOriented	SERVICE_TYPE_VALUE_CONN	ConnectionOriented	Any
TCPPORT	SERVICE_TYPE_VALUE_TCPPORT	NS_DNS	TCP Port
UDPPORT	SERVICE_TYPE_VALUE_UDPPORT	NS_DNS	UDP Port

After installing the new service class that describes the general properties of your service, you must call WSASetService() to register an instance of the service to make it visible on the network. This function requires three parameters: *lpqsRegInfo*, *essoperation*, and *dwControlFlags*. The first parameter is a pointer to the WSAQUERYSET data structure, which is defined in Winsock2.pas:

```
WSAQUERYSET = record
  dwSize: DWORD;
  lpszServiceInstanceName: LPWSTR;
  lpServiceClassId: PGUID;
  lpVersion: LPWSAVERSION;
  lpszComment: LPWSTR;
  dwNameSpace: DWORD;
  lpNSProviderId: PGUID;
  lpszContext: LPWSTR;
  dwNumberOfProtocols: DWORD;
  lpafpProtocols: LPAFPROTOCOLS;
  lpszQueryString: LPWSTR;
  dwNumberOfCsAddrs: DWORD;
  lpcsaBuffer: LPCSADDR_INFO;
  dwOutputFlags: DWORD;
  lpBlob: LPBLOB;
end;
```

The second parameter, *essoperation*, specifies the type of operation to take place. Table 4-5 defines the three types of operation.

Table 4-5: Types of operation for WSASetService

Operation Flag	Meaning
RNRSERVICE_REGISTER	Register the service.
RNRSERVICE_DEREGISTER	Remove the entire service from memory.
RNRSERVICE_DELETE	Remove the given instance of the service from the name space.

The final parameter, *dwControlFlags*, specifies either a value of zero or the flag SERVICE_MULTIPLE. You should use the SERVICE_MULTIPLE setting when you have a service that runs on more than one machine. For example, you could have a special service that runs on ten machines. The SERVICE_MULTIPLE value tells the WSASetService() function that the WSAQUERYSET data structure, which is pointed to by the first parameter, would have details for all ten machines providing the service. Table 4-6 enumerates possible flags that you could use with one of the operation flags in Table 4-5 to specify the service.

Table 4-6: Possible flags for WSASetService() operations

Operation	Flags	Existing Service	Non-existent Service
RNRSERVICE_REGISTER	None	Overwrite the object. Use only addresses specified. Object is REGISTERED.	Create a new object. Use only addresses specified. Object is REGISTERED.
RNRSERVICE_REGISTER	SERVICE_MULTIPLE	Update object. Add new addresses to existing set. Object is REGISTERED.	Create a new object. Use all addresses specified. Object is REGISTERED.
RNRSERVICE_DEREGISTER	None	Remove all addresses, but do not remove object from name space. Object is DEREGISTERED.	WSASERVICE_NOT_FOUND
RNRSERVICE_DEREGISTER	SERVICE_MULTIPLE	Update object. Remove only addresses that are specified. Only mark object as DEREGISTERED if no addresses are present. Do not remove from the name space.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	None	Remove object from the name space.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	SERVICE_MULTIPLE	Remove only addresses that are specified. Only remove object from the name space if no addresses remain.	WSASERVICE_NOT_FOUND

Table 4-7 lists the fields of the TWSAQuerySet data structure.

Table 4-7: The fields of the TWSAQuerySet data structure

Field Name	Description
dwSize	Must be set to the size of TWSAQuerySet data structure. This is a versioning mechanism.
lpszServiceInstanceName	Referenced string contains the service instance name
lpServiceClassId	The GUID corresponding to this service class
lpVersion	(Optional) Supplies service instance version number
lpszComment	(Optional) An optional comment string
dwNameSpace	See Table 4-8.
lpNSProviderId	See Table 4-8.
lpszContext	(Optional) Specifies the starting point of the query in a hierarchical name space

Field Name	Description
dwNumberOfProtocols	The size of the protocol constraint array in bytes. Note that this can be zero.
lpafpProtocols	Ignored
lpszQueryString	Ignored
dwNumberOfCsAddrs	The number of elements in the array of CSADDRO_INFO records referenced by lpcsaBuffer
lpcsaBuffer	A pointer to an array of CSADDRO_INFO records which contain the address(es) that the service is listening on
dwOutputFlags	Not applicable and ignored
lpBlob	(Optional) A pointer to a provider-specific entity

In Table 4-8, by combining the *dwNameSpace* and *lpNSProviderId* parameters, you could determine which name space providers to modify by this function.

Table 4-8: Different combinations of dwNameSpace and lpNSProviderID parameters

dwNameSpace	lpNSProviderId	Scope of Impact
Ignored	Non NIL	The specified name space provider
A valid name space ID	NIL	All name space providers that support the indicated name space
NS_ALL	NIL	All name space providers

The *dwNumberOfProtocols* field returns the number of supplied protocols, each of which is pointed to by the AFPROTOCOLS data structure contained in the PAFPROTOCOLS field.

The AFPROTOCOLS data structure, defined in Winsock2.pas, is:

```
AFPROTOCOLS = record
  iAddressFamily: Integer;
  iProtocol: Integer;
end;
```

The first field is the address family constant, such as AF_INET. The second field is the protocol that is supported by the selected address family, such as AF_INET. In this case, the protocol is IPPROTO_TCP.

In the WSAQUERYSET data structure, we have two important fields, *dwNumberOfCsAddrs* and *lpcsaBuffer*. The *dwNumberOfCsAddrs* is the number of CSADDR_INFO data structures, which you pass in the buffer pointed to by lpcsaBuffer.

The CSADDR_INFO data structure, which is defined in Winsock2.pas, defines the address family and the actual address at which the service is located. In the mythical case of ten machines providing a service, there would be ten instances of these data structures.

```
CSADDR_INFO = record
  LocalAddr: SOCKET_ADDRESS;
  RemoteAddr: SOCKET_ADDRESS;
  iSocketType: Integer;
  iProtocol: Integer;
end;
```

The *LocalAddr* and *RemoteAddr* fields specify the local and remote addresses, respectively. During registration, the service is bound to the address set by *LocalAddr*, and the *RemoteAddr* is the address that the client should use for the connection and exchange of data. The *iSocketType* (for example, SOCK_STREAM) and *iProtocol* (for example, PF_INET) fields specify which socket type and protocol type the client should use, respectively.

The SOCKET_ADDRESS data structure, which is defined in Winsock2.pas, is a container describing the properties of the addresses.

```
SOCKET_ADDRESS = record
  lpSockaddr: LPSOCKADDR;
  iSockaddrLength: Integer;
end;
```

Finally, registration of a service does not require the *dwOutputFlags* and *lpBlob* fields to be populated. However, you must use these fields when querying a service, which we'll cover later in this chapter.

When a service class is no longer required (for example, for an update of the service class), you call the WSARemoveServiceClass() function. This requires just one parameter, *lpServiceClassId*, which is a pointer to the GUID identifying that service class.

To complete this section, we give a formal definition of the WSAInstallServiceClass(), WSASetService(), and WSARemoveServiceClass() functions for the installation, registration, and removal of a service class, respectively.

function WSAInstallServiceClass   Winsock2.pas

Syntax

WSAInstallServiceClass(lpServiceClassInfo: LPWSASERVICECLASSINFOW): Integer; stdcall;

Description

This function registers a service class schema within a name space. The schema includes the class name, class ID, and any name space-specific information that is common to all instances of the service, such as the SAP ID or object ID.

Parameters

lpServiceClassInfo: Contains service class to name space-specific type mapping information. Multiple mappings can be handled at one time.

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAEACCES, WSAEALREADY, WSAEINVAL, WSANOTINITIALISED, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSARemoveServiceClass, WSASetService

Example

Listing 4-5 (program EX45) shows how to create, install, and advertise a service.

Listing 4-5: Using WSAInstallServiceClass() and WSASetService()

```
{

  EX45 - This example demonstrates how to create, install, and advertise a service using
  WSAInstallServiceClass and WSASetService.

No command line parameters are required.
}

program EX45;

{$APPTYPE CONSOLE}

uses
  SysUtils, Windows, WinSock2, NspApi, Common;

const
  MaxNumOfCSAddr = 10;           // advertise at most 10 addresses

var
  GSocket: TSocket;              // Socket to server
  ServiceRegInfo: WSAQUERYSET;   // QuerySet to advertise service
  EndProgram: Boolean = False;   // signal the end of program when user hits "Ctrl-C"

function CtrlHandler(dwEvent: DWORD): BOOL; stdcall;
var
  R: Integer;
begin
  Result := True;
  case dwEvent of
    CTRL_C_EVENT,
    CTRL_BREAK_EVENT,
    CTRL_LOGOFF_EVENT,
    CTRL_SHUTDOWN_EVENT,
    CTRL_CLOSE_EVENT:
      begin
        EndProgram := True;
        WriteLn('CtrlHandler: cleaning up...');
        WriteLn('delete service instance...');
        R := WSASetService(@ServiceRegInfo, RNRSERVICE_DELETE, 0);
        if R = SOCKET_ERROR then WriteLn(Format('WSASetService DELETE error %d',
               [WSAGetLastError]));
        WriteLn('Removing Service class...');
        R := WSARemoveServiceClass(ServiceGuid);
        if R = SOCKET_ERROR then WriteLn(Format('WSARemoveServiceClass error %d',
               [WSAGetLastError]));
        if GSocket <> INVALID_SOCKET then
        begin
          closesocket(GSocket);
          GSocket := INVALID_SOCKET;
```

```
        end;
      end;
  else
    Result := False;
  end;
end;

function GetNTDSAvailable(var NTDSAvailable: Boolean): Boolean;
var
  BufferLength: DWORD;
  Buffer, Name Space: PWSANameSpaceInfo;
  I, Count: Integer;
begin
  NTDSAvailable := False;
  Result := False;
  BufferLength := 0;
  if (WSAEnumNameSpaceProviders(BufferLength, nil) = SOCKET_ERROR) and (WSAGetLastError =
      WSAEFAULT) then
  begin
    Buffer := AllocMem(BufferLength);
    try
      Count := WSAEnumNameSpaceProviders(BufferLength, Buffer);
      if Count <> SOCKET_ERROR then
      begin
        Namespace := Buffer;
        for I := 0 to Count - 1 do
        begin
          if Namespace^.dwNameSpace = NS_NTDS then
          begin
            NTDSAvailable := True;
            Break;
          end;
          Inc(Namespace);
        end;
        Result := True;
      end
      else
      begin
        WriteLn('Error retrieving name space provider information.');
        WriteLn('Error: ' + SysErrorMessage(WSAGetLastError));
      end;
    finally
      FreeMem(Buffer);
    end;
  end
  else
  begin
    WriteLn('Error retrieving required buffer size for WSAEnumNamespaceProviders.');
    WriteLn('Error: ' + SysErrorMessage(WSAGetLastError));
  end;
end;

function InstallClass: BOOL;
var
  ServiceClassInfo: WSASERVICECLASSINFO;
  NameSpaceClassInfo: array [0..1] of WSANSCLASSINFO;
  Zero: DWORD;
  ServiceClassName: string;
  R: Integer;
  NtdsAvailable: Boolean;
```

4 Chapter

```
begin
  Result := False;
  Zero :=0;
  ServiceClassName := Format('TypeId %d', [ServerType]);
  WriteLn(Format('Installing ServiceClassName: %s', [ServiceClassName]));
  if GetNTDSAvailable(NtdsAvailable) and NtdsAvailable then
  begin
    // Setup Service Class info
    FillChar(ServiceClassInfo, SizeOf(ServiceClassInfo), 0);
    ServiceClassInfo.lpServiceClassId := @ServiceGuid;
    ServiceClassInfo.lpszServiceClassName := PChar(ServiceClassName);
    ServiceClassInfo.dwCount := 2;
    ServiceClassInfo.lpClassInfos := @NameSpaceClassInfo;
    FillChar(NameSpaceClassInfo, SizeOf(WSANSCLASSINFO) * 2, 0);
    WriteLn('NTDS name space class installation');
    NameSpaceClassInfo[0].lpszName := SERVICE_TYPE_VALUE_CONN;
    NameSpaceClassInfo[0].dwNameSpace := NS_NTDS;
    NameSpaceClassInfo[0].dwValueType := REG_DWORD;
    NameSpaceClassInfo[0].dwValueSize := sizeof(DWORD);
    NameSpaceClassInfo[0].lpValue := @Zero;
    NameSpaceClassInfo[1].lpszName := SERVICE_TYPE_VALUE_UDPPORT;
    NameSpaceClassInfo[1].dwNameSpace := NS_NTDS;
    NameSpaceClassInfo[1].dwValueType := REG_DWORD;
    NameSpaceClassInfo[1].dwValueSize := sizeof(DWORD);
    NameSpaceClassInfo[1].lpValue := @Zero;
    // Install the service class information
    R := WSAInstallServiceClass(@ServiceClassInfo);
    if R = SOCKET_ERROR then
    begin
      WriteLn(Format('WSAInstallServiceClass error %d', [WSAGetLastError]));
      Exit;
    end;
    Result := True;
  end;
end;
function Advertise: BOOL;
var
  R: Integer;
  NumOfCSAddr: Integer;
  SockAddresses: array [0..MaxNumOfCSAddr - 1] of SOCKADDR;
  CSAddresses: array [0..MaxNumOfCSAddr - 1] of CSADDR_INFO;
  ComputerName: string;
  Size: Cardinal;
  HostEnt: PHostEnt;
  SockAddr: SOCKADDR_IN;
  NameLength: Integer;
  AddressCount: Integer;
  pSaIn: LPSOCKADDR_IN;
  I: Integer;
begin
  Result := False; // assume failure...
  NumOfCSAddr := 0;
  // Set up the WSAQuery data
  FillChar(ServiceRegInfo, SizeOf(WSAQUERYSET), 0);
  ServiceRegInfo.dwSize := SizeOf(WSAQUERYSET);
  ServiceRegInfo.lpszServiceInstanceName := PChar(ServerName); // service instance name
  ServiceRegInfo.lpServiceClassId := @ServiceGuid; // associated service class id
  ServiceRegInfo.dwNameSpace := NS_ALL;            // advertise to all name spaces
  ServiceRegInfo.lpNSProviderId := nil;
  ServiceRegInfo.lpcsaBuffer := @CSAddresses;      // our bound socket addresses
```

```
ServiceRegInfo.lpBlob := nil;

Size := 255;
SetLength(ComputerName, Size);
GetComputerName(PChar(ComputerName), Size);
SetLength(ComputerName, StrLen(PChar(ComputerName)));
WriteLn(Format('HostName: %s', [ComputerName]));
HostEnt := gethostbyname(PChar(ComputerName));
if HostEnt = nil then Exit;

// bind to local host ip addresses and let system to assign a port number

FillChar(SockAddr, 0, SizeOf(SockAddr));
SockAddr.sin_family := AF_INET;
SockAddr.sin_addr.s_addr := htonl(INADDR_ANY);
SockAddr.sin_port := 0;

GSocket := socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if INVALID_SOCKET = GSocket then
begin
  WriteLn(Format('GSocket error %d', [WSAGetLastError]));
  Exit;
end;

R := bind(GSocket, @SockAddr, SizeOf(SockAddr));
if SOCKET_ERROR = R then
begin
  WriteLn(Format('bind error %d', [WSAGetLastError]));
  Exit;

end;

NameLength := SizeOf(SockAddr);
if getsockname(GSocket, @SockAddr, NameLength) = SOCKET_ERROR then
begin
  WriteLn(Format('getsockname error %d', [WSAGetLastError]));
  Exit;
end;

AddressCount := 0; // total number of Ip Addresses for this host
while PPCharArray(HostEnt^.h_addr_list)^[AddressCount] <> nil do Inc(AddressCount);

WriteLn('IP addresses bound...');

for I := 0 to AddressCount - 1 do
begin
  if I >= MaxNumOfCSAddr then
  begin
    WriteLn(Format('Max. number of GSocket address (%d) reached. We will not advertise
            extra ones', [MaxNumOfCSAddr]));
    Break;
  end;
  pSaIn := @SockAddresses[I];
  Move(SockAddr, pSaIn^, SizeOf(SockAddr));
  pSaIn^.sin_addr.s_addr :=  PInteger(PPCharArray(HostEnt^.h_addr_list)^[I])^;
  pSaIn^.sin_port := SockAddr.sin_port;
  WriteLn(Format('%40s', [GetSockAddrString(@SockAddresses[I])]));
  CSAddresses[I].iSocketType := SOCK_DGRAM;
  CSAddresses[I].iProtocol := IPPROTO_UDP;
  CSAddresses[i].LocalAddr.lpSockaddr := @SockAddresses[I];
```

```
    CSAddresses[I].LocalAddr.iSockaddrLength := SizeOf(SockAddr);
    CSAddresses[I].RemoteAddr.lpSockaddr := @SockAddresses[I];
    CSAddresses[I].RemoteAddr.iSockaddrLength := SizeOf(SockAddr);
    Inc(NumOfCSAddr); // increase the number SOCKADDR buffer used
  end;

  // update counters

  ServiceRegInfo.dwNumberOfCsAddrs := NumOfCSAddr;

  // Call WSASetService

  WriteLn(Format('Advertise server of instance name: %s ...', [ServerName]));
  R := WSASetService(@ServiceRegInfo, RNRSERVICE_REGISTER, 0);
  if R = SOCKET_ERROR then
  begin
    WriteLn(Format('WSASetService error %d', [WSAGetLastError]));
    Exit;
  end;

  WriteLn('Wait for client talking to me, hit Ctrl-C to terminate...');
  Result := True;
end;

function ServerRecv: BOOL;
var
  BytesReceived: Integer;
  Buffer: array [0..1023] of Char;
  PeerAddress: SOCKADDR;
  PeerAddressLength: Integer;
  R: Integer;
begin
  PeerAddressLength := SizeOf(SOCKADDR);
  BytesReceived := recvfrom(GSocket, Buffer, SizeOf(Buffer), 0, @PeerAddress,
                            PeerAddressLength);
  if BytesReceived = SOCKET_ERROR then
  begin
    R := WSAGetLastError;
    if (R <> WSAEWOULDBLOCK) and (R <> WSAEMSGSIZE) then
    begin
      WriteLn(Format('recv error: %d', [R]));
      Result := False;
      Exit;
    end;
  end
  else
  begin
    WriteLn(Format('received: [%s ', [Buffer]));
    WriteLn(Format(': %s]', [GetSockAddrString(@PeerAddress)]));
  end;
  Result := True;
end;

function DoRnrServer: BOOL;
var
  Argp: Cardinal;
begin
  // We're pessimistic, assume failure
  Result := False;
  // Install CTRL handler
```

```
  if not SetConsoleCtrlHandler(@CtrlHandler, True) then
  begin
    WriteLn(Format('SetConsoleCtrlHandler failed to install console handler: %d',
                   [GetLastError]));
    Exit;
  end;
  // Install the server class
  if not InstallClass then Exit;
  // Advertise the server instance
  if not Advertise then Exit;
  // Make our bound sockets non-blocking such that we can loop and test for data sent by
  // client without blocking.
  Argp := 1;
  if ioctlsocket(GSocket, Integer(FIONBIO), Argp) = SOCKET_ERROR then
  begin
    WriteLn(Format('ioctlsocket[%d] error %d', [0, WSAGetLastError]));
    Exit;
  end;
  // receive data from client who find our address thru Winsock 2 RnR
  while True do
  begin
    if not ServerRecv then Exit;
    if EndProgram then
    begin
      Result := True;
      Exit;
    end;
    Sleep(100);
  end;
end;

var
  StartupData: TWSAData;
  R: DWORD;
begin
  R := WSAStartup($0202, StartupData);
  if R = 0 then
  try
    GSocket := INVALID_SOCKET;
    DoRnrServer;
  finally
    SetConsoleCtrlHandler(@CtrlHandler, False);
    if WSACleanup = SOCKET_ERROR then
    begin
      WriteLn('Failed to clean-up Winsock.');
      WriteLn('Error: ' + SysErrorMessage(WSAGetLastError));
    end;
  end
  else
  begin
    WriteLn('Failed to initialize Winsock.');
    WriteLn('Error: ' + SysErrorMessage(R));
  end;
end.
```

function WSASetService   Winsock2.pas

Syntax

WSASetService(lpqsRegInfo: LPWSAQUERYSETW; essoperation: WSAESETSERVICEOP; dwControlFlags: DWORD): Integer; stdcall;

Description

This function registers or removes a service instance within one or more name spaces.

Parameters

lpqsRegInfo: Specifies service information for registration or identifies service for removal

essoperation: An enumeration value, which may be one of the values in Table 4-5

dwControlFlags: This parameter can be set to either no value or SERVICE_MULTIPLE. The function combines the value of *dwControlFlags* with the *essoperation* parameter to set the behavior of WSASetService(). Table 4-6 lists all possible operating flags.

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAEACCES, WSAEINVAL, WSANOTINITIALISED, WSA_NOT_ENOUGH_MEMORY, and WSASERVICE_NOT_FOUND.

See Appendix B for a detailed description of the error codes.

See Also

WSAInstallServiceClass, WSARemoveServiceClass

Example

See Listing 4-5 (program EX45).

function WSARemoveServiceClass   Winsock2.pas

Syntax

WSARemoveServiceClass(const lpServiceClassId: TGUID): Integer; stdcall;

Description

This function removes a service class permanently.

Parameters

lpServiceClassId: Pointer to the GUID identifying the service class for removal

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSATYPE_NOT_FOUND, WSAEACCES, WSANOTINITIALISED, WSAEINVAL, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSAInstallServiceClass, WSASetService

Example

See Listing 4-5 (program EX45).

Service Queries

We have discussed how to register and advertise a service on the server. From the client side's point of view, how does it locate such a service? To find the service, the client has to query the name space for the service. To perform this query, your client has to call three functions: WSALookupServiceBegin(), WSALookupServiceNext(), and WSALookupServiceEnd(), in that order.

A call to WSALookupServiceBegin() initiates the process by setting the parameters that define the query. The function prototype, which is defined in Winsock2.pas, is:

```
function WSALookupServiceBegin(lpqsRestrictions: LPWSAQUERYSETW;
                               dwControlFlags: DWORD;
                               var lphLookup: HANDLE): Integer; stdcall;
```

The first parameter, *lpqsRestrictions*, is a pointer to the WSAQUERYSET data structure. You should set the fields of this structure to limit the name spaces to query. (Remember that you could have more than one name space on your machine.) The second parameter, *dwControlFlags*, defines the depth of search as well as the type of data to return, which we'll examine shortly. The last parameter, *lphLookup*, is a pointer to THandle, which WSALookupServiceNext() uses for searching.

To define the depth of searching and the type of data to return, use one flag or a combination of flags from Table 4-9.

Table 4-9: Flags for queries

Flag	Description
LUP_DEEP	Query deep as opposed to just the first level
LUP_CONTAINERS	Return containers only
LUP_NOCONTAINERS	Don't return any containers
LUP_FLUSHCACHE	If the provider has been caching information, ignore the cache and go query the name space itself.
LUP_FLUSHPREVIOUS	Used as a value for the dwControlFlags argument in WSALookupService-Next. Setting this flag instructs the provider to discard the last result set, which was too large for the supplied buffer, and move on to the next result set.
LUP_NEAREST	If possible, return results in the order of distance. The measure of distance is provider specific.
LUP_RES_SERVICE	Indicates whether prime response is in the remote or local part of CSADDR_INFO record. The other part needs to be "useable" in either case.
LUP_RETURN_ALIASES	Any available alias information is to be returned in successive calls to WSALookupServiceNext, and each alias returned will have the RESULT_IS_ALIAS flag set.
LUP_RETURN_NAME	Retrieve the name as lpszServiceInstanceName
LUP_RETURN_TYPE	Retrieve the type as lpServiceClassId
LUP_RETURN_VERSION	Retrieve the version as lpVersion
LUP_RETURN_COMMENT	Retrieve the comment as lpszComment
LUP_RETURN_ADDR	Retrieve the addresses as lpcsaBuffer
LUP_RETURN_BLOB	Retrieve the private data as lpBlob
LUP_RETURN_QUERY_STRING	Retrieve unparsed remainder of the service instance name as lpszQueryString
LUP_RETURN_ALL	Retrieve all of the information

After a successful call to WSALookupServiceBegin(), the return value will be zero. Otherwise, the function will return a SOCKET_ERROR, which you should check for the cause of the error. The function returns a pointer to HANDLE, *lphLookup*, which you pass to WSALookupServiceNext(). The function prototype for WSALookupServiceNext(), which is defined in Winsock2.pas, is as follows:

```
function WSALookupServiceNext(hLookup: HANDLE;
                              dwControlFlags: DWORD;
                              var lpdwBufferLength: DWORD;
                              lpqsResults: LPWSAQUERYSETW): Integer; stdcall;
```

The handle returned by WSALookupServiceBegin() is passed into the first parameter, *hLookup*, in WSALookupServiceNext(). The second parameter, *dwControlFlags*, is similar to *dwControlFlags* in WSALookupServiceBegin(), except WSALookupServiceNext() only supports LUP_FLUSHPREVIOUS (see Table 4-9). The third parameter, *lpdwBufferLength*, is the size of the buffer passed in the final parameter, *lpqsResults*. In a query, after calling WSALookup-

ServiceBegin(), you should call WSALookupServiceNext() repetitively (inside a loop, for example) until there is no more data to be retrieved, which is indicated by the WSA_E_NO_MORE or WSAENOMORE value returned by WSALookupServiceNext(). The data returned by WSALookupServiceNext() is contained in the buffer, *lpqsResults*. To retrieve the data, you must dereference the pointer to the WSAQUERYSET data structure.

When WSALookupServiceNext() has done its searching, you must call WSALookupServiceEnd() to release any resources allocated for the query. The function prototype for WSALookupServiceEnd(), which is defined in Winsock2.pas, is as follows:

```
function WSALookupServiceEnd(hLookup: HANDLE): Integer; stdcall;
```

The parameter, *hLookup*, is the same handle that WSALookupServiceBegin() and WSALookupServiceNext() use.

We complete our coverage of these functions by giving a formal description of them.

function WSALookupServiceBegin Winsock2.pas

Syntax

WSALookupServiceBegin(lpqsRestrictions: LPWSAQUERYSETW; dwControlFlags: DWORD; var lphLookup: HANDLE): Integer; stdcall;

Description

This function initiates a client query that is constrained by the information contained within a WSAQUERYSET record. The function only returns a handle, which WSALookupServiceNext() uses to get the actual results.

As mentioned above, you use a pointer to the WSAQUERYSET record as an input parameter to WSALookupServiceBegin() to qualify the query. Table 4-10 explains how you would use the WSAQUERYSET structure to construct a query. Setting the optional fields of WSAQUERYSET to NIL will indicate to the function not to include these fields as part of its search criteria.

Table 4-10: Fields to specify the type of query

TWSAQuerySet Field Name	Query Interpretation
dwSize	Must be set to the size of WSAQUERYSET. This is a versioning mechanism.
lpszServiceInstanceName	(Optional) Referenced string contains service name. The semantics for wildcarding within the string are not defined but may be supported by certain name space providers.
lpServiceClassId	(Required) The GUID corresponding to the service class
lpVersion	(Optional) References desired version number and provides version comparison semantics (i.e., version must match exactly or version must not be less than the value supplied)
lpszComment	Ignored for queries

TWSAQuerySet Field Name	Query Interpretation
dwNameSpace	Identifier of a single name space in which to constrain the search or NS_ALL to include all name spaces. See important tip below.
lpNSProviderId	(Optional) References the GUID of a specific name space provider and limits the query to this provider only
lpszContext	(Optional) Specifies the starting point of the query in a hierarchical name space
dwNumberOfProtocols	Size of the protocol constraint array; may be zero
lpafpProtocols	(Optional) References an array of AFPROTOCOLS record. Only services that utilize these protocols will be returned.
lpszQueryString	(Optional) Some name spaces (such as whois++) support enriched SQL-like queries which are contained in a simple text string. This parameter is used to specify that string.
dwNumberOfCsAddrs	Ignored for queries
lpcsaBuffer	Ignored for queries
dwOutputFlags	Ignored for queries
lpBlob	(Optional) This is a pointer to a provider-specific entity.

TIP: In most cases, applications that require a particular transport protocol should constrain their query by address family and protocol rather than by name space. This would allow an application that wishes to locate a TCP/IP service, for example, to have its query processed by all available name spaces, such as the local hosts file, DNS, NIS, etc.

Parameters

lpqsRestrictions: Contains the search criteria. See Table 4-9 for details.

dwControlFlags: Controls the depth of the search

lphLookup: Handle to be used when calling WSALookupServiceNext() to start retrieving the results set

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAEINVAL, WSANO_DATA, WSANOTINITIALISED, WSASERVICE_NOT_FOUND, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSALookupServiceEnd, WSALookupServiceNext

Example

Listing 4-6 (program EX46) shows how to use the WSALookupServiceBegin(), WSALookupServiceNext(), and WSALookupServiceEnd() functions.

Listing 4-6: Calling WSALookupServiceBegin(), WSALookupServiceNext(), and WSALookupServiceEnd()

```
program EX46;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows,
  WinSock,
  WinSock2,
  NspApi,
  common in 'common.pas';

procedure ClientSend(AddrInfo: LPCSADDR_INFO);
var
  ComputerName, Message: string;
  Size: Cardinal;
  S: TSocket;
begin
  // set up the message text to send
  Size := 255;
  SetLength(ComputerName, Size);
  GetComputerName(PChar(ComputerName), Size);
  SetLength(ComputerName, StrLen(PChar(ComputerName)));
  Message := 'A message from the client: ' + ComputerName;
  // create the socket
  S := socket(AddrInfo^.RemoteAddr.lpSockaddr^.sa_family, AddrInfo^.iSocketType,
             AddrInfo^.iProtocol);
  if S <> INVALID_SOCKET then
  begin
    // connect, send message and close
    if connect(s, PSockAddr(AddrInfo^.RemoteAddr.lpSockaddr),
               AddrInfo^.RemoteAddr.iSockaddrLength) <> SOCKET_ERROR then
    begin
      if send(S, Message[1], Length(Message) + 1, 0) <> SOCKET_ERROR then
        WriteLn('send a message to the peer...')
      else
        WriteLn(Format('send failed %d', [WSAGetLastError]))
    end
    else WriteLn(Format('connect failed %d', [WSAGetLastError]));
    CloseSocket(S);
  end
  else WriteLn(Format('Failed socket call %d', [WSAGetLastError]));
end;

type
  TCSAddrInfoArray = array [0..1024] of CSADDR_INFO;
  PCSAddrInfoArray = ^TCSAddrInfoArray;

procedure DoRnrClient;
var
  Restrictions: WSAQUERYSET;
```

```
  Protocols: array [0..1] of AFPROTOCOLS; // = {{AF_IPX,  NSPROTO_IPX}, {AF_INET,
             IPPROTO_UDP}};
  Lookup: THandle;
  R: Integer;
  Length: DWORD;
  ResultSet: LPWSAQUERYSET;
  I: Integer;
  RemoteAddr: LPSOCKADDR;
  Buffer: Pointer;
begin
  // Set up the query restrictions. We are only interested in a specific service over a
     specific protocol.
  Protocols[0].iAddressFamily := AF_INET;
  Protocols[0].iProtocol := IPPROTO_UDP;
  ZeroMemory(@Restrictions, SizeOf(Restrictions));
  Restrictions.dwSize := SizeOf(Restrictions);
  Restrictions.lpszServiceInstanceName := PChar(ServerName);
  Restrictions.lpServiceClassId := @ServiceGuid;
  Restrictions.dwNameSpace := NS_ALL;
  Restrictions.dwNumberOfProtocols := 2;
  Restrictions.lpafpProtocols := @Protocols;
  // Execute query
  if WSALookupServiceBegin(@Restrictions, LUP_RETURN_ADDR or LUP_RETURN_NAME, Lookup) =
     SOCKET_ERROR then
  begin
    PrintError('WSALookupServiceBegin');
    Exit;
  end;
  WriteLn(Format('Performing Query for service (type, name) = (%d, %s) . . .', [ServerType,
          ServerName]));
  // Now retrieve the result. Each call to WSALookupServiceNext returns one result set. We
     use the very first
  // one and ignore all others (if any). To retrieve all result sets, just put a loop around
     the following code
  // that terminates when WSALookupServiceNext returns SOCKET_ERROR and WSAGetLastError
     returns WSA_E_NO_MORE
  Buffer := nil;
  try
    // Note that the ResultSet record is actually variable length. Therefore we allocate a
       buffer and let
    // ResultSet point to that buffer. We quess that 1024 bytes will be sufficient for most
       ResultSets
    Length := 1024;
    Buffer := AllocMem(Length);
    ResultSet := Buffer;
    R := WSALookupServiceNext(Lookup, 0, Length, ResultSet);
    if (R = SOCKET_ERROR) and (WSAGetLastError = WSAEFAULT) then
    begin
      // Our 1024 bytes wasn't enough, allocate a larger buffer and try again. This time the
         function should
      // succeed because the function told us what size the buffer has to be (through the
         Length parameter)
      ReallocMem(Buffer, Length);
      ResultSet := Buffer;
      R := WSALookupServiceNext(Lookup, 0, Length, ResultSet);
    end;
    if R = SOCKET_ERROR then
    begin
      PrintError('WSALookupServiceNext');
      WSALookupServiceEnd(Lookup);
```

```
      Exit;
    end;
    // Success. Now loop through all the transport addresses in the result set and send a
       message to each of them
    if R = 0 then
    begin
      for I := 0 to ResultSet^.dwNumberOfCsAddrs - 1 do
      begin
        RemoteAddr := PCSAddrInfoArray(ResultSet^.lpcsaBuffer)^[I].RemoteAddr.lpSockaddr;
        if RemoteAddr <> nil then
        begin
          WriteLn(Format('Name[%d]: %30s', [I, ResultSet^.lpszServiceInstanceName]));
          WriteLn(Format('%40s', [GetSockAddrString(RemoteAddr)]));
          ClientSend(@(PCSAddrInfoArray(ResultSet^.lpcsaBuffer)^[I]));
        end;
      end;
    end;
  finally
    // Release query resources and buffer
    WSALookupServiceEnd(Lookup);
    FreeMem(Buffer);
  end;
end;

var
  StartupData: TWSAData;
  R: DWORD;
begin
  R := WSAStartup($0202, StartupData);
  if R = 0 then
  try
    DoRnrClient;
  finally
    if WSACleanup = SOCKET_ERROR then
    begin
      WriteLn('Failed to clean-up Winsock.');
      WriteLn('Error: ' + SysErrorMessage(WSAGetLastError));
    end;
  end
  else
  begin
    WriteLn('Failed to initialize Winsock.');
    WriteLn('Error: ' + SysErrorMessage(R));
    Exit;
  end;
end.
```


function WSALookupServiceNext   Unit Winsock2.pas

Syntax

WSALookupServiceNext(hLookup: HANDLE; dwControlFlags: DWORD; var lpdwBufferLength: DWORD; lpqsResults: LPWSAQUERYSETW): Integer; stdcall;

Description

We call this function with the *hLookup* parameter assigned by a previous call to WSALookupServiceBegin() to retrieve the requested service information. The provider will pass back a pointer to the WSAQUERYSET record in the *lpqs-Results* buffer. The client should continue to call this function until it returns WSA_E_NO_MORE, indicating that all of the WSAQUERYSET records have been returned.

The *dwControlFlags* field specified in this function and in WSALookup-ServiceBegin() are treated as "restrictions" for the purpose of combination. The restrictions are combined between those at the invocation of WSALookup-ServiceBegin() and those at the invocation of WSALookupServiceNext(). Therefore, the flags in WSALookupServiceNext() can never increase the amount of data returned beyond what was requested in WSALookup-ServiceBegin(), although it is not an error to specify more or fewer flags. The flags specified at a given WSALookupServiceNext() apply only to that call.

The field *dwControlFlags* that is set either to LUP_FLUSHPREVIOUS or LUP_RES_SERVICE are exceptions to the "combined restrictions" rule (because they are "behavior" flags instead of "restriction" flags). If either of these flags are used in WSALookupServiceNext(), they have their defined effect regardless of the setting of the same flags in WSALookupServiceBegin().

For example, if LUP_RETURN_VERSION is specified in WSALookup-ServiceBegin(), the service provider retrieves records including the "version." If LUP_RETURN_VERSION is not specified at WSALookupServiceNext(), the returned information does not include the "version," even though it was available. No error is generated.

Also, if LUP_RETURN_BLOB is not specified in WSALookupServiceBegin() but is specified in WSALookupServiceNext(), the returned information does not include the private data. No error is generated.

Table 4-11 describes how the query results are represented in the WSAQUERYSET record.

Table 4-11: Query results in the WSAQUERYSET record

WSAQUERYSET Field Name	Result Interpretation
dwSize	Will be set to the size of the WSAQUERYSET. This is used as a versioning mechanism.
lpszServiceInstanceName	Referenced string contains service name
lpServiceClassId	The GUID corresponding to the service class
lpVersion	References version number of the particular service instance
lpszComment	Optional comment string supplied by service instance
dwNameSpace	Name space in which the service instance was found
lpNSProviderId	Identifies the specific name space provider that supplied this query result

WSAQUERYSET Field Name	Result Interpretation
lpszContext	Specifies the context point in a hierarchical name space at which the service is located
dwNumberOfProtocols	Undefined for results
lpafpProtocols	Undefined for results; all needed protocol information is in the CSADDR_INFO records.
lpszQueryString	When dwControlFlags includes LUP_RETURN_QUERY_STRING, this field returns the unparsed remainder of the lpszServiceInstanceName specified in the original query. For example, in a name space that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the lpszServiceInstanceName is fully parsed and LUP_RETURN_QUERY_STRING is used, this field is NULL or points to a zero-length string.
dwNumberOfCsAddrs	Indicates the number of elements in the array of CSADDR_INFO records
lpcsaBuffer	A pointer to an array of CSADDR_INFO records, with one complete transport address contained within each element
dwOutputFlags	RESULT_IS_ALIAS flag indicates this is an alias result.
lpBlob	(Optional) A pointer to a provider-specific entity

Parameters

hLookup: Handle returned from the previous call to WSALookupServiceBegin()

dwControlFlags: Flags to control the next operation. Currently only LUP_FLUSHPREVIOUS is defined as a means to cope with a result set that is too large. If an application does not wish to (or cannot) supply a large enough buffer, setting LUP_FLUSHPREVIOUS instructs the provider to discard the last result set, which was too large, and move on to the next set for this call.

lpdwBufferLength: On input, the number of bytes contained in the buffer pointed to by *lpqsResults*. On output, if the API fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpqsResults* to retrieve the record.

lpqsResults: A pointer to a block of memory, which will contain one result set in a WSAQUERYSET record on return

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSA_E_NO_MORE, WSA_E_CANCELLED, WSAEFAULT, WSAEINVAL, WSA_INVALID_HANDLE, WSANOTINITIALISED, WSANO_DATA, WSASERVICE_NOT_FOUND, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSALookupServiceBegin, WSALookupServiceEnd

Example

See Listing 4-6 (program EX46).

WSALookupServiceEnd Winsock2.pas

Syntax

WSALookupServiceEnd(hLookup: HANDLE): u_int; stdcall;

Description

This function frees the handle, *hLookup*, after previous calls to WSALookup-ServiceBegin() and WSALookupServiceNext().

Parameters

hLookup: Handle previously obtained by calling WSALookupServiceBegin()

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSA_INVALID_HANDLE, WSANOTINITIALISED, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSALookupServiceBegin, WSALookupServiceNext

Example

See Listing 4-6 (program EX46).

Helper Functions

We include the following functions for completeness, but we do not propose to cover these in great detail. We have already come across two helper functions early on in this chapter, WSAAddressToString() and WSAStringToAddress(). We will now look at two more functions, WSAGetServiceClassInfo() and WSAGet-ServiceClassNameByClassId().

function WSAGetServiceClassInfo Winsock2.pas

Syntax

WSAGetServiceClassInfo(const lpProviderId, lpServiceClassId: TGUID; var lpdwBufSize: DWORD; lpServiceClassInfo: LPWSASERVICECLASSINFOW): Integer; stdcall;

Description

This function retrieves all information pertaining to a specified service class from a specified name space provider.

The service class information retrieved from a particular name space provider may not necessarily be the complete set of class information that was supplied when the service class was installed. Individual name space providers are only required to retain service class information that is applicable to the name spaces that they support.

Parameters

lpProviderId: Pointer to a GUID, which identifies a specific name space provider

lpServiceClassId: Pointer to a GUID identifying the service class in question

lpdwBufSize: On input, the number of bytes contained in the buffer pointed to by *lpServiceClassInfo*. On output, if the API fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for *lpServiceClassInfo* to retrieve the record.

lpServiceClassInfo: Service class information from the indicated name space provider for the specified service class

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it returns SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAEACCES, WSAEFAULT, WSAEINVAL, WSANOTINITIALISED, WSATYPE_NOT_FOUND, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSAStartup

Example

None

function WSAGetServiceClassNameByClassId Winsock2.pas

Syntax

```
WSAGetServiceClassNameByClassId(const lpServiceClassId: TGUID;
lpszServiceClassName: LPWSTR; var lpdwBufferLength: DWORD): Integer; stdcall;
```

Description

This function returns the name of the service associated with the given type, such as the generic service name, like FTP or SMTP.

Parameters

lpServiceClassId: Pointer to the GUID for the service class

lpszServiceClassName: Service name such as FTP, SMTP, etc.

lpdwBufferLength: On input, length of buffer returned by *lpszServiceClassName*. On output, it is the length of the service name copied into *lpszServiceClassName*.

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it returns SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSAEFAULT, WSAEINVAL, WSANOTINITIALISED, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSAStartUp

Example

None

Apart from the helper functions that we just discussed (WSAGetServiceClassNameByClassId() and WSAGetServiceClassInfo()), there are other functions that help us map well-known ports, services and service classes, and name spaces to their allocated GUIDs, and vice versa. These are defined in SvcGuid.pas. The following list shows these functions. There is one sting in the tail. Remember, in Chapter 3, we stated that port numbers must be in network byte order. Well, when we use the following helper functions, we break this cardinal rule. Instead, you supply and receive port numbers in host byte order.

TIP: When using the helper functions in the following lists, you must supply and receive port numbers in host byte order.

Functions that define and test TCP and UDP GUIDs from well-known ports

```
function SVCID_TCP_RR(Port, RR: Word): TGUID;
function SVCID_TCP(Port: Word): TGUID;
function SVCID_DNS(RecordType: Word): TGUID;
function IS_SVCID_DNS(const Guid: TGUID): Boolean;
function IS_SVCID_TCP(const Guid: TGUID): Boolean;
function PORT_FROM_SVCID_TCP(const Guid: TGUID): Word;
function RR_FROM_SVCID(const Guid: TGUID): Word;
procedure SET_TCP_SVCID_RR(var Guid: TGUID; _Port, _RR: Word);
procedure SET_TCP_SVCID(var Guid: TGUID; Port: Word);
```

```
function SVCID_UDP_RR(Port, RR: Word): TGUID;
function SVCID_UDP(Port: Word): TGUID;
function IS_SVCID_UDP(const Guid: TGUID): Boolean;
function PORT_FROM_SVCID_UDP(const Guid: TGUID): WORD;
procedure SET_UDP_SVCID_RR(var Guid: TGUID; Port, RR: WORD);
procedure SET_UDP_SVCID(var Guid: TGUID; Port: WORD);
```

Functions that define and test NetWare (SAP) services based on the SAP IDs

```
function SVCID_NETWARE(SapId: WORD): TGUID;
function IS_SVCID_NETWARE(const Guid: TGUID): Boolean;
function SAPID_FROM_SVCID_NETWARE(const Guid: TGUID): WORD;
procedure SET_NETWARE_SVCID(var Guid: TGUID; SapId: WORD);
```

Functions for the Future

Perhaps the title for this section is a bit misleading, as the functions that we are about to discuss have been implemented on Windows XP, Windows 2000, and NT 4.0. However, these new functions are not supported on Windows 95 and Windows 98. These new functions came into being to support IPv6, a 128-bit version of IP, which is known to followers of the *Star Trek* genre (I count myself as one) as IPng (Internet Protocol the Next Generation). Why do we need a new version of IP? Simply put, the projection is that the Internet will run out of addresses by 2020. The design of IPv4 over the past 20 years or so has proved to be stable and effective. Unfortunately, with the explosive growth of the Internet (and it is showing no signs of abating), the address space is becoming a scarce resource. Coupled with that is the problem of maintaining huge address tables on DNS servers. After a long period of gestation, worthy of a book, IPv6 is now available on a limited basis. At present, there are islands of web servers that use IPv6.

What benefits does IPv6 have over IPv4? There are several benefits but the most important is the almost unlimited address space that 128-bit addressing provides. Superficially, IPv4 and IPv6 are similar conceptually but the underlying schema is so different that functions such as gethostbyname() don't cut the mustard with IPv6. Enter these new functions:

- getaddrinfo()
- freeaddrinfo()
- gai_strerror()
- getnameinfo()

In the case of gethostbyname(), you would use getaddrinfo() instead. The nice thing about these new functions is that they work with IPv4 and IPv6, which will enable you to support Winsock applications for IPv4 and IPv6. Not surprisingly, Microsoft calls these new functions agnostic functions. However, there are still traps for the unwary, which we will explore in the next section.

Now that we know the reasons for moving away from IPv4 to IPv6, we need to address the question, how different is IPv4 from IPv6? To answer this question, let's go back to the form of the IP address. All hosts (this is a generic term for PCs, routers, servers, clients, etc.) on the Internet use the 32-bit IP dotted address format, aaa.bbb.ccc.ddd. I do not propose to explain in great detail the taxonomy of different types of addresses, but please refer to any good TCP/IP and Windows Sockets texts (see Appendix C). Instead, I want to illustrate the difference between an IPv4 IP address and an IPv6 IP address. Because IPv4 uses 32-bit addressing, IP dotted address format is relatively straightforward to configure. Not so with IPv6 addresses, which, as you would expect with a 128-bit address scheme, are so much more complex that ordinary users are not able to configure them manually. To illustrate this complexity, any IPv4 address is always in the same format, aaa.bbb.ccc.ddd, or a 32-bit number (4 blocks times 8 bytes); an IPv6 address is a 128-bit number in the following dotted decimal format:

```
aaa.bbb.ccc.ddd.eee.fff.ggg.hhh.iii.jjj.kkk.lll.mmm.nnn.ooo.ppp
```

This represents a 128-bit address, which, you will agree, is much more complex than an IPv4 address. It is much more difficult for a user to configure, simply because it is longer. To make the IPv6 address more compact, the designers have chosen the following format in hexadecimal notation:

```
aaa.bbb.ccc.ddd.eee.fff.ggg.hhh
```

This is called colon hex notation. Like IPv4, IPv6 has name-based addresses. We will not explore this in any more detail, as this is a topic to which we will return in a future book on advanced communications.

Making Your Winsock Applications Agnostic

To make your Winsock application capable of working with both IPv4 and IPv6, you will need to follow the simple guidelines given below. For a detailed description, please refer to MSDN Platform SDK.

- Avoid using hard-coded IPv4 addresses in your application, such as 127.0.0.1 (INADDR_LOOPBACK), which is the loopback address. There is a strong argument against hard coded IP addresses in an application because the application can break if the network configuration changes; for example, a host's IP address is changed.
- Use data structures that are agnostic. That is, use SOCKADDR_ STORAGE to replace the IPv4 address structures SOCKADDR and SOCKADDRIN.

- Replace IPv4-specific functions. Use getaddrinfo() to replace gethostbyname().
- Always call Winsock 2.
- Adapt any dialogs that handle IPv4 addresses for handling IPv6 addresses, which are more complex and vary unpredictably in length. At best, because of the complex nature of IPv6 addresses, your application should not require users to configure such addresses. Indeed, it has been argued that IPv6- (and even IPv4-) based applications shouldn't require the user to enter or modify an IP address of a host, but instead rely on host names to be resolved to their IP addresses transparently by the application.

As noted, we use getaddrinfo() to perform any required resolution of hosts, services, protocols, and ports. The prototype for getaddrinfo() is defined in WS2TCPIP.pas and is as follows:

```
function getaddrinfo(nodename, servname: PChar; hints: PAddrInfo; var res: PAddrInfo):
Integer; stdcall;
```

When you use getaddrinfo(), either or both *nodename* or *servname* must point to a NULL-terminated string. The *hints* parameter is a pointer to the addrinfo structure. On success, getaddrinfo() returns a linked list of addrinfo structures in the *res* parameter.

The addrinfo structure is defined in WS2TCPIP.pas as follows:

```
LPADDRINFO = ^addrinfo;
addrinfo = record
  ai_flags: Integer;       // AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST
  ai_family: Integer;      // PF_xxx
  ai_socktype: Integer;    // SOCK_xxx
  ai_protocol: Integer;    // 0 or IPPROTO_xxx for IPv4 and IPv6
  ai_addrlen: size_t;      // Length of ai_addr
  ai_canonname: PChar;     // Canonical name for nodename
  ai_addr: PSockAddr;      // Binary address
  ai_next: LPADDRINFO;     // Next structure in linked list
end;
TAddrInfo = addrinfo;
PAddrInfo = LPADDRINFO;
```

To process the list, you need to use the pointer stored in the *ai_next* field of each returned addrinfo structure until the *ai_next* field is a NIL pointer.

The *ai_family*, *ai_socktype*, and *ai_protocol* fields of the addrinfo structure correspond to arguments in the socket function. The *ai_addr* field points to a populated socket address. The length of *ai_addr* is stored in the *ai_addrlen* field.

You can determine which type of socket to use by assigning a value to the *ai_socktype* field. For example, if your application "doesn't care" about the type of socket (for example, SOCK_STREAM, SOCK_RAW, or SOCK_DGRAM), you could specify a value of zero for *ai_socktype*. For your application to use TCP, you would assign a value of SOCK_STREAM to *ai_socktype*. The *hints* parameter is used to pass the addrinfo structure.

Before calling getaddrinfo(), there are rules that you must follow with respect to the addrinfo structure:

- A value of PF_UNSPEC for *ai_family* indicates the caller will accept any protocol family.
- A value of zero for *ai_socktype* indicates the caller will accept any socket type.
- A value of zero for *ai_protocol* indicates the caller will accept any protocol.
- *ai_addrlen* must be zero.
- *ai_canonname* must be zero.
- *ai_addr* must be NIL.
- *ai_next* must be NIL.

However, if you want your application to work only with IPv6, then you should assign PF_INET6 to *ai_family*. Occasionally, though, you might want your application to use the default values. To do this, you should set the *hints* parameter to NIL, which will enable your application to work with either IPv4 or IPv6. The other fields are set to zero.

The last field in the addrinfo structure is *ai_flags*. Flags in this field are used to determine the behavior of the getaddrinfo() function. There are three flags:

- AI_PASSIVE
- AI_CANONNAME
- AI_NUMERICHOST

If we want to use the returned socket address structure for binding (as you would if your application is a server), you set *ai_flags* to AI_PASSIVE. If the *nodename* parameter is NIL, the socket address in the addrinfo structure is set to INADDR_ANY for IPv4 and IN6ADDR_ANY_INIT for IPv6. If, on the other hand, *ai_flags* is not set to AI_PASSIVE, the returned socket address structure is ready for a call, either to the connect(), send(), or sendto() functions. Note that if nodename is NIL in this case, the socket address is set to the loopback address.

If neither AI_CANONNAME nor AI_NUMERICHOST are used (that is, *ai_flags* is zero), the getaddrinfo() function will attempt to resolve if the *nodename* parameter contains the host name. If you set *ai_flags* to AI_CANONNAME, getaddrinfo() will return the canonical name of the host in the *ai_canonname* field of the addrinfo structure on success. Beware, though, that when getaddrinfo() returns successfully using the AI_CANNONNAME flag, the *ai_canonnname* field could be set to NIL. Therefore, when your application uses the AI_CANONNAME flag it must check that *ai_canonname* is not set to NIL.

When you use the AI_NUMERICHOST flag, the *nodename* parameter must contain a host address; otherwise, the EAI_NONAME error is returned. This prevents a name resolution service from being called.

As getaddrinfo() dynamically allocates memory for the addrinfo structure, it has to be freed when your application is done with that information. Call the freeaddrinfo() function.

The getnameinfo() function provides name resolution from an address to a host name. The function prototype is defined in WS2TCPIP.pas and is as follows:

```
function getnameinfo(sa: PSockAddr; salen: socklen_t; host: PChar; hostlen: DWORD; serv:
PChar; servlen: DWORD; flags: Integer): Integer; stdcall;
```

To simplify determining buffer requirements for the *host* and *serv* parameters, the following values for maximum host name length and maximum service name are defined in the Ws2tcpip.pas header file:

```
NI_MAXHOST  =  1025;
NI_MAXSERV  =  32;
```

To modify the behavior of getnameinfo(), set the *flags* parameter to one of the following:

- **NI_NOFQDN**: Forces local hosts having only their Relative Distinguished Name (RDN) returned in the *host* parameter
- **NI_NUMERICHOST**: Returns the numeric form of the host name instead of its name. The numeric form of the host name is also returned if the host name cannot be resolved by DNS.
- **NI_NAMEREQD**: Host names that cannot be resolved by the Domain Name System (DNS) result in an error.
- **NI_NUMERICSERV**: Returns the port number of the service instead of its name
- **NI_DGRAM**: Indicates that the service is a datagram service. This flag is necessary for the few services that provide different port numbers for UDP and TCP service.

Now it's time to give a formal definition of these new functions.

function getaddrinfo   Ws2tcpip.pas

Syntax

getaddrinfo(nodename, servname: PChar; hints: PAddrInfo; var res: PAddrInfo): Integer; stdcall;

Description

This function provides protocol-independent translation from host name to address.

Parameters

nodename: A pointer to a NULL-terminated string containing a host name or a numeric host address string. The numeric host address string is a dotted decimal IPv4 address or an IPv6 hexadecimal address.

servname: A pointer to a NULL-terminated string containing either a service name or port number

hints: A pointer to an addrinfo structure that provides hints about the type of socket the caller supports

res: A pointer to a linked list of one or more addrinfo structures for the host

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return a nonzero Windows Sockets error code, as found in Appendix B. Error codes returned by getaddrinfo() map to the error codes based on IEFT recommendations. Table 4-12 shows this mapping between Windows Sockets error codes (denoted as WSA*) and their IEFT equivalents.

Table 4-12: IEFT error codes mapped to their Winsock error codes

Error	WSA* Equivalent	Description
EAI_AGAIN	WSATRY_AGAIN	Temporary failure in name resolution
EAI_BADFLAGS	WSAEINVAL	Invalid value for ai_flags
EAI_FAIL	WSANO_RECOVERY	Non-recoverable failure in name resolution
EAI_FAMILY	WSAEAFNOSUPPORT	The ai_family member is not supported.
EAI_MEMORY	WSA_NOT_ENOUGH_MEMORY	Memory allocation failure
EAI_NODATA	WSANO_DATA	No address associated with nodename
EAI_NONAME	WSAHOST_NOT_FOUND	Neither nodename nor servname provided, or not known
EAI_SERVICE	WSATYPE_NOT_FOUND	The servname parameter is not supported for ai_socktype.
EAI_SOCKTYPE	WSAESOCKTNOSUPPORT	The ai_socktype member is not supported.

Instead of calling WSAGetLastError(), you can use the gai_strerror() function to retrieve error messages based on the EAI_* codes returned by the getaddrinfo() function. However, gai_strerror() is not thread safe. Therefore, you should still continue to use WSAGetLastError().

See Appendix B for a detailed description of the error codes.

See Also

freeaddrinfo, gai_strerror

Example

Listing 4-7 (program EX47) shows how to use the getaddrinfo(), freeaddrinfo(), and getnameinfo() functions.

Listing 4-7: Calling the getaddrinfo(), freeaddrinfo(), and getnameinfo() functions with select()

```
program EX47;

{$APPTYPE CONSOLE}

uses
  Dialogs,
  SysUtils,
  Winsock2,
  WS2tcpip;

const
 DEFAULT_FAMILY     =  PF_UNSPEC;// // Accept either IPv4 or IPv6
 DEFAULT_SOCKTYPE   = SOCK_STREAM;  // TCP
 DEFAULT_PORT       = '5001';       // Arbitrary, albeit a historical test port

 BUFFER_SIZE        = 64;           // Set very small for demonstration purposes

var
  Buffer: array[0..BUFFER_SIZE - 1] of Char;
  Hostname: string;                //[NI_MAXHOST];
  Family: Integer = DEFAULT_FAMILY;
  SocketType: Integer = DEFAULT_SOCKTYPE;
  Port: string = DEFAULT_PORT;
  Address: PChar = nil;
  i, NumSocks, Res, FromLen, AmountRead: Integer;
  From: SOCKADDR_STORAGE;
  wsaData: TWSADATA;
  Hints: TADDRINFO;
  AddrInfo, AI: PAddrInfo;
  ServSock: array [0..FD_SETSIZE-1] of TSocket;
  SockSet: fd_set;
  sktConnect: TSocket;

begin
  if WSAStartup($202, wsaData) <> 0 then
  begin
    WriteLn('Call to WSAStartup failed!'); //failed to call
    Exit;
  end;

  try
    FillChar(Hints, SizeOf(Hints), 0);
    with Hints do
    begin
      ai_family := Family;
      ai_socktype := SocketType;
      ai_flags := AI_NUMERICHOST or AI_PASSIVE;
      Res := getaddrinfo(Address, PChar(Port), @Hints, AddrInfo);
      if Res = SOCKET_ERROR then
      begin
        ShowMessage('Call to getaddrinfo failed.Error ' + IntToStr(WSAGetLastError));
        Exit;
      end;
    end;
```

```
{
  By setting the AI_PASSIVE flag in the hints to getaddrinfo, we're
  indicating that we intend to use the resulting address(es) to bind
  to a socket(s) for accepting incoming connections. This means that
  when the Address parameter is NULL, getaddrinfo will return one
  entry per allowed protocol family containing the unspecified address
  for that family.


  For each address getaddrinfo returned, we create a new socket,
  bind that address to it, and create a queue to listen on.

}
    AI := AddrInfo;
    i := 0;

    while AI <> nil do
    begin
      if i = FD_SETSIZE then
      begin
        ShowMessage('getaddrinfo returned more addresses than we could use');
        Break;
      end;

      if (AI^.ai_family <> PF_INET) and (AI^.ai_family <> PF_INET6) then
      begin
        AI := AddrInfo^.ai_next;
        Inc(i);
        Continue;
      end;

      // Open a socket with the correct address family for this address.

      ServSock[i] := socket(AI^.ai_family, AI^.ai_socktype, AI^.ai_protocol);
      if ServSock[i] = INVALID_SOCKET then
      begin
        WriteLn(Format('Call to socket() failed with error %d',[WSAGetLastError]));
        AI := AddrInfo^.ai_next;
        Inc(i);
        Continue;
      end;
{
      bind() associates a local address and port combination
      with the socket just created. This is most useful when
      the application is a server that has a well-known port
      that clients know about in advance.
}
      if bind(ServSock[i], AI^.ai_addr, AI^.ai_addrlen) = SOCKET_ERROR then
      begin
        WriteLn(Format('Call to bind() failed with error %d', [WSAGetLastError]));
        AI := AddrInfo^.ai_next;
        Inc(i);
        Continue;
      end;
{
      So far, everything we did was applicable to TCP as well as UDP.
      However, there are certain fundamental differences between stream
      protocols, such as TCP, and datagram protocols, such as UDP.

      Only connection-orientated sockets, for example those of type
```

```
      SOCK_STREAM, can listen() for incoming connections.
}
      if SocketType = SOCK_STREAM then
      begin
        if listen(ServSock[i], 5) = SOCKET_ERROR then
        begin
          WriteLn(Format('Call to listen() failed with error %d', [WSAGetLastError]));
          AI := AddrInfo^.ai_next;
          Inc(i);
          Continue;
        end;
      end;

      WriteLn(Format('Listening on port %s, protocol %d, protocol family %d', [Port,
              SocketType, AI^.ai_family]));

      AI := AI^.ai_next;
      Inc(i);
    end;

    freeaddrinfo(AddrInfo);

    if i = 0 then
    begin
      WriteLn('Fatal error: unable to serve on any address.');
      WSACleanup;
      Halt;
    end;

    NumSocks := i;

{
    We now put the server into an eternal loop,
    serving requests as they arrive.
}

    FD_ZERO(SockSet);

    while TRUE do
    begin
      FromLen := SizeOf(From);

{
      Check to see if we have any sockets remaining to be served
      from previous time through this loop. If not, call select()
      to wait for a connection request or a datagram to arrive.
}

      for i := 0 to NumSocks - 1 do
      begin
        if FD_ISSET(ServSock[i], SockSet) then break;
      end;

      if i = NumSocks then
      begin
        for i := 0 to NumSocks - 1 do _FD_SET(ServSock[i], SockSet);
        if select(NumSocks, @SockSet, nil, nil, nil) = SOCKET_ERROR then
        begin
          WriteLn(Format('Call to select() failed with error %d', [WSAGetLastError]));
          WSACleanup;
```

```
          Halt;
        end;
      end;

      for i := 0 to NumSocks - 1 do
      begin
        if FD_ISSET(ServSock[i], SockSet) then
        begin
          FD_CLR(ServSock[i], SockSet);
          Break;
        end;
      end;

      if SocketType = SOCK_STREAM then
      begin
{
        Since this socket was returned by the select(), we know we
        have a connection waiting and that this accept() won't block.
}

        sktConnect := accept(ServSock[i], @From, @FromLen);
        if sktConnect = INVALID_SOCKET then
        begin
          WriteLn(Format('Call to accept() failed with error %d',[WSAGetLastError]));
          WSACleanup;
          Halt;
        end;

        SetLength(HostName, NI_MAXHOST);
        if getnameinfo(@From, FromLen, PChar(HostName), NI_MAXHOST, nil, 0, NI_NUMERICHOST)
                       <> 0 then
          HostName := '<unknown>'
        else
          SetLength(HostName, StrLen(PChar(HostName)));
        WriteLn(Format('Accepted connection from %s', [HostName]));

{
        This sample server only handles connections sequentially.
        To handle multiple connections simultaneously, a server
        would likely want to launch another thread or process at this
        point to handle each individual connection. Alternatively,
        it could keep a socket per connection and use select()
        on the fd_set to determine which to read from next.

        Here we just loop until this connection terminates.
}

        while True do
        begin
{
          We now read in data from the client. Because TCP
          does NOT maintain message boundaries, we may recv()
          the client's data grouped differently than it was
          sent. Since all this server does is echo the data it
          receives back to the client, we don't need to concern
          ourselves about message boundaries. But it does mean
          that the message data we print for a particular recv()
          below may contain more or less data than was contained
          in a particular client send().
}
```

```
          AmountRead := recv(sktConnect, Buffer, sizeof(Buffer), 0);
          if AmountRead = SOCKET_ERROR then
          begin
            WriteLn(Format('Call to recv() failed with error %d', [WSAGetLastError]));
            closesocket(sktConnect);
            Break;
          end;
          if AmountRead = 0 then
          begin
            WriteLn('Client closed connection...');
            closesocket(sktConnect);
            Break;
          end;
          WriteLn(Format('Received %d bytes from client: %s', [AmountRead, Buffer]));
          WriteLn('Echoing same data back to client...');
          Res := send(sktConnect, Buffer, AmountRead, 0);
          if Res = SOCKET_ERROR then
          begin
            WriteLn(Format('Call to send() failed with error %d', [WSAGetLastError]));
            closesocket(sktConnect);
            Break;
          end;
        end
      end
      else
      begin
{
        Since UDP maintains message boundaries, the amount of data
        we get from a recvfrom() should match exactly the amount of
        data the client sent in the corresponding sendto().
}
        AmountRead := recvfrom(ServSock[i], Buffer, sizeof(Buffer), 0, @From, FromLen);
        if AmountRead = SOCKET_ERROR then
        begin
          WriteLn(Format('Call to recvfrom() failed with error %d',[WSAGetLastError]));
          closesocket(ServSock[i]);
          Break;
        end;
        if AmountRead = 0 then
        begin
{           This should never happen on an unconnected socket, but...}
          WriteLn('recvfrom() returned zero, aborting...');
          closesocket(ServSock[i]);
          Break;
        end;
        Res := getnameinfo(@From, FromLen, PChar(HostName), SizeOf(HostName), nil, 0,
                           NI_NUMERICHOST);
        if Res <> 0 then
        begin
          WriteLn(Format('Call to getnameinfo() failed with error %d', [Res]));
          StrPCopy(PChar(Hostname), '<unknown>');
        end;
        WriteLn(Format('Received a %d byte datagram from %s', [AmountRead, HostName]));
        WriteLn('Echoing same data back to client...');
        Res := sendto(ServSock[i], Buffer, AmountRead, 0, @From, FromLen);
        if Res = SOCKET_ERROR then
        begin
          WriteLn(Format('Call to send() failed with error %d', [WSAGetLastError]));
        end;
```

```
      end;

    end;
  finally
    WSACleanup;
  end;

end.
```

procedure freeaddrinfo Ws2tcpip.pas

Syntax

freeaddrinfo(ai: PAddrInfo); stdcall;

Description

This function frees address information that getaddrinfo() dynamically allocates in its addrinfo data structures.

Parameters

ai: A pointer to the addrinfo structure or linked list of addrinfo structures to be freed. All dynamic storage pointed to within the addrinfo structure(s) is also freed.

The freeaddrinfo() function frees the initial addrinfo structure pointed to in its *ai* parameter, including any buffers to which its members point, and then continues freeing any addrinfo structures linked by its *ai_next* member. The freeaddrinfo() function continues freeing linked structures until *ai_next* is NULL.

Return Value

This procedure doesn't return any error codes.

See Also

gai_strerror, getaddrinfo

Example

See Listings 4-7 and 4-8 (EX47 and EX48).

function getnameinfo Ws2tcpip.pas

Syntax

getnameinfo(sa: PSockAddr; salen: socklen_t; host: PChar; hostlen: DWORD; serv: PChar; servlen: DWORD; flags: Integer): Integer; stdcall;

Description

The function provides name resolution from an address to a host name.

Parameters

sa: A pointer to a socket address structure containing the address and port number of the socket. For IPv4, the *sa* parameter points to a sockaddr_in structure; for IPv6, the *sa* parameter points to a sockaddr_in6 structure.

salen: The length of the structure pointed to in the *sa* parameter

host: A pointer to the host name. The host name is returned as a fully qualified domain name (FQDN) by default.

hostlen: The length of the buffer pointed to by the *host* parameter. The caller must provide a buffer large enough to hold the host name, including terminating NULL characters. A value of zero indicates the caller does not want to receive the string provided in *host*.

serv: A pointer to the service name associated with the port number

servlen: The length of the buffer pointed to by the *serv* parameter. The caller must provide a buffer large enough to hold the service name, including terminating NULL characters. A value of zero indicates the caller does not want to receive the string provided in *serv*.

flags: Used to customize processing of the getaddrinfo() function

Return Value

On success, the function will return zero. Otherwise, any nonzero value will indicate failure. Use the WSAGetLastError() function to retrieve error information.

See Also

getaddrinfo

Example

See Listings 4-7 and 4-8 (programs EX47 and EX48).

Listing 4-8: Calling the getaddrinfo() and getnameinfo() functions

```
program EX48;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Winsock2,
  WS2tcpip;

{
  This code assumes that at the transport level, the system only supports
  one stream protocol (TCP) and one datagram protocol (UDP). Therefore,
  specifying a socket type of SOCK_STREAM is equivalent to specifying TCP
  and specifying a socket type of SOCK_DGRAM is equivalent to specifying UDP.
}
```

```
const
  DEFAULT_SERVER   = nil; // Will use the loopback interface
  DEFAULT_FAMILY   = PF_UNSPEC;// Accept either IPv4 or IPv6
  DEFAULT_SOCKTYPE = SOCK_STREAM; // TCP
  DEFAULT_PORT     = '5001';// Arbitrary, albeit a historical test port
  DEFAULT_EXTRA    = 0; // Number of "extra" bytes to send
  BUFFER_SIZE      = 65536;

type
  TCharArray = array [0..BUFFER_SIZE - 1] of Char;

function ReceiveAndPrint(sktConn: TSocket; var Buffer: TCharArray; BufLen: Integer): Integer;
var
  AmountRead: Integer;
begin
  AmountRead := recv(sktConn, Buffer, BufLen, 0);
  if AmountRead = SOCKET_ERROR then
  begin
    WriteLn(Format('Call to recv() failed with error %d', [WSAGetLastError]));
    closesocket(sktConn);
    WSACleanup;
    Halt;
  end;

{
  We are not likely to see this with UDP, since there is no 'connection' established.
}

  if AmountRead = 0 then
  begin
    WriteLn('Server closed connection...');
    closesocket(sktConn);
    WSACleanup;
    Halt;
  end;
  WriteLn(Format('Received %d bytes from server: %s',[AmountRead, Buffer]));
  Result := AmountRead;
end;

var
  Buffer: TCharArray;
  AddrName: array [0..NI_MAXHOST - 1] of Char;
  Server: PChar = DEFAULT_SERVER;
  Family: Integer = DEFAULT_FAMILY;
  SocketType : Integer = DEFAULT_SOCKTYPE;
  Port: string = DEFAULT_PORT;
  i, Res, AddrLen, AmountToSend: Integer;
  ExtraBytes: Integer = DEFAULT_EXTRA;
  Iteration: Byte = 0;
  MaxIterations: Byte = 1;
  RunForever: Boolean = FALSE;
  wsaData: TWSADATA;
  Hints: TAddrInfo;
  AddrInfo, AI: PAddrInfo;
  sktConn: TSocket;
  Addr: SOCKADDR_STORAGE;

begin
  if WSAStartup($0202,wsaData) <> 0 then
  begin
```

```
    WriteLn('Call to WSAStartup() failed...');
    Exit;
  end;

  try
{
    By not setting the AI_PASSIVE flag in the hints to getaddrinfo, we're
    indicating that we intend to use the resulting address(es) to connect
    to a service. This means that when the Server parameter is NULL,
    getaddrinfo will return one entry per allowed protocol family
    containing the loopback address for that family.
}

    FillChar(Hints, SizeOf(Hints), 0);
    Hints.ai_family := Family;
    Hints.ai_socktype := SocketType;
    Res := getaddrinfo(Server, PChar(Port), @Hints, AddrInfo);
    if Res <> 0 then
    begin
      WriteLn(Format('Call to getaddrinfo() failed with error %d. Unable to resolve address
                 [%s] and port [%s]', [gai_strerror(Res), Server, Port]));
      WSACleanup;
      Halt;
    end;

{
    Try each address getaddrinfo returned, until we find one to which
    we can successfully connect.
}

    AI := AddrInfo;
    i := 0;
    while AI <> NIL do
    begin
{     Open a socket with the correct address family for this address. }
      sktConn := socket(AI^.ai_family, AI^.ai_socktype, AI^.ai_protocol);
      if sktConn = INVALID_SOCKET then
      begin
        WriteLn(Format('Call to socket() failed with error %d',[WSAGetLastError]));
        ai := ai^.ai_next;
        inc(i);
        Continue;
      end;

{
      Notice that nothing in this code is specific to whether we
      are using UDP or TCP.

      When connect() is called on a datagram socket, it does not
      actually establish the connection as a stream (TCP) socket
      would. Instead, TCP/IP establishes the remote half of the
      (LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
      This enables us to use send() and recv() on datagram sockets,
      instead of recvfrom() and sendto().
}

      if Server <> nil then
        WriteLn(Format('Attempting to connect to: %s', [Server]))
      else
        WriteLn('Attempting to connect');
```

```
      if connect(sktConn, AI^.ai_addr, AI^.ai_addrlen) <> SOCKET_ERROR then
        Break;
      i := WSAGetLastError;
      if getnameinfo(AI^.ai_addr, AI^.ai_addrlen, AddrName, SizeOf(AddrName), nil, 0,
                     NI_NUMERICHOST) <> 0 then
        StrPCopy(AddrName, '<unknown>');
      WriteLn(Format('Call to connect() to %s failed with error %d', [AddrName, i]));
      closesocket(sktConn);
      ai := ai^.ai_next;
      Inc(i);
    end;

    if AI = nil then
    begin
      WriteLn('Fatal error: unable to connect to the server...');
      WSACleanup;
      Halt;
    end;

{
    This demonstrates how to determine to where a socket is connected.
}

    AddrLen := sizeof(Addr);
    if getpeername(sktConn, @Addr, AddrLen) = SOCKET_ERROR then
    begin
      WriteLn(Format('Call to getpeername() failed with error %d', [WSAGetLastError]));
    end
    else
    begin
      if getnameinfo(@Addr, AddrLen, AddrName, SizeOf(AddrName), nil, 0, NI_NUMERICHOST) <> 0
                     then
        StrPCopy(AddrName, '<unknown>');
      WriteLn(Format('Connected to %s, port %u, protocol %u, protocol family %u',
        [AddrName, ntohs(SS_PORT(@Addr)), AI^.ai_socktype, AI^.ai_family]));
    end;

{   We are done with the address info chain, so we can free it. }

    freeaddrinfo(AddrInfo);

{
    Find out what local address and port the system picked for us.
}

    AddrLen := SizeOf(Addr);
    if getsockname(sktConn, @Addr, AddrLen) = SOCKET_ERROR then
    begin
      WriteLn(Format('Call to getsockname() failed with error %d',[WSAGetLastError]));
    end else
    begin
      if getnameinfo(@Addr, AddrLen, AddrName, SizeOf(AddrName), NIL, 0, NI_NUMERICHOST) <> 0
                     then
        StrPCopy(AddrName, '<unknown>');
      WriteLn(Format('Using local address %s, port %d',[AddrName, ntohs(SS_PORT(@Addr))]));
    end;
```

```
{
    Send and receive in a loop for the requested number of iterations.
}

    while RunForever or (Iteration < MaxIterations) do
    begin
{      Compose a message to send. }

      StrPCopy(Buffer, 'Message #' + IntToStr(Iteration + 1));
      AmountToSend := Length('Message #' + IntToStr(Iteration + 1));

{     Send the message. Since we are using a blocking socket, this
      call shouldn't return until it's able to send the entire amount.
}
      Res := send(sktConn, Buffer, AmountToSend, 0);
      if Res = SOCKET_ERROR then
      begin
        WriteLn(Format('Call to send() failed with error %d',[WSAGetLastError]));
        WSACleanup;
        Halt;
      end;

      WriteLn(Format('Sent %d bytes (out of %d bytes) of data',[Res, AmountToSend]));

{     Clear buffer just to prove we're really receiving something. }

      FillChar(Buffer, sizeof(Buffer), #0);

{     Receive and print server's reply. }

      ReceiveAndPrint(sktConn, Buffer, sizeof(Buffer));
      Inc(Iteration);
    end;// while RunForever

{   Tell system we're done sending. }

    WriteLn('Done sending...');
    shutdown(sktConn, SD_SEND);

{
    Since TCP does not preserve message boundaries, there may still
    be more data arriving from the server. So we continue to receive
    data until the server closes the connection.
}

    if SocketType = SOCK_STREAM then
      while ReceiveAndPrint(sktConn, Buffer, sizeof(Buffer)) <> 0 do ;

    closesocket(sktConn);

  finally
    WSACleanup;
  end;

end.
```

function gai_strerror Ws2tcpip.pas

Syntax

gai_strerror(ecode: Integer): PChar;

Description

This function retrieves error messages based on the EAI_* errors returned by the getaddrinfo() function. Note that the gai_strerror() function is not thread safe, and therefore, you should use WSAGetLastError() instead.

If the *ecode* parameter is not an error code value that getaddrinfo() returns, the gai_strerror() function returns a pointer to a string that indicates an unknown error.

Parameters

ecode: Error code from the list of available getaddrinfo() error codes. For a complete listing of these error codes, see Table 4-12.

See Also

WSAGetLastError

Example

See Listing 4-7 (program EX47).

Obsolete Functions

Other functions that Winsock 1.1 developers use as part of their repertoire of resolution tools are now obsolete and no longer supported. Although these functions are retained for backward compatibility, you shouldn't be tempted to use them; instead, use the functions that we explored in this chapter. Learn to use these new functions in your Winsock applications and you will reap the dividends of ease of use for your applications. The following obsolete functions should be avoided:

- GetAddressByName()
- EnumProtocols()
- GetNameByType()
- GetService()
- GetTypeByName()
- SetService()

These Microsoft-specific functions are defined in NspAPI.pas.

For more information on these obsolete functions, please refer to the MSDN Library Platform SDK (see Appendix C).

Summary

In this chapter, we showed you how to use the Winsock 2 resolution and registration functions. Equipped with the knowledge gained from this and preceding chapters, we are ready to explore the world of peer-to-peer communications.

Chapter 5

Communications

In the last two chapters, we covered the resolution issues that an application must address before communication with Winsock can begin. In this chapter, we will come to grips with the communications process itself. As this is a huge subject to cover, this chapter will be the longest by far on Winsock 2. However, to make our voyage of discovery in this chapter easier to handle, we will examine the subject topic by topic, as follows:

- Socket creation
- Making the connection
- Data exchange
- Breaking the connection
- I/O schemes
- Raw sockets
- Microsoft extensions
- Microsoft extensions to Winsock 2 for Windows XP and Windows .NET Server
- IP Multicast
- Obsolete functions

Unlike Winsock 1.1 applications, which use the TCP/IP protocol suite to communicate almost exclusively, Winsock 2 applications can select an appropriate protocol from a pool of available protocols. This is a powerful and flexible feature. For example, a server application could select a protocol, such as IPX, in response to a client using that same protocol and simultaneously servicing other clients that are using TCP/IP. The design of Winsock 2 permits the addition of new protocols as they become available. One such protocol, IrDA, is a relatively recent addition to Winsock that allows it to be used also for IR (infrared sockets) communication. In theory, Winsock 1.1 was designed to use other protocols such as IPX/SPX in addition to TCP/IP; however, it was never used with other protocols in the real world.

For space reasons, we will focus exclusively on the TCP/IP protocol suite, which, in any case, is the most common set of protocols for communication on the Internet and intranets. However, with the exception of socket creation, which is protocol dependent, the principles that we will learn here for TCP/IP also apply to other protocols such as IPX/SPX, etc.

The Mechanics of Data Exchange

Before examining these topics, we provide an overview of how data exchange operates in practice. In general, in any Winsock conversation between the client and server, the client application must initiate the connection by performing these basic steps:

- Call WSAStartUp() to initialize Winsock (Chapter 2).
- If a host name is used, then resolve the target host's Internet address. Otherwise, skip this step (Chapters 3 and 4).
- Create a socket using socket() or WSASocket().
- Use the connect() or WSAConnect() function to link the client with the server. Note that client applications using UDP do not require this step.
- Send and receive data until done.
- Close the socket by calling shutdown() and closesocket().
- Call WSACleanup() to free resources allocated by the application.

Depending on the type of application and the protocol used, the steps described above can vary considerably. For example, an FTP client creates at least two sockets: one socket for commands to send over the control channel and one or more sockets for data transmission. (In FTP, a socket is created whenever data is required, such as directory listings, file transfers, etc. When the data transfer is complete, the socket is closed.)

Although things are deceptively simpler on the server side, a server application must perform the following basic steps:

- Call WSAStartUp() to initialize Winsock.
- Create a socket using socket() or WSASocket().
- Call bind() to associate the socket with the local address, address family, and port.
- Call listen() to listen for a connection on the designated port.
- On connection, call either accept() or WSAAccept() to accept the connection request and create a new socket for the connection. After accepting the connection, the server continues to listen for new connections.

- Exchange data with the connected client until complete.
- On shutdown, call WSACleanup() to free resources.

As we can see, the steps that we have itemized above are for a server that services many clients at a time. The steps above do not show, however, the implementation of an I/O scheme that makes it possible for a server to serve more than one client. We will cover such schemes in this chapter.

Before you can establish a communication link with another machine, you need to create a socket first. But before we explore the process of creating a socket, we must answer the question, "what is a socket?" A *socket* is an abstract entity that describes an endpoint of the communication link. In terms of functionality, a socket is like an electrical socket through which an electrical current can pass. Using the electrical socket analogy, the current is the data that flows from one socket to another across the circuit. So, when the socket is closed, no data can enter the socket. Having defined what a socket is, we can now discuss the creation of sockets.

Socket Creation

To create a socket, you may use one of two functions: socket() or WSASocket(). We learned from Chapter 4 that we need to select the appropriate address family and transport protocol in order to use the service that is available, such as FTP, SMTP, and other well-known protocols. For applications that use Winsock 1.1, the address family is usually AF_INET for the Internet. In addition to the AF_INET address family, Winsock 2 provides additional address families, such as AF_ATM and AF_IPX. In Chapter 4, we introduced different transport protocols that require different address families. For example, you use the AF_ATM address family for the ATM transport protocol.

The Transmission Control Protocol (TCP) sits on top of the IP's datagram service, thus providing *reliability* and *flow control*. TCP provides a virtual circuit between the client and server, one that provides a reliable means of exchanging *data streams* across a virtual circuit between server and client, and vice versa. Why are we belaboring this point? It is a common misconception among neophyte network programmers and even some who are more experienced that data is transmitted in packets. That is not the case with TCP. So, a data stream is simply that. For example, when a server sends data to a client, the server sends a continuous stream of bytes without any boundaries. That is, TCP doesn't care in what format the data is being transmitted; to TCP, the data is just a stream of bytes. (We saw in Chapter 1 that the TCP protocol sits on top of the IP layer, which is the layer that actually transmits data as packets. To all intents and purposes, though, TCP sees the data as byte streams.) Hence, the allegory of using an electrical socket becomes very clear; like an electrical current, the data stream is simply a continuous stream of bytes that make up the data.

The fact that there are no boundaries to demarcate the start and end of different sets of data is important. That is, any application using TCP has to send and receive until there is no more data, and it is up to the application to handle the data that it receives correctly. For example, let's take the SMTP protocol; the smtp server receives and forwards e-mail messages in the correct format required for smtp, but as far as the TCP protocol is concerned, the data is transmitted as a stream of bytes. We will come back to this topic of how TCP handles the data when we discuss the send(), recv(), WSASend(), and WSARecv() functions later in this chapter. The disadvantage of using TCP is its considerable overhead, but it has the advantage of guaranteeing reliable delivery of data. This apparent weakness is usually of little significance to the majority of network applications. The protocols that use TCP are FTP, SMTP, POP3, NNTP, and HTTP. A typical server (for example, an FTP server) usually handles hundreds of clients with each client being connected via a virtual circuit.

Up to this point, we have been saying that TCP is "reliable," but we do not mean that TCP is infallible, which is a different matter. Let's demonstrate what we mean by this subtle distinction with a simple scenario, which is one that is likely to happen when data is exchanged across the network, notably the Internet. For example, take a server that uses the FTP protocol; TCP guarantees the reliable delivery of data leaving the server and reliable reception of the data by the client, but it does not guarantee that the data, which is encapsulated as IP datagrams, will be transferred flawlessly over numerous routers between the server and client. A router could fail, thus breaking the virtual circuit to send the data into a cyber hole.

TIP: Although TCP is reliable, it is not infallible.

In contrast, the User Datagram Protocol (UDP) is a much simpler (some might say "primitive") protocol than TCP in that it adds only a checksum facility to the basic IP datagram service. Hence, as UDP does not provide flow control, it provides a one-shot connection, or connectionless transport, to transmit the data. Because there is no flow control, this protocol does not guarantee reliable delivery of data at all. However, unlike TCP, it is capable of exchanging data between multiple sources. As there is little overhead, a UDP client sends data immediately. The server and recipient, however, do not send acknowledgments of receipt of data. Because UDP has this property of transmitting data to multiple recipients, IP Multicast uses UDP as its transport protocol.

When you design a network application, you must ask yourself several questions, one of which is "Which protocol should I use, TCP or UDP?" There is a basic rule to follow: If the data is required to be sent reliably, you must use TCP. If not, you can use UDP to send "messages" or "heartbeats" between server and client. If you wish to send data to more than one client, you would use UDP, which is the protocol that IP Multicast uses. However, this simple rule falls away if you wish to use Reliable IP Multicast to send data reliably to hundreds or even thousands of clients.

After selecting a transport protocol and compatible address family, you then create the socket. There are two functions, socket() and WSASocket(), to create a socket. We will consider the socket() function first, which is the simpler of the two. The prototype for socket() is:

```
function socket(af, type_, protocol: Integer): TSocket; stdcall;
```

The function creates a socket that is a combination of the address family, socket type, and protocol parameters (which are the *af*, *type_*, and *protocol* parameters, respectively). Every socket that you create will always have the overlapped attribute set by default. What do we mean by an overlapped socket? An overlapped socket is simply an asynchronous socket. We will come back to this later in this chapter. If you want to create a socket without the overlapped attribute, you should call WSASocket() instead. You should use overlapped sockets in an overlapped I/O scheme, which we will also cover later in this chapter.

Before learning about WSASocket(), let's touch upon the socket types that Winsock provides. Currently, Winsock supports five socket types, SOCK_STREAM, SOCK_DGRAM, SOCK_RDM, SOCK_SEQPACKET, and SOCK_RAW. According to the Winsock 2 specification, SOCK_RAW is an optional socket type. Table 5-1 shows the different types of sockets. For the moment, we will focus on the SOCK_STREAM and SOCK_DGRAM socket types, but we will discuss the SOCK_RAW socket type later in this chapter. If you wish to send data reliably, you should use SOCK_STREAM for TCP. Otherwise, for transmission of messages or heartbeats, you should use SOCK_DGRAM for UDP. (One such possible application is the synchronization of computer clocks on the network.) For ICMP, such as that used by the ping and traceroute type applications, you should use the SOCK_RAW socket type.

Table 5-1: Socket types supported by Winsock 2

Type	Description
SOCK_STREAM	Provides sequenced, reliable, two-way, connection-based byte streams with an out-of-band data transmission mechanism. This type uses TCP for the Internet address family.
SOCK_DGRAM	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. This type uses UDP for the Internet address family.
SOCK_RAW	Uses datagrams.
SOCK_SEQPACKET	DECnet sockets use sequenced packets that maintain message boundaries across the network.
SOCK_RDM	Provides reliably delivered messages. That is, message boundaries in data are preserved.

The second function, WSASocket(), creates a non-overlapped socket by default. Unlike socket(), you can specify whether the socket is to be in overlapped or non-overlapped mode. In certain situations, using an overlapped socket can speed up data exchange considerably, which we will discuss under the "I/O Schemes" section of this chapter.

The prototype for WSASocket() is:

```
function WSASocket(af, type_, protocol: Integer; lpProtocolInfo: LPWSAPROTOCOL_INFOW;
  g: GROUP; dwFlags: DWORD): TSocket; stdcall;
```

Looking at the prototype, you can see that WSASocket() is considerably more complex than the humble socket() function. The first three parameters are the same as in socket(). The *lpProtocolInfo* parameter is a pointer to the WSAPROTOCOL_INFO record, which defines the transport protocol for the socket. When *lpProtocolInfo* is set to NIL, Winsock uses the first three parameters for the address family, socket type, and protocol to define the socket. The next parameter, *g*, is for the concept of socket groups that was introduced in earlier Winsock 2 specifications but not used in the present incarnation of Winsock 2.

If you wish to use overlapped I/O, you need an overlapped socket. To obtain such a socket, you should set the *dwFlags* parameter to the WSA_FLAG_OVERLAPPED constant. This constant is in Table 5-2. The other constants, such as WSA_FLAG_MULTIPOINT_C_ROOT, are for use with multicast applications.

Table 5-2: Flags to determine socket behavior

Flag	Description
WSA_FLAG_OVERLAPPED	This flag creates an overlapped socket. Overlapped sockets may use WSASend(), WSASendTo(), WSARecv(), WSARecvFrom(), and WSAIoctl() for overlapped I/O operations, which initiates multiple operations simultaneously. All functions that allow overlapped operation (WSASend(), WSARecv(), WSASendTo(), WSARecvFrom(), and WSAIoctl()) also support non-overlapped usage on an overlapped socket if the values for parameters related to overlapped operation are NIL.
WSA_FLAG_MULTIPOINT_C_ROOT	Indicates that the socket created will be a c_root in a multipoint session. It is only allowed if a rooted control plane is indicated in the protocol's WSAPROTOCOL_INFO structure.
WSA_FLAG_MULTIPOINT_C_LEAF	Indicates that the socket created will be a c_leaf in a multicast session. It is only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the protocol's WSAPROTOCOL_INFO structure.
WSA_FLAG_MULTIPOINT_D_ROOT	Indicates that the socket created will be a d_root in a multipoint session. It is only allowed if a rooted data plane is indicated in the protocol's WSAPROTOCOL_INFO structure.
WSA_FLAG_MULTIPOINT_D_LEAF	Indicates that the socket created will be a d_leaf in a multipoint session. It is only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the protocol's WSAPROTOCOL_INFO structure.

Now we present a formal definition of the functions.

function socket *Winsock2.pas*

Syntax

socket(af, struct, protocol: integer): TSocket; stdcall;

Description

This function creates a socket.

Parameters

af: Address family

struct: Socket type

protocol: Protocol to use with the socket

Return Value

If the function succeeds, it will return a descriptor referencing the new socket. If the function fails, it will return a value of INVALID_SOCKET. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEAFNOSUPPORT, WSAEINPROGRESS, WSAEMFILE, WSAENOBUFS, WSAEPROTONOSUPPORT, WSAEPROTOTYPE, and WSAESOCKTNOSUPPORT.

See Appendix B for a detailed description of the error codes.

See Also

accept, bind, connect, getsockname, getsockopt, ioctlsocket, listen, recv, recvfrom, select, send, sendto, setsockopt, shutdown, WSASocket

Example

See Listing 5-1 (program EX51).

Listing 5-1: A simple and generic blocking echo client that uses the UDP protocol

```
program EX51;

{$APPTYPE CONSOLE}

uses
  SysUtils,

  WinSock2;

const
 MaxEchoes = 10;
 DataBuffSize = 1024;
 S = 'Hello';

var
  WSAData: TWSAData;
  Host: PHostent;
  HostAddr: TSockAddrIn;
  Addr: PChar;
  skt: TSocket;
```

```
   NoEchoes,
   Size: Integer;
   HostName: String;
   Res: Integer;
   Buffer: array[0..DataBuffSize - 1] of char;

   procedure CleanUp(S : String);
   begin
     WriteLn('Call to ' + S + ' failed with error: ' +  SysErrorMessage(WSAGetLastError));
     WSACleanUp;
     Halt;
   end;

begin
// Check for hostname from option ...
 if ParamCount <> 1 then
 begin
   WriteLn('To run the echo client you must give a host name. For example, localhost for your
            machine.');
   Halt;
 end;
 if WSAStartUp($0202, WSAData) = 0 then
 try
  skt := socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if skt = INVALID_SOCKET then
   CleanUp('socket()');
  HostName := ParamStr(1);
  if inet_addr(PChar(HostName)) <> INADDR_NONE then
   CleanUp('inet_addr()');
   Host := gethostbyname(PChar(HostName));
  if Host = NIL then
   CleanUp('gethostbyname()');
  move(Host^.h_addr_list^, Addr, SizeOf(Host^.h_addr_list^));
  HostAddr.sin_family := AF_INET;
  HostAddr.sin_port := htons(IPPORT_ECHO);
  HostAddr.sin_addr.S_un_b.s_b1 := Byte(Addr[0]);
  HostAddr.sin_addr.S_un_b.s_b2 := Byte(Addr[1]);
  HostAddr.sin_addr.S_un_b.s_b3 := Byte(Addr[2]);
  HostAddr.sin_addr.S_un_b.s_b4 := Byte(Addr[3]);
  StrPCopy(Buffer, S);
  Size := SizeOf(HostAddr);
  for NoEchoes := 1 to MaxEchoes do
  begin
    Res := sendto(skt, Buffer, SizeOf(Buffer) ,0, @HostAddr, SizeOf(HostAddr));
    if Res = SOCKET_ERROR then
      CleanUp('sendto()');
    Res := recvfrom(skt, Buffer, SizeOf(Buffer),0, @HostAddr, Size);
    if Res = SOCKET_ERROR then
      CleanUp('recv()');
    WriteLn(Format('Message [%s] # %2d echoed from %s',[Buffer, NoEchoes,
inet_ntoa(HostAddr.sin_addr)]));
  end;
  closesocket(skt);
 finally
  WSACleanUp;
 end
 else WriteLn('Failed to load Winsock...');
end.
```

function WSASocket Winsock2.pas

Syntax

WSASocket(af: u_int; atype: u_int; protocol: u_int; lpProtocolInfo: PWSAPROTOCOL_INFO; g: TGROUP; dwFlags: DWORD): TSocket; stdcall;

Description

This function creates a socket. By default, the socket created does not have the overlapped attribute set.

Parameters

af: An address family specification

atype: A type specification for the new socket

protocol: A particular protocol to be used with the socket that is specific to the indicated address family

lpProtocolInfo: A pointer to a WSAPROTOCOL_INFO structure that defines the characteristics of the socket to be created

g: Reserved for future use with socket groups; the identifier of the socket group

dwFlags: The socket attribute specification

Return Value

If no error occurs, WSASocket() will return a descriptor referencing the new socket. Otherwise, the function will return a value of INVALID_SOCKET. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEAFNOSUPPORT, WSAEINPROGRESS, WSAEMFILE, WSAENOBUFS, WSAEPROTONOSUPPORT, WSAEPROTOTYPE, WSAESOCKTNOSUPPORT, WSAEINVAL, WSAEFAULT, WSAINVALIDPROVIDER, and WSAINVALIDPROCTABLE.

See Appendix B for a detailed description of the error codes.

See Also

accept, bind, connect, getsockname, getsockopt, ioctlsocket, listen, recv, recvfrom, select, send, sendto, setsockopt, shutdown, socket

Example

See Listings 5-2 and 5-3 (programs EX52 and EX53).

Making the Connection

After creating a socket, you are ready to exchange data—or are you? You cannot exchange data on sockets of the SOCK_STREAM type until the socket is in a connected state. We say that a connection exists when a local socket is connected to the remote socket. With sockets of the SOCK_DGRAM type, you do not normally need to connect with a peer before transmitting the data; however, see the sidebar later in this section titled "Connected and Connectionless Sockets."

There are two functions that you can use to set up a connection with a socket on the remote machine—connect() or WSAConnect(). You should use the WSAConnect() function if you want to specify a minimum level of service for the connection. To specify the required level of service, use the QOS (Quality of Service) specific parameters based on the supplied flow specification. We will not cover QOS, as it is beyond the scope of this book.

Let's consider the simpler function first, which is connect(). We give the prototype, which is defined in Winsock2.pas:

```
function connect(s: TSocket; name: PSockAddr; namelen: Integer): Integer; stdcall;
```

To create a connection with a peer, you need to supply three parameters to the connect() function, which are *s*, the unconnected socket; *name*, a pointer to the sockaddr_in record; and *namelen*, the size of the sockaddr_in record. You have already seen how to create a socket, but we still need to define the details of the peer with which to connect. To define the details of the peer, assign the values to the sockaddr_in record, which is defined in WinSock2.pas as:

```
sockaddr_in = record
  sin_family: Smallint;
  sin_port: u_short;
  sin_addr: in_addr;
  sin_zero: array [0..7] of Char;
end;
TSockAddrIn = sockaddr_in;
PSockAddrIn = ^sockaddr_in;
```

Usually, you need to only assign sensible values to the first three fields—*sin_family*, *sin_port*, and *sin_addr*. The last field, *sin_zero*, can be safely ignored, as it is used to make the size of the record 16 bytes long. However, some implementations use this field to distinguish different addresses bound to the interfaces, which requires *sin_zero* to be populated with zeroes. Delphi automatically assigns the *sin_zero* field to zero. To belabor the point, you should ensure that the *sin_zero* field is set to zero by calling the Win32 function ZeroMemory(), like this:

```
ZeroMemory(sockAddr, SizeOf(TSockAddrIn))
```

Calling this function will zero out all fields including *sin_zero*. Obviously, you should call this function before assigning values.

The *sin_family* field is the protocol family, which is usually PF_INET for the Internet. Note that when you create a socket that uses an address family, say, AF_INET, you must also use the same family, which is PF_INET. The *sin_port* field is the port for the service an application requires. For example, for FTP, this would be 21.

TIP: Recall the fact about byte ordering from Chapter 3 that you use the network byte order for the sin_port field. For example, to use the port for FTP, you would do the following assignment:

```
sockaddr. sin_port := htons(21)
```

where sockAddr is a sockaddr_in record
Ignore this simple caveat at your peril!

The *sin_addr* field is actually a variant record, as shown below:

```
in_addr = record
  case Integer of
    0: (S_un_b: SunB);
    1: (S_un_c: SunC);
    2: (S_un_w: SunW);
    3: (S_addr: u_long);
end;
TInAddr = in_addr;
PInAddr = ^in_addr;
```

How you assign these fields depends on how you resolve the name of the peer with which you wish to connect. When you call any of the following functions, you must use the THostEnt record (see Chapter 3 for details of the structure and how to call these functions to fill the THostEnt record) to populate the fields of the sockaddr_in record: gethostbyname(), WSAGetHostByName(), gethostbyaddr(), and WSAGetHostByAddr(). The following code snippet shows how this is done:

```
Var
 Hostent: PHostent;
 h_addr: PChar;
 HostAddress: TSockAddrIn; // remember this is an alias for sockaddr_in
begin

  Hostent := gethostbyname(PChar(HostName));
  if Hostent <> NIL then
  begin
   Move(Hostent^.h_addr_list^, h_addr, SizeOf(Hostent^.h_addr_list^));
   with HostAddress.sin_addr do
   begin
    S_un_b.s_b1 := Byte(h_addr[0]);
    S_un_b.s_b2 := Byte(h_addr[1]);
    S_un_b.s_b3 := Byte(h_addr[2]);
    S_un_b.s_b4 := Byte(h_addr[3]);
```

After assigning the fields of the TSockAddrIn record, call connect() like this:

```
Res:= connect(skt, @HostAddr, SizeOf(TSockAddrIn));
```

If no error occurs, connect() returns zero to indicate that the connection now exists. Otherwise, it returns SOCKET_ERROR, and you should always call WSAGetLastError() to retrieve the error code.

Similarly, use the WSAConnect() function to set up a connection. However, the function has four more parameters, as the following prototype clearly shows:

```
function WSAConnect(s: TSocket; name: PSockAddr; namelen: Integer; lpCallerData: LPWSABUF;
  lpCalleeData: LPWSABUF; lpSQOS: LPQOS; lpGQOS: LPQOS): Integer; stdcall;
```

However, by setting the last four parameters to NIL, you can call the function in the same way you would call connect(), like this:

```
Res:= WSAConnect(skt, @HostAddr, SizeOf(TSockAddrIn), NIL, NIL, NIL, NIL);
```

However, using WSAConnect() this way is rather pointless as you can achieve the same purpose with the simpler connect() function. To use the WSAConnect() function to its full potential, you need to use parameters like *lpCallerData*, *lpCalleeData*, *lpSQOS*, and *lpGQOS*. The parameters *lpCallerData* and *lpCalleeData* are pointers to user data that is transferred to and from the peer, respectively. The definition of LPWSABUF is defined in Winsock2.pas:

```
  _WSABUF = record
    len: u_long;     // the length of the buffer
    buf: PChar;      // the pointer to the buffer
  end;
  WSABUF = _WSABUF;
  LPWSABUF = ^_WSABUF;
  TWsaBuf = WSABUF;
  PWsaBuf = LPWSABUF;
```

The WSAConnect() function enables the application to request Quality of Service (QOS) for incoming and outgoing traffic. QOS is not discussed in detail in this book.

After a successful connection, you can use the getsockname() and getpeername() functions to retrieve the names of the local and remote sockets, respectively.

> **Connected and Connectionless Sockets**
>
> One of the established wisdoms in Winsock 1.1 is that all connected sockets use SOCK_STREAM (TCP protocol) and connectionless sockets use SOCK_DGRAM (UDP protocol). To use connectionless sockets, you would use the sendto() and recvfrom() functions to send and receive data. With the introduction of Winsock 2, these wisdoms are no longer strictly true. In Winsock 2, you can use the send() and recv() functions, which are normally used for connected sockets, with connectionless sockets. As you shall discover later in this chapter, Winsock 2 has introduced the WSARecv() and WSASend() functions, which are extended versions of recv() and send(), respectively, that you can also use with connectionless sockets. This will only be true provided you use either WSAConnect() or connect() to create the connection in the first place, which you can use to get the default peer address that is required for a connectionless socket. You can also use connected sockets with the sendto(), recvfrom(), WSARecvFrom() and WSASendTo() functions.

function connect Winsock2.pas

Syntax

connect(s: TSocket; name: PSockAddr; namelen: Integer): Integer; stdcall;

Description

This function establishes a connection to a peer.

Parameters

s: An unconnected socket

name: The name of the socket in the sockaddr_in structure

namelen: The length of the name

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEADDRINUSE, WSAEINTR, WSAEINPROGRESS, WSAEALREADY, WSAEADDRNOTAVAIL, WSAEAFNOSUPPORT, WSAECONNREFUSED, WSAEFAULT, WSAEINVAL, WSAEISCONN, WSAENETUNREACH, WSAENOBUFS, WSAENOTSOCK, WSAETIMEDOUT, WSAEWOULDBLOCK, and WSAEACCES.

See Appendix B for a detailed description of the error codes.

See Also

accept, bind, getsockname, select, socket, WSAAsyncSelect, WSAConnect

Example

See Listing 5-2 (program EX54).

Listing 5-2: A simple and generic blocking echo client that uses the TCP protocol (socket stream)

```
program EX54;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  WinSock2;

const
 MaxEchoes = 10;
 DataBuffSize = 1024;
 S = 'Hello';

var
  WSAData: TWSAData;
  Host: PHostent;
  HostAddr,
  RemoteAddr: TSockAddrIn;
  Addr: PChar;
  Msg: PChar;
  skt: TSocket;
  NoEchoes,
  Len,
  Size: Integer;
  HostName : String;
  Res : Integer;
  Buffer: array[0..1024 - 1] of char;
  procedure CleanUp(S : String);
  begin
    WriteLn('Call to ' + S + ' failed with error: ' + SysErrorMessage(WSAGetLastError));
    WSACleanUp;
    Halt;
  end;

begin
// Check for hostname from option ...
 if ParamCount <> 1 then
 begin
   WriteLn('To run the echo client you must give a host name!');
   Halt;
 end;
 if WSAStartUp($0202, WSAData) = 0 then
 try
  skt := socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if skt = INVALID_SOCKET then
   CleanUp('socket()');
  HostName := ParamStr(1);
  if inet_addr(PChar(HostName)) <> INADDR_NONE then
   CleanUp('inet_addr()');
   Host := gethostbyname(PChar(HostName));
  if Host = NIL then
```

```
    CleanUp('gethostbyname()');
   move(Host^.h_addr_list^, Addr, SizeOf(Host^.h_addr_list^));
   HostAddr.sin_family := AF_INET;
   HostAddr.sin_port := htons(IPPORT_ECHO);
   HostAddr.sin_addr.S_un_b.s_b1 := Byte(Addr[0]);
   HostAddr.sin_addr.S_un_b.s_b2 := Byte(Addr[1]);
   HostAddr.sin_addr.S_un_b.s_b3 := Byte(Addr[2]);
   HostAddr.sin_addr.S_un_b.s_b4 := Byte(Addr[3]);
   StrPCopy(Buffer,S);

   Len := Length(S);
   Msg := S;
   Size := SizeOf(HostAddr);
 // Attempt to connect first ...
   Res := connect(skt,@HostAddr, Size);
   if Res = SOCKET_ERROR then
     CleanUp('connect()');
 // Now call getpeername() to get the details of the remote host ...
   Res := getpeername(skt, @RemoteAddr, Size);
   if Res = SOCKET_ERROR then
    CleanUp('getpeername()');
   WriteLn('Details of the remote host:');
   WriteLn(Format('Host name : %s',[String(inet_ntoa(RemoteAddr.sin_addr))]));
   WriteLn(Format('Port      : %d',[ntohs(RemoteAddr.sin_port)]));
   WriteLn;
   for NoEchoes := 1 to MaxEchoes do
   begin
     Res := send(skt, Buffer, SizeOf(Buffer) ,0);
     if Res = SOCKET_ERROR then
       CleanUp('send()');
     Msg := '';
     Res := recv(skt, Buffer, SizeOf(Buffer),0);
     if Res = SOCKET_ERROR then
       CleanUp('recv()');
     WriteLn(Format('Message [%s] # %2d echoed from %s',[Buffer, NoEchoes,
 inet_ntoa(HostAddr.sin_addr)]));
   end;
   closesocket(skt);
  finally

   WSACleanUp;
  end
  else WriteLn('Failed to load Winsock...');
 end.
```


function WSAConnect Winsock2.pas

Syntax

WSAConnect(s: TSocket; name: PSockAddr; namelen: Integer; lpCallerData: LPWSABUF; lpCalleeData: LPWSABUF; lpSQOS: LPQOS; lpGQOS: LPQOS): Integer; stdcall;

Description

The WSAConnect() function establishes a connection to another socket application, exchanges connect data, and specifies needed Quality of Service based on the specified FLOWSPEC structure, which is not discussed here.

Parameters

s: A descriptor identifying an unconnected socket

name: The name of the peer to which the socket is to be connected

namelen: The length of *name*

lpCallerData: A pointer to the user data that is to be transferred to the peer during connection establishment

lpCalleeData: A pointer to the user data that is to be transferred back from the peer during connection establishment

lpSQOS: A pointer to the flow specs for socket *s*, one for each direction

lpGQOS: Reserved for future use with socket groups. This is not implemented in Winsock 2.2.

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEADDRINUSE, WSAEINTR, WSAEINPROGRESS, WSAEALREADY, WSAEADDRNOTAVAIL, WSAEAFNOSUPPORT, WSAECONNREFUSED, WSAEFAULT, WSAEINVAL, WSAEISCONN, WSAENETUNREACH, WSAENOBUFS, WSAENOTSOCK, WSAEOPNOTSUPP, WSAEPROTONOSUPPORT, WSAETIMEDOUT, WSAEWOULDBLOCK, and WSAEACCES.

See Appendix B for a detailed description of the error codes.

See Also

accept, bind, connect, getsockname, getsockopt, select, socket, WSAAsyncSelect, WSAEventSelect

Example

Listing 5-3 (program EX53) provides an example of using a generic echo server with overlapped I/O.

Listing 5-3: A generic echo server that uses overlapped I/O with event notification

```
program EX53;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows,
  WinSock2;

const
 MaxEchoes = 10;
 DataBuffSize = 8192;
```

```
type

 PSocketInfo = ^TSocketInfo;
 TSocketInfo = record
                Overlapped : WSAOverlapped;
                skt : TSocket;
                Buffer : array[0..DataBuffSize - 1] of char;
                DataBuffer : WSABuf;
                BytesSend,
                BytesRecv : DWORD;
end;


var
  WSAData: TWSAData;
  DummyAddr,
  HostAddr: TSockAddrIn;
  sktListen,
  sktAccept: TSocket;
  Size: Integer;
  EventTotal,
  Flags,
  ThreadID,
  RecvBytes: DWORD;
  EventArray : array[0..WSA_MAXIMUM_WAIT_EVENTS - 1] of WSAEVENT;
  SocketInfo : array[0..WSA_MAXIMUM_WAIT_EVENTS - 1] of PSocketInfo;
  Res : Integer;
  CriticalSection : TRTLCriticalSection;

  procedure CleanUp(S : String);
  begin
    WriteLn('Call to ' + S + ' failed with error: ' + SysErrorMessage(WSAGetLastError));
    WSACleanUp;
    Halt;
  end;

 function ProcessIO(lpParameter : Pointer) : DWORD; stdcall;
 var
  BytesTransferred,
  Flags,
  Index,
  RecvBytes,
  i: DWORD;
  SktInfo: PSocketInfo;
 begin
  EventArray[EventTotal] := WSAEVENT(lpParameter);
  while TRUE do
  begin
    Index := WSAWaitForMultipleEvents(EventTotal, @EventArray, FALSE, WSA_INFINITE, FALSE);
    if Index = WSA_WAIT_FAILED then
    begin
      WriteLn('Call to WSAWaitForMultipleEvents() failed with error: ' +
               SysErrorMessage(WSAGetLastError));
      Result := 0;
      Exit;
    end;
    if (Index -  WSA_WAIT_EVENT_0) = 0 then
    begin
     WSAResetEvent(EventArray[0]);
     continue;
```

```
    end;
    SktInfo := PSocketInfo(GlobalAlloc(GPTR, SizeOf(TSocketInfo)));
    SktInfo := SocketInfo[Index - WSA_WAIT_EVENT_0];
    WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
    if (WSAGetOverlappedResult(SktInfo^.skt,@SktInfo^.Overlapped, BytesTransferred, FALSE,
        Flags) = FALSE) then
     if (BytesTransferred = 0) then
     begin
       WriteLn(Format('Closing socket %d',[SktInfo^.skt]));
       if closesocket(SktInfo^.skt) = SOCKET_ERROR then
       begin
         WriteLn(Format('Call to closesocket() failed with error: %s',
                [SysErrorMessage(WSAGetLastError)]));
       end;
       GlobalFree(Cardinal(SktInfo));
       WSACloseEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
// Clean up SocketInfo & EventArray ...
       EnterCriticalSection(CriticalSection);
       if Index - WSA_WAIT_EVENT_0 + 1 <> EventTotal then
        for i := Index - WSA_WAIT_EVENT_0 to EventTotal - 1 do
        begin
          EventArray[i] := EventArray[i+1];
          SocketInfo[i] := SocketInfo[i+1];
        end;
       dec(EventTotal);
       LeaveCriticalSection(CriticalSection);
       continue;
     end;
// Check if the BytesRecv field = 0 ...
    if SktInfo^.BytesRecv = 0 then
    begin
      SktInfo^.BytesRecv := BytesTransferred;
      SktInfo^.BytesSend := 0;
    end
    else
    begin
      SktInfo^.BytesSend := SktInfo^.BytesSend + BytesTransferred;
    end;
    if SktInfo^.BytesRecv > SktInfo^.BytesSend then
    begin
// Post another WSASend() request ...
      ZeroMemory(@SktInfo^.Overlapped, SizeOf(TOverlapped));
      SktInfo^.Overlapped.hEvent := EventArray[Index - WSA_WAIT_EVENT_0];
      SktInfo^.DataBuffer.buf := SktInfo^.Buffer + SktInfo^.BytesSend;
      SktInfo^.DataBuffer.len := SktInfo^.BytesRecv - SktInfo^.BytesSend;
      if WSASend(SktInfo^.skt, @SktInfo^.DataBuffer, 1, SktInfo^.BytesSend, 0,
                 @SktInfo^.Overlapped, NIL) = SOCKET_ERROR then
        if WSAGetLastError <> ERROR_IO_PENDING then
        begin
         WriteLn(Format('Call to WSASend() failed with error: %s',
                [SysErrorMessage(WSAGetLastError)]));
         Result := 0;
         Exit;
        end
    end else
    begin
      SktInfo^.BytesRecv := 0;
// We have more no bytes of data to receive ...
      Flags := 0;
      ZeroMemory(@SktInfo^.Overlapped, SizeOf(TOverlapped));
```

```
      SktInfo^.Overlapped.hEvent := EventArray[Index - WSA_WAIT_EVENT_0];
      SktInfo^.DataBuffer.len := DataBuffSize;
      SktInfo^.DataBuffer.buf := SktInfo^.Buffer;
      if WSARecv(SktInfo^.skt,@SktInfo^.DataBuffer, 1, RecvBytes, Flags,
                 @SktInfo^.Overlapped, NIL) = SOCKET_ERROR then
       if WSAGetLastError <> ERROR_IO_PENDING then
       begin
         WriteLn(Format('Call to WSARecv() failed with error: %s',
                [SysErrorMessage(WSAGetLastError)]));
         Result := 0;
         Exit;
       end;
    end;
  end; // while ...
 end;

begin
 EventTotal := 0;
 InitializeCriticalSection(CriticalSection);
 if WSAStartUp($0202, WSAData) = 0 then
 try
  sktListen := WSASocket(AF_INET, SOCK_STREAM, 0, NIL, 0, WSA_FLAG_OVERLAPPED);
  if sktListen = INVALID_SOCKET then
   CleanUp('WSASocket()');
  HostAddr.sin_family := AF_INET;
  HostAddr.sin_port := htons(IPPORT_ECHO);
  HostAddr.sin_addr.S_addr := htonl(INADDR_ANY);
  Res := bind(sktListen, @HostAddr, SizeOf(HostAddr));
  if Res = SOCKET_ERROR then
   CleanUp('bind()');
  Res := listen(sktListen,5);
  if Res = SOCKET_ERROR then
   CleanUp('listen()');
// Create a socket for accepting connections ...
  sktAccept := WSASocket(AF_INET, SOCK_STREAM, 0, NIL, 0, WSA_FLAG_OVERLAPPED);
  if sktAccept = INVALID_SOCKET then
   CleanUp('WSASocket()');
// Create an event object ...
  EventArray[0] := WSACreateEvent;
  if EventArray[0] = WSA_INVALID_EVENT then
   CleanUp('WSACreateEvent()');
  if CreateThread(NIL, 0, @ProcessIO, NIL, 0, ThreadID) = 0{ NIL} then
   CleanUp('CreateThread()');
  EventTotal := 1;
  DummyAddr.sin_family := AF_INET;
  DummyAddr.sin_port := htons(IPPORT_ECHO);
  DummyAddr.sin_addr.S_addr := INADDR_ANY;
  Size := SizeOf(DummyAddr);
  EventTotal := 1;
// Enter an infinite loop ...
  while TRUE do
  begin
    sktAccept := accept(sktListen, @DummyAddr, @Size);
    if sktAccept = INVALID_SOCKET then
     CleanUp('accept()');
    EnterCriticalSection(CriticalSection);
// Create a socket information structure to associate with the accepted socket ...
    SocketInfo[EventTotal] := PSocketInfo(GlobalAlloc(GPTR, SizeOf(TSocketInfo)));
    if SocketInfo[EventTotal] = NIL then
     CleanUp('GlobalAlloc()');
```

```
// Populate the SktInfo structure ...
   SocketInfo[EventTotal]^.skt := sktAccept;
   ZeroMemory(@SocketInfo[EventTotal]^.Overlapped, SizeOf(TOverlapped));
   SocketInfo[EventTotal]^.BytesSend := 0;
   socketInfo[EventTotal]^.BytesRecv := 0;
   socketInfo[EventTotal]^.DataBuffer.len := DataBuffSize;
   SocketInfo[EventTotal]^.DataBuffer.buf := SocketInfo[EventTotal]^.Buffer;
   EventArray[EventTotal] := WSACreateEvent;
   if EventArray[EventTotal] = WSA_INVALID_EVENT then
    CleanUp('WSACreateEvent()');
  SocketInfo[EventTotal]^.Overlapped.hEvent := EventArray[EventTotal];
// Post a WSARecv() request to begin receiving data on the socket ...
   Flags := 0;
   Res := WSARecv(SocketInfo[EventTotal]^.skt,@SocketInfo[EventTotal]^.DataBuffer,
          1, RecvBytes, Flags, @SocketInfo[EventTotal]^.Overlapped, NIL);
   if Res = SOCKET_ERROR then
    if WSAGetLastError <> ERROR_IO_PENDING then
    begin
     CleanUp('WSARecv()');
     Exit;
    end;
   inc(EventTotal);
   LeaveCriticalSection(CriticalSection);
// Signal the first event in the event array to tell the worker thread to service
// an additional event in the event array ...
   if WSASetEvent(EventArray[0]) = FALSE then
   begin
    CleanUp('WSASetEvent()');
    Exit;
   end;
  end;// while ...
 finally
  WSACleanUp;
 end
 else WriteLn('Failed to load Winsock...');
end.
```

function getpeername   Winsock2.pas

Syntax

getpeername(s: TSocket; name: PSockAddr; var namelen: Integer): Integer; stdcall;

Description

This function retrieves the name of the peer connected to the socket *s* and stores it in the TSockAddr record in the *name* parameter.

Parameters

s: A descriptor identifying a connected socket

name: A pointer to the sockaddr_in record which is to receive the name of the peer

namelen: A pointer to the size of the *name* record

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAEINPROGRESS, WSAENOTCONN, and WSAENOTSOCK.

See Appendix B for a detailed description of the error codes.

See Also

bind, getsockname, socket

Example

See Listing 5-2 (program EX54).

function getsockname Winsock2.pas

Syntax

getsockname(s: TSocket; name: PSockAddr; var namelen: Integer): Integer; stdcall;

Description

This function retrieves the local name for a connected socket specified in the *name* parameter.

Parameters

s: A descriptor identifying a bound socket

name: A pointer to the sockaddr_in record to receive the address (name) of the socket

namelen: The size of the *name* parameter

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAEINPROGRESS, WSAENOTSOCK, and WSAEINVAL.

See Appendix B for a detailed description of the error codes.

See Also

bind, getpeername, socket

Example

See Listings 4-5 and 5-6 (programs EX45 and EX58).

Listing 5-4 provides an example of a generic echo server using the select() function.

Listing 5-4: A generic echo server that uses the select() model

```
program EX55;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows,
  WinSock2;

const
 MaxEchoes = 10;
 DataBuffSize = 8192;
 S = 'Hello';
 TotalSockets : Integer = 0;

type

 PSocketInfo = ^TSocketInfo;
 TSocketInfo = record
                 Overlapped : WSAOverlapped;
                 skt : TSocket;
                 Buffer : array[0..DataBuffSize - 1] of char;
                 DataBuffer : WSABuf;
                 BytesSend,
                 BytesRecv : DWORD;
       end;



var
  WSAData: TWSAData;
  Host: PHostent;
  DummyAddr,
  HostAddr: TSockAddrIn;
  Addr: PChar;
  Msg: PChar;
  sktListen,
  sktAccept: TSocket;
  NoEchoes,
  Len,
  i,
  Size: Integer;
  Flags,
  Total: DWORD;
  ThrdHandle: THandle;
  ThreadID: DWORD;
  AcceptEvent: WSAEvent;
  HostName : String;
  Res : Integer;
  WriteSet,
  ReadSet: FD_Set;
  NonBlock: u_long;
  SendBytes,
  RecvBytes: DWORD;
  SocketArray: array[0..FD_SETSIZE - 1] of PSocketInfo;
  SocketInfo: PSocketInfo;
```

```
 function CreateSocketInformation(skt: TSocket) : Boolean;
 var
   SI: PSocketInfo;
 begin
   WriteLn(Format('Accepted socket number %d', [skt]));
   SI := PSocketInfo(GlobalAlloc(GPTR, SizeOf(TSocketInfo)));
   if SI = NIL then
   begin
      WriteLn('Typecast failed with error');
      Result := FALSE;
      Exit;
   end;
// Prepare SocketInfo structure for use.
   SI^.skt := skt;
   SI^.BytesSEND := 0;
   SI^.BytesRECV := 0;
   SocketArray[TotalSockets] := SI;
   inc(TotalSockets);
   Result := TRUE;
end;

 procedure FreeSocketInformation(Index: DWORD);
 var
   SI : PSocketInfo;
   i: DWORD;
 begin
   SI := PSocketInfo(SocketArray[Index]);
   closesocket(SI^.skt);
   WriteLn(Format('Closing socket number %d', [SI^.skt]));
   GlobalFree(Cardinal(SI));
// Squash the socket array
   for i := Index to TotalSockets do
     SocketArray[i] := SocketArray[i + 1];
   dec(TotalSockets);
   RecvBytes := 0;
   SendBytes := 0;

 end;

  procedure CleanUp(S : String);
  begin
    WriteLn('Call to ' + S + ' failed with error: ' + SysErrorMessage(WSAGetLastError));
    WSACleanUp;
    Halt;
  end;

begin
 if WSAStartUp($0202, WSAData) = 0 then
 try
  sktListen := WSASocket(AF_INET, SOCK_STREAM, 0, NIL, 0, WSA_FLAG_OVERLAPPED);
  if sktListen = INVALID_SOCKET then
   CleanUp('WSASocket()');
  HostAddr.sin_family := AF_INET;
  HostAddr.sin_port := htons(IPPORT_ECHO);
  HostAddr.sin_addr.S_addr := htonl(INADDR_ANY);
  Res := bind(sktListen, @HostAddr, SizeOf(HostAddr));
  if Res = SOCKET_ERROR then
   CleanUp('bind()');
  Res := listen(sktListen,5);
  if Res = SOCKET_ERROR then
```

5 Chapter

```
   CleanUp('listen()');
  for i:= 0 to FD_SETSIZE - 1 do
   SocketArray[i] := AllocMem(SizeOf(TSocketInfo));
// Change the socket mode on the listening socket from blocking to
// non-block so the application will not block waiting for requests.
  NonBlock := 1;
  Res := ioctlsocket(sktListen, FIONBIO, NonBlock);
  if Res = SOCKET_ERROR then
  begin
    WriteLn(Format('Call to ioctlsocket() failed with error %s',
           [SysErrorMessage(WSAGetLastError)]));
    Exit;
  end;
    FD_ZERO(ReadSet);
    FD_ZERO(WriteSet);
  while TRUE do
  begin
// Prepare the Read and Write socket sets for network I/O notification.
    FD_ZERO(ReadSet);
    FD_ZERO(WriteSet);
// Always look for connection attempts.
     _FD_SET(sktListen, ReadSet);
// Set Read and Write notification for each socket based on the
// current state the buffer. If there is data remaining in the
// buffer then set the Write set otherwise the read set.
    for i := 0 to TotalSockets - 1 do
     if SocketArray[i]^.BytesRecv > SocketArray[i]^.BytesSend then
      _FD_SET(SocketArray[i]^.skt, WriteSet)
     else
      _FD_SET(SocketArray[i]^.skt, ReadSet);
    Total := select(0, @ReadSet, @WriteSet, NIL, NIL);
    if Total = SOCKET_ERROR then
    begin
      WriteLn(Format('Call to select() returned with error %d', [WSAGetLastError]));
      Exit;
    end;
// Check for arriving connections on the listening socket.
    if  FD_ISSET(sktListen, ReadSet) then
    begin
      dec(Total);
      sktAccept := WSAAccept(sktListen, NIL, NIL, NIL, 0);
      if sktAccept <> INVALID_SOCKET then
      begin
// Set the accepted socket to non-blocking mode so the server will
// not get caught in a blocked condition on WSASends
        NonBlock := 1;
        if ioctlsocket(sktAccept, FIONBIO, NonBlock) = SOCKET_ERROR then
        begin
          WriteLn(Format('Call to ioctlsocket() failed with error %d', [WSAGetLastError]));
          Exit;
        end;
        if CreateSocketInformation(sktAccept) = FALSE then
          Exit;
      end
      else
      begin
        if WSAGetLastError <> WSAEWOULDBLOCK then
        begin
          WriteLn(Format('Call to accept() failed with error %d', [WSAGetLastError]));
          Exit;
```

```
        end
      end;
    end;
// Check each socket for Read and Write notification until the number
// of sockets in Total is satisfied.
      if  total > 0 then
      for i := 0 to TotalSockets - 1 do //; i++)
      begin
       SocketInfo := PSocketInfo(SocketArray[i]);
// If the ReadSet is marked for this socket then this means data
// is available to be read on the socket.
       if FD_ISSET(SocketInfo^.skt, ReadSet) then
       begin
        dec(Total);
        SocketInfo^.DataBuffer.buf := SocketInfo^.Buffer;
        SocketInfo^.DataBuffer.len := DataBuffSize;
        Flags := 0;
        if WSARecv(SocketInfo^.skt, @SocketInfo^.DataBuffer, 1, RecvBytes,
                   Flags, NIL, NIL) = SOCKET_ERROR then
        begin
          if WSAGetLastError <> WSAEWOULDBLOCK then
          begin
            WriteLn(Format('Call to WSARecv() failed with error %d', [WSAGetLastError]));
            FreeSocketInformation(i);
          end;
          continue;
        end
        else
        begin
          SocketInfo^.BytesRecv := RecvBytes;
    // If zero bytes are received, this indicates the peer closed the
    // connection.
          if RecvBytes = 0 then
          begin
            FreeSocketInformation(i);
            continue;
          end
        end;
       end;//
       // If the WriteSet is marked on this socket then this means the internal
       // data buffers are available for more data.
       if FD_ISSET(SocketInfo^.skt, WriteSet) then
       begin
        dec(Total);
        SocketInfo^.DataBuffer.buf := SocketInfo^.Buffer + SocketInfo^.BytesSEND;
        SocketInfo^.DataBuffer.len := SocketInfo^.BytesRECV - SocketInfo^.BytesSEND;
        if WSASend(SocketInfo^.skt, @SocketInfo^.DataBuffer, 1, SendBytes, 0,
                   NIL, NIL) = SOCKET_ERROR then
         if WSAGetLastError <> WSAEWOULDBLOCK then
         begin
           WriteLn(Format('Call to WSASend() failed with error %d', [WSAGetLastError]));
           FreeSocketInformation(i);
         end;
         continue;
     end
     else
     begin
       SocketInfo^.BytesSend := SocketInfo^.BytesSend + SendBytes;
       if SocketInfo^.BytesSEND = SocketInfo^.BytesRECV then
       begin
```

```
        SocketInfo^.BytesSend := 0;
        SocketInfo^.BytesRECV := 0;
      end;
    end;
 end;
end;

 closesocket(sktListen);
finally
 WSACleanUp;
end
else WriteLn('Failed to load Winsock...');
end.
```

Sending Data

Now that we have shown how to initiate a session with the peer using TCP, we will consider how to send data on UDP and TCP. For TCP, you should use the send() and WSASend() functions; for UDP, you should use the sendto() and WSASendTo() functions. The WSASend() and WSASendTo() functions are Winsock 2 specific functions that extend considerably the scope of the original send() and sendto() functions.

Having initiated the connection with the peer using TCP, you may call either the send() or WSASend() function to dispatch the data. If you are using UDP, you can call either the sendto() or WSASendTo() function.

The send() function sends data on a connected socket. A successful completion of the call to send() does not mean that the data was delivered successfully. You should use the sendto() function on a connectionless socket.

Although you can use the WSASend() and WSASendTo() functions with overlapped and non-overlapped sockets, you should use these functions for overlapped I/O operations. These functions use multiple buffers to perform a "scatter and gather" type of I/O, which will be described in detail in the section titled "I/O Schemes."

By varying the *flags* and *dwFlags* parameters with a constant from Table 5-3, you can modify how you call any of these functions. Briefly, the MSG_DONTROUTE constant tells the function not to perform any routing of data. Routing and diagnostic applications only need to use this constant. The MSG_PEEK constant tells the function to peek at the data in the receiving buffer without taking any data out of the buffer. The MSG_OOB constant tells the function to send or read urgent data in parallel with sending and receiving the normal data stream. This is a potentially useful feature, but as you will see in the sidebar titled "Out-of-Band Data Etiquette," later in the chapter, the constant is not as useful in practice as in theory.

When you use sendto(), you must never send data in chunks greater than SO_MAX_MSG_SIZE, or fragmentation will occur. Not all networks have the same maximum transmission unit (MTU), so sending a datagram greater than

SO_MAX_MSG_SIZE will probably result in broken datagrams, thus increasing unnecessary overheads. In addition, not all TCP/IP service providers at the receiving end are capable of reassembling a large fragmented datagram.

TIP: Before an application sends a datagram, make sure that its size does not exceed SO_MAX_MSG_SIZE. To determine the largest possible datagram, call the getsockopt() function. You will learn how to use this function in Chapter 6.

Table 5-3: Possible values for the flags parameter

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing.
MSG_OOB	Send out-of-band data on a stream style socket (SOCK_STREAM).
MSG_PARTIAL	Specifies that lpBuffers only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned by transports that do not support partial message transmissions.
MSG_PEEK	Copies the data from the system buffer into the receive buffer. The original data remains in the system buffer.

function send Winsock2.pas

Syntax

send(s: TSocket; var buf; len, flags: Integer): Integer; stdcall;

Description

This function sends data on a connected socket *s*. The successful completion of the call to send() does not mean that the data was successfully delivered.

Parameters

s: A descriptor identifying a connected socket

buf: A buffer containing the data to be transmitted

len: The length of the data in *buf*

flags: Specifies the way in which the call is made (see Table 5-3).

Return Value

If the function succeeds, it will return the number of bytes sent. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEACCES, WSAEINTR, WSAEINPROGRESS, WSAEFAULT, WSAENETRESET, WSAENOBUFS, WSAENOTCONN, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAEHOSTUNREACH, WSAEINVAL, WSAECONNABORTED, WSAECONNRESET, and WSAETIMEDOUT.

See Appendix B for a detailed description of the error codes.

See Also

recv, recvfrom, select, sendto, socket, WSAAsyncSelect, WSAEventSelect

Example

See Listing 5-6 (program EX58).

function WSASend   Winsock2.pas

Syntax

WSASend(s: TSocket; lpBuffers: LPWSABUF; dwBufferCount: DWORD; var lpNumberOfBytesSent: DWORD; dwFlags: DWORD; lpOverlapped: LPWSA-OVERLAPPED; lpCompletionRoutine: LPWSAOVERLAPPED_COMPLETION_ROUTINE): Integer; stdcall;

Description

This function extends the functionality provided by the send() function in two important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped send operations.
- It allows multiple send buffers to be specified, making it applicable to the scatter and gather type of I/O.

Parameters

s: A descriptor identifying a connected socket

lpBuffers: A pointer to an array of _WSABUF records

dwBufferCount: The number of _WSABUF records in the *lpBuffers* array

lpNumberOfBytesSent: A pointer to the number of bytes sent by this call if the I/O operation completes immediately

dwFlags: Specifies the way in which the call is made (see Table 5-3)

lpOverlapped: A pointer to a WSAOVERLAPPED record. This is ignored for non-overlapped sockets.

lpCompletionRoutine: A pointer to the completion routine called when the send operation has been completed. This is ignored for non-overlapped sockets.

Return Value

If no error occurs and the send operation has completed immediately, WSASend() will return zero. To retrieve the error code, call the function WSAGetLastError(). Possible error values are WSANOTINITIALISED, WSAENETDOWN, WSAEACCES, WSAEINTR, WSAEINPROGRESS, WSAEFAULT, WSAENETRESET, WSAENOBUFS, WSAENOTCONN,

WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULD-BLOCK, WSAEMSGSIZE, WSAEINVAL, WSAECONNABORTED, WSAECONNRESET, WSA_IO_PENDING, and WSA_OPERATION_ABORTED.

See Appendix B for a detailed description of the error codes.

See Also

WSACloseEvent, WSACreateEvent, WSAGetOverlappedResult, WSASocket, WSAWaitForMultipleEvents

Example

See Listings 5-3, 5-4, 5-5, 5-7, and 5-8 (programs EX53, EX55, EX56, EX52, and EX57).

function sendto Winsock2.pas

Syntax

sendto(s: TSocket; var buf; len, flags: Integer; toaddr: PSockAddr; tolen: Integer): Integer; stdcall;

Description

This function sends a datagram on a connectionless socket to a specific destination. The successful completion of a call to sendto() does not indicate that the data was successfully transmitted. As with the send() and WSASend() functions, by using the *flags* parameter from Table 5-3, you can determine how you should call sendto().

Parameters

s: A connected socket

buf: A buffer containing the data to send

len: The size of the data in *buf*

flags: Specifies the way in which the call is made (see Table 5-3)

toaddr: An optional pointer to the address of the target socket

tolen: The size of the address in *toaddr*

Return Value

If the function succeeds, it will return the number of bytes sent. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEACCES, WSAEINVAL, WSAEINTR, WSAEINPROGRESS, WSAEFAULT, WSAENETRESET, WSAENOBUFS, WSAENOTCONN, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAEHOSTUNREACH,

WSAECONNABORTED, WSAECONNRESET, WSAEADDRNOTAVAIL, WSAEAFNOSUPPORT, WSAEDESTADDRREQ, WSAENETUNREACH, and WSAETIMEDOUT.

See Appendix B for a detailed description of the error codes.

See Also

recv, recvfrom, select, send, socket, WSAAsyncSelect, WSAEventSelect

Example

See Listing 5-3 (program EX53).

function WSASendTo   Winsock2.pas

Syntax

WSASendTo(s: TSocket; lpBuffers: LPWSABUF; dwBufferCount: DWORD; var lpNumberOfBytesSent: DWORD; dwFlags: DWORD; lpTo: PSockAddr; iTolen: Integer; lpOverlapped: LPWSAOVERLAPPED; lpCompletionRoutine: LPWSA-OVERLAPPED_COMPLETION_ROUTINE): Integer; stdcall;

Description

Like the sendto() function, this function sends a datagram to a specific destination. However, the function uses overlapped I/O where applicable and multiple buffers, if applicable, to perform the scatter and gather type of I/O.

Parameters

s: A descriptor identifying a (possibly connected) socket

lpBuffers: A pointer to an array of TWSABUF records. Each TWSABUF record contains a pointer to a buffer and the length of the buffer. This array must remain valid for the duration of the send operation.

dwBufferCount: The number of TWSABUF records in the *lpBuffers* array

lpNumberOfBytesSent: A pointer to the number of bytes sent by this call if the I/O operation completes immediately

dwFlags: Specifies the way in which the call is made (see Table 5-3)

lpTo: An optional pointer to the address of the target socket

iTolen: The size of the address in *lpTo*

lpOverlapped: A pointer to a WSAOVERLAPPED record, which is ignored for non-overlapped sockets

lpCompletionRoutine: A pointer to the completion routine called when the send operation has been completed, which is ignored for non-overlapped sockets

Return Value

If no error occurs and the operation has completed immediately, the function will return the value of zero. To retrieve the error code, call the function WSAGetLastError(). Possible error values are WSANOTINITIALISED, WSAEACCES, WSAEINTR, WSAEINPROGRESS, WSAEFAULT, WSAENET-RESET, WSAENOBUFS, WSAENOTCONN, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSA-EMSGSIZE, WSAEINVAL, WSAECONNABORTED, WSAECONNRESET, WSAEADDRNOTAVAIL, WSAEAFNOSUPPORT, WSAEDESTADDRREQ, WSAENETUN- REACH, WSA_IO_PENDING, and WSA_OPERA-TION_ABORTED.

See Appendix B for a detailed description of the error codes.

See Also

WSACloseEvent, WSACreateEvent, WSAGetOverlappedResult, WSASocket, WSAWaitForMultipleEvents

Example

None

Receiving Data

Now that we know how to transmit data, we must consider how the peer receives the data. For the TCP protocol, these receiving functions are recv() and WSARecv(), and for UDP, they are recvfrom() and WSARecvFrom().

The recvfrom() function uses a connectionless socket to receive a datagram and captures the source address from which the datagram was sent. You should use WSARecvFrom() primarily on a connectionless socket. By selecting a constant from Table 5-3, you can set the *flags* or *lpFlags* parameters in recv(), recvfrom(), WSARecv(), and WSARecvFrom() to modify how you call the function.

function recv   Winsock2.pas

Syntax

recv(s: TSocket; var buf; len, flags: Integer): Integer; stdcall;

Description

This function receives data from a connected socket

Parameters

s: A descriptor identifying a connected socket

buf: A buffer for the incoming data

len: The length of *buf*

flags: Specifies the way in which the call is made (see Table 5-3)

Return Value

If the function succeeds, it will return the number of bytes received. If the connection has been closed gracefully and all data received, the return value will be zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAENOTCONN, WSAEINTR, WSAEINPROGRESS, WSAENETRESET, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAEINVAL, WSAECONNABORTED, WSAETIMEDOUT, and WSAECONNRESET.

See Appendix B for a detailed description of the error codes.

See Also

recvfrom, select, send, socket, WSAAsyncSelect

Example

See Listing 5-6 (program EX58).

function WSARecv Winsock2.pas

Syntax

WSARecv(s: TSocket; lpBuffers: LPWSABUF; dwBufferCount: DWORD; var lpNumberOfBytesRecvd, lpFlags: DWORD; lpOverlapped: LPWSAOVERLAPPED; lpCompletionRoutine: LPWSAOVERLAPPED_COMPLETION_ROUTINE): Integer; stdcall;

Description

This function receives data from a connected socket and extends functionality over the recv() function in three important areas:

- It can be used with overlapped sockets to perform overlapped receive operations.
- It allows multiple receive buffers to be specified, making it applicable to the scatter and gather type of I/O.

- The *lpFlags* parameter is both an INPUT and an OUTPUT parameter, allowing applications to sense the output state of the MSG_PARTIAL flag bit. Note, however, that the MSG_PARTIAL flag bit is not supported by all protocols.

Parameters

s: A descriptor identifying a connected socket

lpBuffers: A pointer to an array of TWSABUF records. Each TWSABUF record contains a pointer to a buffer and the length of the buffer.

dwBufferCount: The number of WSABUF records in the *lpBuffers* array

lpNumberOfBytesRecvd: A pointer to the number of bytes received by this call if the receive operation completes immediately

lpFlags: A pointer to flags

lpOverlapped: A pointer to a WSAOVERLAPPED record (ignored for non-overlapped sockets)

lpCompletionRoutine: A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets)

Return Value

If no error occurs and the operation has completed immediately, the function will return zero. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur. To retrieve the error code, call the function WSAGetLastError(). Possible error values are WSANOTINITIALISED, WSAENETDOWN, WSAENOTCONN, WSAEINTR, WSAEINPROGRESS, WSAENETRESET, WSAENOTSOCK, WSAEFAULT, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAEINV, WSAECONNABORTED, WSAECONNRESET, WSAEDISCON, WSA_IO_PENDING, and WSA_OPERATION_ABORTED.

See Appendix B for a detailed description of the error codes.

See Also

WSACloseEvent, WSACreateEvent, WSAGetOverlappedResult, WSASocket, WSAWaitForMultipleEvents

Example

See Listings 5-3 and 5-7 (programs EX53 and EX57).

function recvfrom Winsock2.pas

Syntax

recvfrom(s: TSocket; var buf; len, flags: Integer; from: PSockAddr; var fromlen: Integer): Integer; stdcall;

Description

This function receives a datagram and captures the source address from which the data was sent.

Parameters

s: A descriptor identifying a bound socket

buf: A buffer for the incoming data

len: The length of *buf*

flags: Specifies the way in which the call is made

from: An optional pointer to a buffer which will hold the source address upon return

fromlen: An optional pointer to the size of the *from* buffer

Return Value

If the function succeeds, it will return the number of bytes received. If the connection has been closed gracefully and all data received, the return value will be zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAEINTR, WSAEINPROGRESS, WSAEINVAL, WSAEISCONN, WSAENETRESET, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAETIMEDOUT, and WSAECONNRESET.

See Appendix B for a detailed description of the error codes.

See Also

recv, send, socket, WSAAsyncSelect, WSAEventSelect

Example

See Listing 5-1 (program EX51).

function WSARecvFrom Winsock2.pas

Syntax

WSARecvFrom(s: TSocket; lpBuffers: LPWSABUF; dwBufferCount: DWORD; var lpNumberOfBytesRecvd, lpFlags: DWORD; lpFrom: PSockAddr; lpFromlen: PInteger; lpOverlapped: LPWSAOVERLAPPED; lpCompletionRoutine: LPWSAOVERLAPPED_COMPLETION_ROUTINE): Integer; stdcall;

Description

This function extends the functionality provided by the recvfrom() function in three important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped receive operations.
- It allows multiple receive buffers to be specified, making it applicable to the scatter and gather type of I/O.
- The *lpFlags* parameter is both an INPUT and an OUTPUT parameter, allowing applications to sense the output state of the MSG_PARTIAL flag bit. Note, however, that the MSG_PARTIAL flag bit is not supported by all protocols.

Parameters

s: A descriptor identifying a socket

lpBuffers: A pointer to an array of TWSABUF records. Each TWSABUF record contains a pointer to a buffer.

dwBufferCount: The number of TWSABUF records in the *lpBuffers* array

lpNumberOfBytesRecvd: A pointer to the number of bytes received by this call if the receive operation completes immediately

lpFlags: A pointer to flags

lpFrom: An optional pointer to a buffer, which will hold the source address upon the completion of the overlapped operation

lpFromlen: A pointer to the size of the *lpFrom* buffer, required only if *lpFrom* is specified

lpOverlapped: A pointer to a WSAOVERLAPPED record (ignored for non-overlapped sockets)

lpCompletionRoutine: A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets)

Return Value

If no error occurs and the operation has completed immediately, the function will return zero. Otherwise, the function will return a value of SOCKET_ERROR. Call WSAGetLastError() to retrieve a specific error code. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated later. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAEINTR, WSAEINPROGRESS, WSAEINVAL, WSAEISCONN, WSAENETRESET, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAECONNRESET, WSAEDISCON, WSA_IO_PENDING, and WSA_OPERATION_ABORTED.

See Appendix B for a detailed description of the error codes.

See Also

WSACloseEvent, WSACreateEvent, WSAGetOverlappedResult, WSASocket, WSAWaitForMultipleEvents

Example

None

Breaking the Connection

When data exchange is complete, you should close the connection with the remote peer and free any sockets allocated for that exchange. To free these sockets, you should call the shutdown() and closesocket() functions, in that order. Calling shutdown() notifies the remote peer that you are done with the data exchange and disables data communication on the socket, which is either receiving or sending data.

If you set the *how* parameter in shutdown() to SD_RECEIVE, the affected socket will not receive any more data from the remote peer. Likewise, if you set *how* to SD_SEND, the socket will not send any data to the remote peer. Setting *how* to SD_BOTH disables both sends and receives.

Then you should call closesocket() to close a socket and free resources allocated to that socket. If you do not call closesocket() to close every socket at the end of a session, you will deplete the pool of socket handles. When you call closesocket(), any data that is pending will be lost. Thus, it is important that an application retrieve any pending data before calling closesocket().

If you attempt to send data on the closed socket, the call will fail with the error WSAENOTSOCK. Note that closing the socket will cause loss of pending data; cancel any pending blocking or asynchronous calls and any pending overlapped operations on WSASend(), WSASendTo(), WSARecv(), WSARecvFrom(), and WSAIoctl() with an overlapped socket.

TIP: For every socket that you open, you must call closesocket() to return socket resources to the system.

You can control the behavior of closesocket() by calling setsockopt() with the socket options SO_LINGER and SO_DONTLINGER, as shown in Table 5-4. We will discuss setsockopt() and these two options in Chapter 6.

Table 5-4: Socket options to control the behavior of closesocket()

Option	Interval	Type of Close	Wait for Close?
SO_DONTLINGER	Don't care	Graceful	No
SO_LINGER	Zero	Hard	No
SO_LINGER	Nonzero	Graceful	Yes

TIP: To prevent accidental loss of pending data on a connection, an application should call shutdown() before calling closesocket().

Winsock 2 introduces two new functions to shut down a connection—WSASendDisconnect() and WSARecvDisconnect(). Calling WSASendDisconnect() is the equivalent of calling shutdown() with SD_SEND, except that WSASendDisconnect() also sends disconnect data in protocols that support it. You should attach the disconnect data to the second parameter for retrieval by the remote peer using WSARecvDisconnect(). If, however, the protocol that you are using does not support the use of disconnect data, you should simply set the second parameter, *lpOutboundDisconnectData*, to NIL. After a successful call, the application cannot send any more data. However, the disabled socket is still open, so you must still call closesocket() to close the socket and release resources allocated to it.

Calling WSARecvDisconnect() is the same as calling shutdown() with SD_RECV, except WSARecvDisconnect() can receive disconnect data in protocols that support it. After a successful call to WSARecvDisconnect(), the application will not receive any more data. Like WSASendDisconnect(), you can receive disconnect data by retrieving the data from the *lpInboundDisconnectData* parameter, provided that it is not set to NIL.

function shutdown *Winsock2.pas*

Syntax

shutdown(s: TSocket; how: Integer): Integer; stdcall;

Description

This function disables data communication on any socket, which is either receiving or sending.

Parameters

s: A descriptor identifying a socket

how: A flag that describes what type of operation will no longer be allowed

Return Value

If the function succeeds, it will return zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINVAL, WSAEINPROGRESS, WSAENOTCONN, and WSAENOTSOCK.

See Appendix B for a detailed description of the error codes.

See Also

connect, socket

Example

See Listing 5-6 (program EX58).

function closesocket *Winsock2.pas*

Syntax

closesocket(s: TSocket): Integer; stdcall;

Description

This function closes a socket.

Parameters

s: A socket to close

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAENOTSOCK, WSAEINPROGRESS, WSAEINTR, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

accept, ioctlsocket, setsockopt, socket, WSAAsyncSelect, WSADuplicateSocket

Example

See Listing 5-1 (program EX51).

function WSASendDisconnect Winsock2.pas

Syntax

```
WSASendDisconnect(s: TSocket; lpOutboundDisconnectData: LPWSABUF):
Integer; stdcall;
```

Description

This function initiates termination of the connection on the connection-oriented socket and sends disconnect data, if any.

> **TIP:** WSASendDisconnect() does not close the socket, and resources attached to the socket will not be freed until closesocket() is invoked.

Parameters

s: A descriptor identifying a socket

lpOutboundDisconnectData: A pointer to the outgoing disconnect data

Return Value

If the function succeeds, it will return zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAENOPROTOOPT, WSAEINPROGRESS, WSAENOTCONN, WSAENOTSOCK, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

connect, socket

Example

None

function WSARecvDisconnect Winsock2.pas

Syntax

WSARecvDisconnect(s: TSocket; lpInboundDisconnectData: LPWSABUF): Integer; stdcall;

Description

This function disables reception on a connection-oriented socket and retrieves the disconnect data from the remote party.

Parameters

s: A descriptor identifying a socket

lpInboundDisconnectData: A pointer to the incoming disconnect data

Return Value

If the function succeeds, it will return zero. Otherwise, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAENOPROTOOPT, WSAEINPROGRESS, WSAENOTCONN, and WSAENOTSOCK.

See Appendix B for a detailed description of the error codes.

See Also

connect, socket

Example

None

Server Applications

Up to now, we have been discussing data exchange from the client's point of view. It is now time for us to examine the functions that any typical server usually requires to service popular protocols, such as FTP, HTTP, SMTP, POP3, and many others.

Preparation

Before a server can service requests from clients on any of these Internet protocols, it has to perform certain operations before it is ready to serve.

To begin with, a server is not required to resolve its own address, so that step falls away. In addition, a server does not require either the connect() or the WSAConnect() functions because it will be listening as opposed to connecting. However, a server follows the same steps as the client to create a socket, but after creating a socket, a server calls the bind() function to associate or *bind* the socket with a port number of the service that the server is to provide. To provide this binding, bind() uses the sockaddr_in data structure, which is the same structure used by the connect() and WSAConnect() functions.

Initially, when you create a socket with the socket() function, it exists in a name space (address family), but it has no name assigned. You should use the bind() function to associate or bind the socket by assigning a local name to it. In the Internet address family, a name space consists of three parts: the address family, a host address, and a port number that identifies the service. The *sin_family* field must contain the address family that you used to create the socket. Otherwise, a WSAEFAULT error will occur.

If you do not care what local address you assign to the server, you may specify the constant, INADDR_ANY, for the *sa_data* field of the *name* parameter. This allows the underlying service provider to use any appropriate network address. For TCP/IP, if you specify the port as zero, the service provider will assign a unique port to the application with a value between 1024 and 5000.

After calling bind(), you can use getsockname() to learn the address and port that has been assigned to the server. However, if the Internet address is set to INADDR_ANY, the getsockname() function will not necessarily be able to supply the address until the socket is connected, since several addresses may be valid if the host is multi-homed. (See Appendix A for the definition of multi-homed.) Then you should call listen() to listen for a connection on the designated port. When a connection request arrives, the listen() function queues the request until the server is ready to deal with the request.

You can only use listen() on sockets that you created using the SOCK_STREAM type. When a connection request arrives, the listen() function queues the request until the server is ready to accept it via the accept() or WSAAccept() function. When the queue is full, the number of connection requests exceeds the backlog value set for the listen() function, and the server sends an error message (WSAECONNREFUSED) back to the client.

An application may call listen() more than once on the same socket, which has the effect of updating the current backlog for the listening socket. The *backlog* parameter is limited to a reasonable value, as determined by the underlying service provider. Illegal values are replaced by the nearest legal value. There is no way to determine the actual backlog value used. However, if you use the SOMAXCONN constant, as defined by Winsock2.pas, the maximum is $7FFFFFFF (2,147,483,647), an extremely large number.

When you get a connection request, call either accept() or WSAAccept() to accept the connection. We will examine accept() first. The details that we provide concerning accept() also apply to WSAAccept(). The prototype for the accept() function is as follows:

```
function accept(s: TSocket; addr: PSockAddr; addrlen: PInteger): TSocket; stdcall;
```

When the server is ready to service a connection request, it will call accept() to accept the connection. The accept() function creates a new socket that has the same properties of the listening socket, including asynchronous events registered with WSAAsyncSelect() or WSAEventSelect(). If there are no connection requests in the queue and the socket is specified as blocking, accept blocks until a connection request appears. Otherwise, if the socket is non-blocking and no pending connections are present on the queue, accept() returns the WSAEWOULDBLOCK error. When this happens, the server application must handle this so that it can continue to listen for more clients.

After accept() returns a new socket handle, the server uses the accepted socket to perform other functions; it does not play any further role in accepting new connection requests. Instead, the original socket allocated to the listen() function continues to listen for new connection requests.

The first parameter, *s*, is the listening socket. The second parameter, *addr*, is filled with the address of the connecting client. The address family in which the communication is occurring determines the exact format of the *addr* parameter. For example, if you use AF_INET (which is the address family you use for the Internet), you should use the sockaddr_in record. The third parameter, *addrlen,* contains the size of the second parameter. Like listen(), you should use accept() only with sockets of the type SOCK_STREAM. If you set either *addr* to NIL or *addrlen* to zero or both, you will not get any information about the remote address of the accepted socket.

The prototype for WSAAccept() is as follows:

```
function WSAAccept(s: TSocket; addr: PSockAddr; addrlen: PInteger;
  lpfnCondition: LPCONDITIONPROC; dwCallbackData: DWORD): TSocket; stdcall;
```

Like accept(), WSAAccept() takes the first connection in the queue of pending connection requests on the listening socket. In addition, if the fourth parameter, a pointer to condition function, *lpfnCondition*, is not NIL, the function checks the request using the callback function. If the condition function returns CF_ACCEPT, this routine creates a new socket. The newly created socket has the same properties as the listening socket, including asynchronous events registered with WSAAsyncSelect() or WSAEventSelect(), which we will cover later in this chapter. If the condition function returns CF_REJECT, then WSAAccept() rejects the connection request. If the decision cannot be made immediately, the condition function will return the value CF_DEFER to indicate that no decision has been made and no action about this connection request should be taken. When the application is ready to act on that connection request, it will invoke WSAAccept() again and return either CF_ACCEPT or CF_REJECT from the condition function.

For sockets that are blocking, and if no pending connections are present in the queue, WSAAccept() will continue to block until a connection request arrives. Otherwise, if the socket is non-blocking and this function is called when no pending connections are present in the queue, WSAAccept() fails with the error WSAEWOULDBLOCK.

After WSAAccept() returns a new socket handle, the server uses the accepted socket to perform a task. The original listening socket remains open for new connection requests.

The second parameter, *addr*, is filled with the address of the connecting client. This call is used with connection-oriented socket types, such as SOCK_STREAM. The condition function, defined in Winsock2.pas, is as follows:

```
LPCONDITIONPROC = function (lpCallerId, lpCallerData: LPWSABUF; lpSQOS, lpGQOS: LPQOS;
  lpCalleeId, lpCalleeData: LPWSABUF; g: PGroup; dwCallbackData: DWORD): Integer; stdcall;
```

The *LPCONDITIONPROC* parameter is a pointer to the callback procedure in WSAAccept(). *lpCallerId* and *lpCallerData* are parameters that contain the address of the connecting client and any user data that was sent with the connection request, respectively. Many network protocols do not support connect-time caller data (*lpCallerData*). However, most conventional network protocols can support caller ID (*lpCallerId*) information at connection-request time. The *buf* field of the _WSABUF (see the definition for the prototype) pointed to by *lpCallerId* points to a sockaddr_in data structure. The sockaddr_in is interpreted according to its address family (typically by casting the sockaddr_in to some type specific to the address family).

The *lpSQOS* parameter references the flow specifications for the socket specified by the caller, one for each direction, and followed by any additional provider-specific parameters. The sending or receiving flow specification values will be ignored as inappropriate for any unidirectional sockets. If *lpSQOS* is set to NIL, there is no caller-supplied QOS and no negotiation is possible. A valid *lpSQOS* indicates that a QOS negotiation is to occur or the provider is prepared to accept the QOS request without negotiation.

A NIL value for *lpGQOS* indicates no caller-supplied group QOS. QOS information may be returned if a QOS negotiation is to occur. (In any case, set this parameter to NIL, as this feature is not implemented in the current version of Winsock 2.)

lpCalleeId is a parameter that contains the local address of the connected client. The *buf* field of the _WSABUF pointed to by *lpCalleeId* points to a sockaddr_in structure. The sockaddr_in is interpreted according to its address family (typically by casting the sockaddr_in to some type that is specific to the address family).

lpCalleeData is a parameter used by the condition function to supply user data back to the connecting client. *lpCalleeData^.len* contains the length of the buffer allocated by Winsock and pointed to by *lpCalleeData^.buf*. If the length of the buffer is zero, that is, empty, no user data will be transmitted back to the connecting client. As data arrives, the condition function copies the amount of the data up to the limit set by *lpCalleeData^.len* bytes of data into *lpCalleeData*^.*buf*, and then updates *lpCalleeData*^.*len* to indicate the actual number of bytes transferred. If no user data is to be passed back to the caller, the condition function should set *lpCalleeData*^.*len* to zero.

The *dwCallbackData* parameter value passed to the condition function is the value passed as the *dwCallbackData* parameter in the original WSAAccept() call. Only the Winsock 2 client interprets this value. This allows a client to pass some context information from the server through to the condition function. This gives the condition function any additional information required to determine whether to accept the connection or not. A typical usage is to pass a (suitably cast) pointer to a data structure containing references to application-defined objects with which this socket is associated.

Duplicated Sockets

Winsock 2 introduces a new function, WSADuplicateSocket(), to allow a server to farm out another socket to serve a client while attending to other requests.

A server, or a *parent process,* calls WSADuplicateSocket() to obtain a special TWSAPROTOCOL_INFO record. The server then passes the contents of this record via a mechanism (usually by an InterProcess Call) to a child process, which in turn uses it in a call to WSASocket() to obtain a descriptor for the duplicated socket. Note that the child process can only access this special TWSAPROTOCOL_INFO record once. Alternatively, you can share sockets across threads in the same process without using WSADuplicateSocket(), since a socket descriptor is valid in all of a process's threads. Because Winsock does not implement any type of access control, you will need to write extra code that will manage the participating processes to coordinate their operations on a shared socket.

When you use shared or duplicated sockets, you need to remember that if you call setsockopt() (see Chapter 6 for details on this function) to change attributes of the original socket, the change will be reflected in the duplicated sockets. Calling closesocket() on a duplicated socket will remove that socket, but the original socket will remain open. Event notification on shared sockets is subject to the usual constraints of WSAAsyncSelect() and WSAEventSelect(). Calling either of these functions on any of the shared sockets will cancel any previous event registration for that socket, regardless of which socket was used to make that registration. Thus, a shared socket cannot deliver FD_READ events to process A and FD_WRITE events to process B. For situations when such tight coordination is required, we suggest that you use threads instead of separate processes.

function bind Winsock2.pas

Syntax

```
bind(s: TSocket; name: PSockAddr; namelen: Integer): Integer; stdcall;
```

Description

This function binds or associates a local address with a socket. Binding to a specific port number (other than port 0) is discouraged for client applications, since there is a danger of conflicting with another socket that is already using that port number.

Parameters

s: A descriptor identifying an unbound socket

name: The address to assign to the socket

namelen: The length of *name*

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEADDRINUSE, WSAEADDRNOTAVAIL, WSAEFAULT, WSAEINPROGRESS, WSAEINVAL, WSAENOBUFS, and WSAENOTSOCK.

See Appendix B for a detailed description of the error codes.

See Also

connect, getsockname, listen, setsockopt, socket, WSACancelBlockingCall

Example

See Listing 5-7 (program EX52).

function listen Winsock2.pas

Syntax

```
listen(s: TSocket; backlog: Integer): Integer; stdcall;
```

Description

This function establishes a socket to listen for an incoming connection.

Parameters

s: A descriptor identifying a bound, unconnected socket

backlog: The maximum length to which the queue of pending connection requests may grow. If this value is SOMAXCONN, then the underlying service provider responsible for socket *s* will set the backlog to a maximum "reasonable" value.

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEADDRINUSE, WSAEINPROGRESS, WSAEINVAL, WSAEISCONN, WSAEMFILE, WSAENOBUFS, WSAENOTSOCK, and WSAEOPNOTSUPP.

See Appendix B for a detailed description of the error codes.

See Also

accept, connect, socket

Example

See Listing 5-7 (program EX52).

function accept   Winsock2.pas

Syntax

accept(s: TSocket; addr: PSockAddr; addrlen: PInteger): TSocket; stdcall;

Description

This function takes the first connection in the queue of pending connections on the listening socket and returns a handle to the new socket created by the function.

Parameters

s: A descriptor for the socket that was called with the listen() function

addr: An optional pointer to a buffer that receives the address of the connecting client. The exact format of the *addr* argument is determined by the address family established when the socket was created.

addrlen: An optional pointer to an integer that contains the length of the address *addr*

Return Value

If successful, the function will return a value of type TSocket, which is a descriptor for the accepted socket. Otherwise, it will return a value of INVALID_SOCKET. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAEINTR, WSAEINPROGRESS, WSAEINVAL, WSAEMFILE, WSAENOBUFS, WSAENOTSOCK, WSAEOPNOTSUPP, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

bind, connect, listen, select, socket, WSAAccept, WSAAsyncSelect

Example

See Listing 5-7 (program EX52).

function WSAAccept   Winsock2.pas

Syntax

WSAAccept(s: TSocket; addr: PSockAddr; addrlen: PInteger; lpfnCondition: LPCONDITIONPROC; dwCallbackData: DWORD): TSocket; stdcall;

Description

This function performs the same operation as accept(). In addition, the function has extra functionality in three areas:

- Conditionally accepts a connection based on the return value of a condition function
- Provides QOS flowspecs
- Allows transfer of connection data

Parameters

s: A descriptor for the socket that was called with the listen() function

addr: An optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* argument is determined by the address family established when the socket was created.

addrlen: An optional pointer to an integer that contains the length of the address *addr*

lpfnCondition: The procedure instance address of the optional, application-supplied, condition function that will make an accept or reject decision based on the caller information passed in as parameters and optionally create and/or join a socket group by assigning an appropriate value to the result parameter *g* of this function.

dwCallbackData: The callback data passed back to the application as the value of the *dwCallbackData* parameter of the condition function. Winsock does not interpret this parameter.

Return Value

If successful, the function will return a value of type TSocket, which is a descriptor for the accepted socket. Otherwise, the function will return a value of INVALID_SOCKET. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED,

WSAECONNREFUSED, WSAENETDOWN, WSAEFAULT, WSAEINTR, WSAEINPROGRESS, WSAEINVAL, WSAEMFILE, WSAENOBUFS, WSAENOTSOCK, WSAEOPNOTSUPP, WSATRY_AGAIN, WSAEWOULDBLOCK, and WSAEACCES.

See Appendix B for a detailed description of the error codes.

See Also

accept, bind, connect, getsockopt, listen, select, socket, WSAAsyncSelect, WSAConnect

Example

See Listing 5-4 (program EX55).

function WSADuplicateSocket Winsock2.pas

Syntax

WSADuplicateSocket(s: TSocket; dwProcessId: DWORD; lpProtocolInfo: LPWSAPROTOCOL_INFOW): Integer; stdcall;

Description

This function returns a pointer to the WSAPROTOCOL_INFO record that you use to create a new socket descriptor for a shared socket.

Parameters

s: Specifies the local socket descriptor

dwProcessId: Specifies the ID of the target process for which the shared socket will be used

lpProtocolInfo: A pointer to a buffer allocated by the client that is large enough to contain a WSAPROTOCOL_INFO data structure. The service provider copies the contents to this buffer.

Return Value

If the function succeeds, it will return zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINVAL, WSAEINPROGRESS, WSAEMFILE, WSAENOBUFS, WSAENOTSOCK, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

WSASocket

Example

None

I/O Schemes

In this section, we will show you how to use standard I/O schemes. These schemes use functions like select(), WSAAsyncSelect(), and WSAEventSelect(). The helper functions for WSAAsyncSelect() and WSAEventSelect() are WSACreateEvent(), WSAEnumNetworkEvents(), WSACloseEvent(), WSAResetEvent(), and WSAWaitForMultipleEvents(). We will also look at the WSAGetOverlappedResult() function for overlapped I/O operations. We will also explain when and how you would use these schemes. For now, we will introduce these functions briefly.

When you use the select() function for network event notification, the sockets that you use block by default. However, you can use ioctlsocket() (see Chapter 6, "Socket Options," for more details on this function) to make the sockets non-blocking. We will discuss non-blocking and blocking sockets later in the "To Block or Not to Block?" section.

The select() function is often known as a socket multiplex handler because it can handle sets of sockets for reading and writing. The maximum number of sockets that select() can handle is 64. To increase the number of sockets for an application using select(), you can use threads—one set of sockets for each thread. However, why make your life harder than it already is? Is there a more sane approach than using select()? Yes; Winsock has an asynchronous version of select() that takes advantage of Windows' messaging system. This is the WSAAsyncSelect() function.

Another function that is similar to WSAAsyncSelect() is WSAEventSelect(). The advantage of using WSAEventSelect() is that it does not require Windows handles. This is perfect for servers and daemons, as they do not usually require GUI front ends. Instead of using window handles, WSAEventSelect() uses the event object model for notification of network events. We will demonstrate the use of the WSAEventSelect() in a console application for a simple echo server. In any case, we would advise you to use WSAAsyncSelect() or WSAEventSelect() over select() since these functions are easier to code and more robust.

Using Select()

Use the select() function to manage a collection of sockets. The function is a Winsock derivative of the select() function in the Berkeley socket implementations and is provided for compatibility reasons for Berkeley socket applications. The function is useful on Windows CE, where the current version of Winsock does not provide asynchronous sockets and event objects. The select() function is a synchronous version of WSAAsyncSelect(), but is much more difficult to program. WSAAsyncSelect() and WSAEventSelect() are much more friendly and efficient to use than select(). However, we will give a brief description of the select() function as well as a code example to complete our coverage of communications functions.

The function responds to three events:

- Detects data on a socket ready to read using the recv() function
- Detects data on a socket ready to write using the send() function
- Detects out-of-band data on sockets

How do you use the select() function to respond to these events? Let's look first at the prototype for select, which is defined in Winsock2.pas as follows:

```
function select(nfds: Integer; readfds, writefds, exceptfds: PFdSet; timeout: PTimeVal):
Integer; stdcall;
```

You should ignore the first parameter, *nfds*, which is kept for compatibility with Berkeley socket applications. More importantly, the next three parameters are the heart of select()—*readfds*, *writefds*, and *exceptfds*. These are pointers to the fd_set record, which is defined in Winsock2.pas as follows:

```
fd_set = record
  fd_count: u_int;                                  // how many are SET?
  fd_array: array [0..FD_SETSIZE - 1] of TSocket;   // an array of SOCKETs
end;
TFdSet = fd_set;
PFdSet = ^fd_set;
```

The *readfds* parameter points to a collection of sockets for reading, and *writefds* points to a similar collection for writing. The *exceptfds* parameter is a pointer to a collection of sockets for out-of-band data.

Another parameter, *timeout*, is a pointer to the TTimeVal packed record for setting timeouts. The prototype of this data structure, defined in Winsock2.pas, is as follows:

```
timeval = record
  tv_sec: Longint;         // seconds
  tv_usec: Longint;        // and microseconds
end;
TTimeVal = timeval;
PTimeVal = ^timeval;
```

If *timeout* is NIL, select() will block indefinitely waiting for data on the receiving or sending sockets. If you provide values for the *tv_sec* and *tv_usec* fields, select() will wait for a number of seconds, as indicated in *tv_sec*, and milliseconds, as set in *tv_usec*. If you set these values to zero, select() will return immediately, but the code will need to poll select() frequently, which is not efficient.

Before using select(), you will need to initialize the data structures by adding socket handles to them. Winsock provides useful routines to manipulate these data structures, including initialization. These routines are in Table 5-5.

Table 5-5: Routes to manipulate data structures

Name	Description
FD_CLR	Removes the descriptor s from set.
FD_ISSET	Nonzero if s is a member of the set; zero otherwise.
_FD_SET	Adds descriptor s to set.
FD_ZERO	Initializes the set to NIL.

Below is a sequence of steps that you must perform before using select():

- Use the FD_ZERO routine to initialize the data structures (i.e., *readfds*, *writefds*, and *exceptfds*).
- Use _FD_SET to add socket handles for reading to *readfds*. Repeat the same procedure for socket handles for writing to *writefds*. In some applications, it may only be necessary to use select() on sockets for reading only, in which case you may just initialize the set of sockets for reading and ignore the set for writing. Optionally, you can add socket handles to *exceptfds* for out-of-band data, but in our opinion, it is poor programming practice to use out-of-band data (see the "Out-of-Band Data Etiquette" section).

The following steps show how you would use select() in a simple application:

Step 1: Call select(), and wait for I/O activity to complete. The function returns the total number of socket handles for each set of sockets.

Step 2: Using the number of socket handles returned by select(), you should call the FD_ISSET routine to check which sockets have pending I/O in what set.

Step 3: Process the sockets with pending I/O and return to Step 1 to call select() again. This scheme continues until some predefined condition is met.

There is a simple echo server example (EX55) that uses select() that you can study in Listing 5-4.

Using WSAAsyncSelect()

Calling WSAAsyncSelect() notifies Winsock to send a message to a nominated window whenever a network event occurs. You should specify which network events to detect when you make a call to WSAAsyncSelect(). Calling WSA-AsyncSelect() automatically sets the socket in non-blocking mode. The prototype for WSAAsyncSelect() is defined in Winsock2.pas as follows:

```
function WSAAsyncSelect(s: TSocket; hWnd: HWND; wMsg: u_int; lEvent: Longint): Integer;
stdcall;
```

The first parameter, *s*, is the socket that you want to put into non-blocking or, more correctly, asynchronous mode. The second parameter, *hWnd*, specifies the handle to the window for notification. The *wMsg* parameter identifies the message that the window handle is to receive when a network event occurs. The value of the *wMsg* parameter must be greater than the value of WM_USER to avoid message conflicts. The last parameter, *lEvent*, specifies which network events to monitor. The network events you may specify are listed in Table 5-6. To monitor more than one network event, you should call WSAAsyncSelect(), like this:

```
WSAAsyncSelect(s, hwnd, WM_SOCKET, FD_CONNECT or FD_READ or FD_WRITE or FD_CLOSE)
```

This tells Winsock to monitor the following network events that occur when a connection is made, pending read I/O, pending write I/O, or a connection is closed, respectively.

Table 5-6: Network events

Value	Meaning
FD_READ	Required to receive notification of readiness for reading
FD_WRITE	Required to receive notification of readiness for writing
FD_OOB	Required to receive notification of the arrival of out-of-band data
FD_ACCEPT	Required to receive notification of incoming connections
FD_CONNECT	Required to receive notification of completed connection or multipoint join operation
FD_CLOSE	Required to receive notification of socket closure
FD_QOS	Required to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Reserved for future use with socket groups; required to receive notification of socket group Quality of Service (QOS) changes
FD_ROUTING_INTERFACE_CHANGE	Required to receive notification of routing interface changes for the specified destination(s)
FD_ADDRESS_LIST_CHANGE	Required to receive notification of local address list changes for the socket's protocol family

During the lifetime of an application, you will often call WSAAsyncSelect() for a socket more than once. A new call to WSAAsyncSelect() will cancel any previous WSAAsyncSelect() or WSAEventSelect() calls for the same socket. For example, to receive notification for reading and writing, the application must call WSAAsyncSelect() with both FD_READ and FD_WRITE, as the following code snippet illustrates:

```
WSAAsyncSelect(s, hWnd, wMsg, FD_READ OR FD_WRITE);
```

It is not possible to specify different messages for different events. The following code will not work properly because the second call to WSAAsyncSelect() will cancel the effects of the first call, and only FD_WRITE events will be reported with message *wMsg2*:

```
WSAAsyncSelect(s, hWnd, wMsg1, FD_READ);  // first call
WSAAsyncSelect(s, hWnd, wMsg2, FD_WRITE); // second call overwrites original event
                                             notification
```

To cancel all notifications, you need to set *lEvent* to zero, like this:

```
WSAAsyncSelect(s, hWnd, 0, 0);
```

Although in this case, calling WSAAsyncSelect() immediately disables event message notification for the socket *s*, it is possible that messages may still be waiting in the application's message queue. The application must still receive network event messages even after cancellation. Closing a socket with closesocket() also cancels WSAAsyncSelect() message sending, but the same caveat about messages in the queue prior to calling the socket() function still applies.

When you call WSAAsyncSelect(), you must always check for any result from the function. It is nearly always the case that the function could return a non-fatal error of WSAEWOULDBLOCK, which means that the socket has no pending data for reading or writing. The code that you write with WSAAsyncSelect() must handle this error as well as other errors. The code in Listing 5-8 shows how you should handle the WSAEWOULDBLOCK error.

So far, we have discussed the notification of events, but we must complete the puzzle by associating a procedure to handle the events themselves. We usually declare a message procedure somewhere in the interface section or in a class or component like this:

```
procedure SomeEvent(var Mess : TMessage); message NETWORK_EVENT;
```

When you call WSAAsyncSelect(), you link this message procedure with the message NETWORK_EVENT like this:

```
WSAAsyncSelect(s, hwnd, NETWORK_EVENT, FD_CONNECT or FD_READ or FD_WRITE or FD_CLOSE);
```

When you get a network event that you have requested Winsock to monitor on your behalf, you must check for any errors on that event. To do this vital check, you should call WSAGetSelectError() to evaluate the *LParam* field of the *Mess* parameter returned in the SomeEvent procedure. If WSAGetSelectError() returns zero, the network event is normal; otherwise, if there is a network error, call WSAGetSelectError() again to determine the actual error. Whatever error you get, your code must handle it gracefully. The prototype of WSAGet-SelectError() is defined in Winsock2.pas as follows:

```
function WSAGetSelectError(Param: Longint): Word;
```

After verifying that the event has no errors, call WSAGetEventSelect() to determine which event has occurred. The prototype is:

```
function WSAGetSelectEvent(Param: Longint): Word;
```

Pass the *LParam* field of the *Mess* parameter for inspection. The function returns a network event. When you get an FD_READ event, the socket has pending data ready to receive. Likewise, with FD_WRITE, the socket is ready to send data.

Using WSAEventSelect()

The WSAEventSelect() function is similar to WSAAsyncSelect(), except that WSAEventSelect() does not use a window handle for network event notification. Instead, WSAEventSelect() creates an event object for each socket. The function associates the event object with the network events that you wish Winsock to monitor on your behalf. WSAEventSelect() processes the same events as enumerated in Table 5-6. The prototype for WSAEventSelect() is defined in Winsock2.pas as follows:

```
function WSAEventSelect(s: TSocket; hEventObject: WSAEVENT; lNetworkEvents: Longint):
Integer; stdcall;
```

The first parameter, *s*, is the socket that you want to monitor. The second parameter, *hEventObject*, is the event object, which you create by calling WSACreateEvent(). The prototype for WSACreateEvent(), which is defined in Winsock2.pas, is as follows:

```
function WSACreateEvent:WSAEVENT; stdcall;
```

After calling WSACreateEvent(), this function creates an event associated with a particular socket. The WSAEVENT data structure, which is also defined in Winsock2.pas, is simply a handle.

The last parameter in WSAEventSelect(), *lNetworkEvents*, is a bit mask that represents the network events of interest, such as those listed in Table 5-6. Like WSAAsyncSelect(), you must perform a bit-wise operation to include more than one event of interest. For example, if you wish to monitor FD_READ and FD_WRITE events, then you would call the function like this:

```
WSAEventSelect(s, hEvent, FD_READ or FD_WRITE);
```

Normally, you would call WSAEventSelect() only once in the lifetime of an application. Sometimes, though, you might find it necessary to call WSAEventSelect() more than once, in which case, the caveat that applies to WSAAsyncSelect() also applies to WSAEventSelect(). That is, calling WSAEventSelect() for the second time will replace the original settings with fresh settings.

The event object has two operating states, signaled and non-signaled, as well as two operating modes, manual reset and auto reset. When the event object is created, it is in the non-signaled state, and its operating mode is manual reset. Whenever a network event occurs that is associated with a socket marked for monitoring, the event object's operating state changes from non-signaled to signaled. When this happens, the application should call WSAResetEvent() to reset the event object back to the non-signaled state for further monitoring. The prototype of WSAResetEvent() is defined in Winsock2.pas as follows:

```
function WSAResetEvent(hEvent: WSAEVENT): BOOL; stdcall;
```

The parameter, *hEvent*, is the event object that you wish to reset. The function returns TRUE if the call is successful; otherwise, it returns FALSE for an error condition. When you are finished with the event object, you must close it by calling WSACloseEvent() to free resources allocated to that event object. The prototype for WSACloseEvent() is defined in Winsock2.pas as follows:

```
function WSACloseEvent(hEvent: WSAEVENT): BOOL; stdcall;
```

After associating a socket with the event object, your application can start processing I/O by waiting for network events to trigger the event object's operating state. The WSAWaitForMultipleEvents() function monitors these network events by waiting on one or more event objects. The function returns whenever a network event occurs to trigger an event object or when a set timeout interval expires. The prototype for WSAWaitForMultipleEvents() is defined in Winsock2.pas as follows:

```
function WSAWaitForMultipleEvents(cEvents: DWORD; lphEvents: PWSAEVENT;
  fWaitAll: BOOL; dwTimeout: DWORD; fAlertable: BOOL): DWORD; stdcall;
```

The first parameter, *cEvents*, specifies the number of event objects in an array. The second parameter, *lphEvents*, specifies a pointer to that array of event objects. In the current implementation of Winsock, WSAWaitForMultiple-Events() can support a maximum of 64 event objects, which means, therefore, that the function can only support 64 sockets. However, to circumvent this restriction, you can create additional worker threads, each using WSAWaitFor-MultipleEvents() for that thread. The third parameter, *fWaitAll*, specifies the behavior of WSAWaitForMultipleEvents(). If this parameter is TRUE, then WSAWaitForMultipleEvents() will only return when all event objects are in a signaled state. Otherwise, the function returns as soon as any of the event objects become signaled. You should set this parameter to FALSE when you are only using one socket at a time.

The fourth parameter, *dwTimeout*, specifies the time in milliseconds for WSAWaitForMultipleEvents() to wait for a network event. If no network events are ready before the timeout interval elapses, then WSAWaitForMultiple-Events() returns the constant WSA_WAIT_TIMEOUT. The last parameter, *fAlertable*, should always be FALSE. This parameter should only be set to TRUE when you use completion routines in an overlapped I/O scheme, which we will discuss later.

When a network event occurs, WSAWaitForMultipleEvents() returns a value indicating which event object caused the function to return. At this point, the application determines which event object caused the function to return by indexing into the event array for a signaled event object and matching the socket associated with the event object. You do this by using the following code snippet:

```
Index := WSAWaitForMultipleEvents(NoEvents, EventArray, …);
SignaledEvent := EventArray[Index-WSA_WAIT_EVENT_0];
```

Having retrieved the event object and its matching socket, you need to determine what type of network event has occurred. You call WSAEnum-NetworkEvents() to enumerate the event that interests you. The prototype for WSAEnumNetworkEvents() is defined in Winsock2.pas as follows:

```
function WSAEnumNetworkEvents(s: TSocket; hEventObject: WSAEVENT; lpNetworkEvents:
LPWSANETWORKEVENTS): Integer; stdcall;
```

The first parameter, *s*, is the socket that is associated with the signaled event object. The second parameter, *hEventObject*, is set to that event object that became signaled. On return, the event object will be reset to the non-signaled state automatically. This is an optional parameter, which you can set to NIL. However, your application must call WSAResetEvent() to reset the signaled event object for further processing. The last parameter, *lpNetworkEvents*, is a pointer to the data structure _WSANETWORKEVENTS, which is defined in Winsock2.pas. We show its prototype here:

```
  WSANETWORKEVENTS = record
    lNetworkEvents: Longint;
    iErrorCode: array [0..FD_MAX_EVENTS - 1] of Integer;
  end;
  WSANETWORKEVENTS   = _WSANETWORKEVENTS;
  LPWSANETWORKEVENTS = ^WSANETWORKEVENTS;
  TWsaNetworkEvents  = WSANETWORKEVENTS;
  PWsaNetworkEvents  = LPWSANETWORKEVENTS;
```

This data structure contains the information that you need to determine which network event has occurred. The *lNetworkEvents* field is a bit mask containing those network events you have specified in the call to WSAEventSelect(). To retrieve the network events from this parameter, you must perform an AND operation, like this following code snippet:

```
if (lNetworkEvents and FD_READ) = FD_READ then
begin// Yes, this is a FD_READ event … so process it …
 if iErrorCode[1] = WSAENETDOWN then
 begin
  Msg := 'Network down...';
  // Broadcast error message …
 end else
begin
 // Process whatever it needs to be done …
end;
end;
```

Notice that in the preceding code snippet, we have used another field of the _WSANETWORKEVENTS data structure, *iErrorCode*, to check for an error condition that may have existed when the FD_READ event occurred. One of the common network errors that you should guard against is a network failure, which can happen at any time.

After processing the event, the application should continue to monitor network events until some condition is met or when the application ends. Listing

5-8 demonstrates a working echo server using WSAEventSelect() and its helper functions.

Using Overlapped Routines

Overlapped routines are those functions that use the overlapped data structure. Those functions are WSAAccept(), WSASend(), WSARecv(), WSASendTo(), and WSARecvFrom(). We examined these functions earlier in this chapter, but we have left the discussion of overlapped I/O until now.

To use these functions to perform overlapped I/O, we must specify the sockets as having the overlapped attribute set. Recall earlier that using the function WSASocket() with the flag WSA_FLAG_OVERLAPPED will create an overlapped socket. Here is a code snippet that shows how to do this:

```
skt := WSASocket(AF_INET, SOCK_STREAM, 0, NIL, WSA_FLAG_OVERLAPPED);
```

You can call the socket() function instead, which will create an overlapped socket by default. The data structure for implementing overlapped I/O is a Win32 structure, which is defined in Windows.pas. We list its prototype here as follows:

```
POverlapped = ^TOverlapped;
_OVERLAPPED = record
  Internal: DWORD;
  InternalHigh: DWORD;
  Offset: DWORD;
  OffsetHigh: DWORD;
  hEvent: THandle;
end;
TOverlapped = _OVERLAPPED;
OVERLAPPED = _OVERLAPPED;
```

You may access the overlapped structure via the WSAOVERLAPPED alias defined in Winsock2.pas. With the exception of the *hEvent* field, the fields are for system use only. The *hEvent* field represents an event that you link with an overlapped I/O request. To create this event, you should call WSACreate-Event(). When you have created the event handle, assign this to the *hEvent* field. Listing 5-3 demonstrates this technique. The second method we shall examine is the completion routine, which operates differently from the event notification method. Listing 5-2 demonstrates this technique.

Recall that functions such as WSASend() that use overlapped data structures always return immediately. If the I/O operation completes successfully, the overlapped function will return a value of zero. The associated event object has been signaled or a completion routine is queued. However, if the I/O operation fails, the function will return a value of SOCKET_ERROR. At this point, you should check if the error code is WSA_IO_PENDING, which indicates that the overlapped operation has been successfully started but completed. Eventually, when the send buffers (in the case of WSASend()) are empty or receive buffers (in the case of WSARecv()) are full, Winsock will provide an overlapped indication. If the error code returned by WSAGetLastError() is not WSA_IO_PENDING, then the overlapped operation has failed and no completion indication will materialize.

How does Winsock provide an indication that an overlapped I/O is complete? Winsock provides two methods to show an overlapped completion: event object signaling and completion routine. Both use an overlapped data structure, WSAOverlapped, which is an alias for the OVERLAPPED data structure that is found in the Win32 API. The data structure is associated with the overlapped operation. Let's consider these two methods in detail.

Event Notification

This method to implement overlapped I/O uses the event objects with WSAOVERLAPPED data structures. To map an event with the WSAOVERLAPPED structure, you should call WSACreateEvent() and assign that event to the overlapped data structure. When you call a function such as WSARecv() with a valid WSAOVERLAPPED structure, it returns immediately, usually with a SOCKET_ERROR. This is normal behavior, and your application must call WSAGetLastError() to check if the error is WSA_IO_PENDING, which means that the I/O is still in progress. However, if WSAGetLastError() reports a different error status, this could be due to a number of factors; one could be a problem with the WSAOVERLAPPED data structure. When your application gets an error other than WSA_IO_PENDING, your application should abort gracefully.

After calling WSARecv(), you will enter an infinite loop structure, in which you may call WSAWaitForMultipleEvents() (which we described earlier). The function waits for a set time for one or more event objects to become signaled. When this happens, you must call WSAGetOverlappedResult() to determine the status of the overlapped data structure associated with that event. The prototype of WSAGetOverlappedResult(), which is defined in Winsock2.pas, is as follows:

```
function WSAGetOverlappedResult(s: TSocket; lpOverlapped: LPWSAOVERLAPPED;
  var lpcbTransfer: DWORD; fWait: BOOL; lpdwFlags: DWORD): BOOL; stdcall;
```

The first parameter, *s*, is the socket that you specified for the WSAOVERLAPPED data structure, which in this case you used with WSARecv(). The second parameter, *lpOverlapped*, is a pointer to the WSAOVERLAPPED data structure associated with WSARecv(). The third parameter, *lpcbTransfer*, is a pointer to a variable containing the amount of data transferred in bytes during a send or receive operation. The fourth parameter, *fWait*, determines whether the function should wait for a pending overlapped I/O operation to complete. If *fWait* is TRUE, the function does not return until the I/O operation is complete. If *fWait* is FALSE, and the I/O operation is still pending, the function returns with the error of WSA_IO_INCOMPLETE. In the case of using event objects, this parameter has no relevance because when an event object is signaled, the overlapped I/O operation is complete. The last parameter, *lpdwFlags*, will receive resulting flags in calls to the WSARecv() or WSARecvFrom() functions.

When WSAGetOverlappedResult() returns TRUE, the call has succeeded, and the data pointed to by *lpcbTransfer* has been updated. If the function returns FALSE, this could indicate one of the following causes:

- Overlapped I/O is still pending.
- Overlapped I/O has completed with errors.

The completion status cannot be determined because of errors in one or more parameters that were supplied to WSAGetOverlappedResult().

When such an error occurs, your application must call WSAGetLastError() to determine the cause of the fatal error.

Completion I/O Schemes

We will now consider the second method of using an overlapped I/O scheme: the completion routine. Essentially, the second method uses a completion routine or callback function with a valid WSAOVERLAPPED data structure to handle overlapped I/O requests. The prototype for a completion routine is:

```
procedure CompletionRoutine(dwError, cbTransferred : DWORD; lpOverlapped : PWSAOVERLAPPED;
dwFlags : DWORD);
```

Whenever an overlapped I/O request completes, the parameters contain information regarding the completed overlapped I/O request. The first parameter, *dwError*, contains the completion status for the overlapped operation. If *dwError* is zero, this indicates a successful completion. Otherwise, if *dwError* is not zero, you should check the cause of the error by calling WSAGetLastError(). The second parameter, *cbTransferred*, indicates the number of bytes transferred for that overlapped I/O request. If *cbTransferred* is zero, this indicates an error condition, which you should check by calling WSAGetLastError(). The third parameter, *lpOverlapped*, is a pointer to the overlapped data structure that you used for the original call to an overlapped function, such as WSASend(), for that overlapped I/O request. The last parameter, *dwFlags*, is not relevant.

Because the current scenario does not use event objects to notify the application of a network event, the *hEvent* field of the overlapped data structure is not used. You should use WSAWaitForMultipleEvents() to wait for a network event to take place, but because this scenario does not use event objects, you have to create a dummy event object for use with the WSAWaitForMultipleEvents() function and set its *fWait* parameter to TRUE. When an overlapped request completes, the completion routine executes and WSAWaitForMultipleEvents() returns the constant WSA_IO_COMPLETION. At the same time, the completion routine posts another overlapped I/O request. This process continues until there are no more overlapped I/O requests. Listing 5-7 demonstrates this I/O scheme.

Completion Port I/O Scheme

This I/O scheme is the most difficult to implement, but it has considerable advantages over the I/O schemes that we have described up to now. The Completion Port I/O scheme scales well and offers the best performance. This scheme is only available on Windows NT 4.0, Windows 2000, and Windows XP, and it is the best possible scheme for servers that have to handle thousands of connections, such as a web server.

This scheme uses a Win32 completion port object that handles overlapped I/O requests using a supplied number of worker threads to service the overlapped requests.

To create a Win32 completion port object, you must call the CreateIoCompletionPort() function. Its prototype is as follows:

```
function CreateIoCompletionPort(FileHandle, ExistingCompletionPort: THandle;
  CompletionKey, NumberOfConcurrentThreads: DWORD): THandle; stdcall;
```

As well as creating the port object, the function returns a handle to the completion port object. The only parameter of interest is *NumberOfConcurrentThreads*, which you need to set. (Ignore the other parameters, as they are not required.) Setting this parameter sets the number of threads for each processor. To prevent needless context switching, you should set the number of threads to one per processor. By setting the parameter to zero, the function will allocate one thread per processor on the system, like this:

```
CompletionPortHandle :=  CreateIOCompletionPort(INVALID_HANDLE_VALUE, 0, 0, 0);
```

The following steps describe briefly the operation of a completion port object:

1. Create a completion port.
2. Determine the number of processors on the server.
3. Create worker threads to service completed I/O requests on the completion port.
4. Start a listening socket (call listen()) on a specified port.
5. Accept an incoming connection request using the accept() function.
6. Create a data structure to encapsulate the data for that client and save the accepted socket in the data structure.
7. Map the socket to the completion port object by calling CreateIoCompletionPort().

8. Start processing on the accepted socket using the overlapped I/O mechanism. For example, when an I/O request completes, a worker thread services the I/O requests.
9. Repeat Steps 5-8 until the server terminates.

We will not dive any further into the I/O Completion Port scheme, as it is a complex topic that deserves a chapter to itself (see Appendix C for more information), but Listing 5-5 will give some idea how to implement a simple I/O Completion Port scheme.

Which I/O Scheme to Use?

Table 5-7 shows the availability of these I/O schemes we have been discussing. After determining which I/O schemes are available on a platform, you need to consider which of the I/O schemes you could use. Certain I/O schemes are not appropriate for either a client or a server application. For example, you should not implement a WSAAsyncSelect() I/O scheme for a web server that handles hundreds, even thousands, of connections for performance reasons. Posting a message to a window handle for every occurrence of a network event for each connection incurs a heavy performance penalty. Simply put, it does not scale well. On the other hand, using WSAAsyncSelect() for a client application is a good move, but using WSAEventSelect() is even better for performance. Remember that WSAEventSelect() does not use window messages for network event handling.

Table 5-7: Availability of I/O schemes

	Win 9x	Win CE	NT 4	Win2000	NT 3.x	Win16	UNIX
Blocking	✓	✓	✓	✓	✓	✓	✓
Non-blocking	✓	✓	✓	✓	✓	✓	X
Asynchronous	✓	X	✓	X	✓	✓	X
Even Objects	✓	✓	✓	X	✓	X	X
Overlapped	✓*	✓	✓	✓	✓	X	X
Threads	✓	✓	✓	✓	✓	X	✓

* Although overlapped I/O is supported on Windows 95/98, it is not a true implementation.

We could discuss the pros and cons for each I/O scheme indefinitely, but a much better solution would be to base our choice on the main points in the following table:

Table 5-8: Scheme suitability for client and server implementation

I/O Scheme	Client	Server
Blocking	✓	✓
Non-blocking	✓	✓
Asynchronous	✓	✓**
Event Objects	✓	✓
Overlapped	✓	✓
Threads	✓	✓

** In some situations, servers can use the asynchronous scheme. For example, notable Internet components for Delphi that use the asynchronous scheme for their servers are Indy, Borland, and ICS.

Observe from Table 5-8 that we have not discussed two schemes yet—blocking and non-blocking socket I/O. We will discuss these in the next section.

If you were to develop a server application that handles thousands of concurrent connections, you would select an overlapped I/O scheme. If the server has more than one processor, you would select the I/O Completion Port scheme.

On the client side, you would use either of the following: WSAAsyncSelect(), WSAEventSelect(), and any of the overlapped I/O schemes. However, there is one caveat that you should be aware of when using an overlapped I/O scheme for a client: Windows 95/98 only supports a pseudo-implementation of overlapped I/O.

To Block or Not to Block?

Conceptually, from the coding point of view, using blocking sockets is easy to implement. However, when you use blocking sockets, the user interface freezes with considerable inconvenience to the user. One way to get around this freezing problem is to use a background thread to handle blocking sockets, thus leaving the user interface to work freely. The other approach is not to use a background thread but instead poll or loop with timeouts. If your application does not require interaction from the user, then using blocking sockets is a simple and straightforward method.

Non-blocking sockets, on the other hand, are much more difficult to handle and maintain. In terms of performance, they are inefficient, because your program has to perform polling on a continuous basis. You could make life easier for yourself by using the select() function to avoid the chore of polling. Although using the select() function certainly makes polling redundant, select() has a downside to it. The disadvantage is that it is inefficient, simply because your application has to service whole sets of sockets in every loop.

However, all is not lost, as the designers of Winsock incorporated an asynchronous version of select(), which is, of course, our old friend the WSAAsyncSelect() function. The designers of Winsock designed WSAAsyncSelect() to use the Windows messaging system to get around the problem of polling that using the select() function would entail. Using WSAEventSelect() is even easier than WSAAsyncSelect() to use because it does not require a window handle to operate.

We have been discussing select() in the GUI environment, but what about using select() in a console application? Putting aside the inefficiency aspect of select(), you could use select() in a server, as demonstrated in Listing 5-4 (program EX55).

If you decide that your application needs to use non-blocking sockets, how do you go about making sockets from a blocking mode to a non-blocking mode?

After creating your socket, simply call ioctlsocket() with the FIONBIO command, as shown in the following code snippet:

```
// Change the socket mode on the listening socket from blocking to
// non-blocking so the application will not block waiting for requests.
  NonBlock := 1;
  Res := ioctlsocket(sktListen, FIONBIO, NonBlock);
  if Res = SOCKET_ERROR then
  begin
    WriteLn(Format('Call to ioctlsocket() failed with error %s',
           [SysErrorMessage(WSAGetLastError)]));
    Exit;
  end;
```

Out-of-Band Data Etiquette

For reasons of portability and performance, we do not advise the use of out-of-band (OOB) data. For most applications, it is not necessary to use OOB data at all. However, some applications, such as Telnet and Rlogin, use OOB data. What is OOB data? It is data that an application can either send or receive bypassing the normal TCP stream. For example, the receiving application could send OOB data to tell the sending application to stop sending data.

Using OOB data to send urgent data is a rather risky strategy. If you really need to send urgent data, we would advise you to use a second socket to send or receive urgent data as the preferred solution. Alternatively, you could use UDP for the exchange of urgent or control data on a second socket to complement the activity of the first socket. As there is plenty of literature on the use of OOB data (see Appendix C for resources), we will not refer to OOB again in the rest of this tome.

Winsock and Multithreading

Any implementation of Winsock is thread safe, but only if you make it so. That is, your application needs to use threads sensibly, which can be achieved by synchronization. It is up to you to develop a multithreaded Winsock application that synchronizes Winsock calls. For example, avoid a situation that could turn nasty when your application fails to notify other threads when one thread closes a socket.

If you want to use multithreading in a Winsock application, consider two simple caveats:

- Don't use more than one thread to receive data on a socket because Winsock does not duplicate data among threads. In other words, if an application is using two threads to receive data on the same socket, data will not be duplicated, but instead the first thread will receive one set of data and the second thread will receive the next batch of data. This makes synchronization difficult.
- If your application uses threads to call WSAAsyncSelect() on a single socket, expect some trouble, as only the thread that made the last call to WSAAsyncSelect() will receive further notification. The other threads will continue to lurk in vain for a notification, since the notification for their WSAAsyncSelect() function has been overridden by the last thread's WSAAsyncSelect() function. This caveat also applies to threads calling WSAEventSelect() on the same socket.

To avoid those pitfalls listed above, you might want to consider using overlapped I/O, as we discussed earlier, because overlapped I/O schemes are essentially thread friendly. One such scheme is the I/O Completion Port (see Listing 5-5).

Listing 5-5: The I/O Completion Port scheme

```
program EX56;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows,
  WinSock2;

const
 MaxEchoes = 10;
 DataBuffSize = 8192;
 S = 'Hello';
 ECHO_PORT   = 9000;

type

 PPerIOOperationData = ^TPerIOOperationData;
```

```
 TPerIOOperationData = record
                      Overlapped: {WSA}TOverlapped;
                      Buffer : array[0..DataBuffSize - 1] of char;
                      DataBuffer : TWSABuf;
                      BytesSend,
                      BytesRecv : DWORD;
            end;

 PPerHandleData = ^TPerHandleData;
 TPerHandleData = record
                 skt : TSocket;
            end;

var
  WSAData: TWSAData;
  HostAddr: TSockAddrIn;
  sktListen,
  sktAccept: TSocket;
  ThrdHandle: THandle;
  Flags,
  RecvBytes,
  ThreadID: DWORD;
  i,
  Res: Integer;
  CompletionPort: THandle;
  PerIoData: PPerIOOperationData;
  PerHandleData: PPerHandleData;
  SystemInfo: TSystemInfo;
  CriticalSection: TRTLCriticalSection;

 function WorkerThread(lpCompletionPortID : Pointer) : DWORD; stdcall;
 var
   PerHandleData: PPerHandleData;
   PerIoData: PPerIOOperationData;
   BytesTransferred,
   SendBytes,
   RecvBytes,
   Flags: DWORD;
  begin
    EnterCriticalSection(CriticalSection);
    CompletionPort := THandle(lpCompletionPortID^);
    PerIoData := PPerIOOperationData(GlobalAlloc(GPTR,SizeOf(TPerIOOperationData)));
    PerHandleData := PPerHandleData(GlobalAlloc(GPTR,SizeOf(TPerHandleData)));
    while TRUE do
    begin
      if not GetQueuedCompletionStatus(CompletionPort, BytesTransferred,
                                 DWORD(PerHandleData){PerHandleData^.skt},
                                 POverlapped(PerIoData),INFINITE) then
      begin
        WriteLn(Format('Call to GetQueuedCompletionStatus() failed with error
                %d',[GetLastError]));
        Result := 0;
        Exit;
      end;
   // First check to see if an error has occurred on the socket and if so
   // then close the socket and cleanup the SOCKET_INFORMATION structure
   // associated with the socket.
      if BytesTransferred = 0 then
      begin
        WriteLn(Format('Closing socket %d', [PerHandleData^.skt]));
```

```
    if closesocket(PerHandleData^.skt) = SOCKET_ERROR then
    begin
      WriteLn(Format('Call to closesocket() failed with error %d', [WSAGetLastError]));
      Result := 0;
      Exit;
    end;
    GlobalFree(Cardinal(PerHandleData));
    GlobalFree(Cardinal(PerIoData));
    continue;
  end;

  // Check to see if the BytesRECV field equals zero. If this is so, then
  // this means a WSARecv call just completed so update the BytesRECV field
  // with the BytesTransferred value from the completed WSARecv() call.
  if PerIoData^.BytesRecv = 0 then
  begin
    PerIoData^.BytesRecv := BytesTransferred;
    PerIoData^.BytesSend := 0;
  end
  else
  begin
    PerIoData^.BytesSend := PerIoData^.BytesSend + BytesTransferred;
  end;
  if PerIoData^.BytesRecv > PerIoData^.BytesSend then
  begin
    // Post another WSASend() request.
    // Since WSASend() is not gauranteed to send all of the bytes requested,
    // continue posting WSASend() calls until all received bytes are sent.
    ZeroMemory(@PerIoData^.Overlapped, sizeof(TOVERLAPPED));
    PerIoData^.DataBuffer.buf := PerIoData^.Buffer + PerIoData^.BytesSEND;
    PerIoData^.DataBuffer.len := PerIoData^.BytesRecv - PerIoData^.BytesSend;
    if WSASend(PerHandleData^.skt, @PerIoData^.DataBuffer, 1, SendBytes, 0,
        @PerIoData^.Overlapped, NIL) = SOCKET_ERROR then
    begin
      if WSAGetLastError <> ERROR_IO_PENDING then
      begin
        WriteLn(Format('Call to WSASend() failed with error %d', [WSAGetLastError]));
        Result := 0;
        Exit;
      end
     end
  end
  else
  begin
    PerIoData^.BytesRecv := 0;
   // Now that there are no more bytes to send post another WSARecv() request.
    Flags := 0;
    ZeroMemory(@PerIoData^.Overlapped, sizeof(TOVERLAPPED));
    PerIoData^.DataBuffer.len := DataBuffSize;
    PerIoData^.DataBuffer.buf := PerIoData^.Buffer;
    if WSARecv(PerHandleData^.skt, @PerIoData^.DataBuffer, 1, RecvBytes, Flags,
        @PerIoData^.Overlapped, NIL) = SOCKET_ERROR then
    begin
      if WSAGetLastError <> ERROR_IO_PENDING then
      begin
         WriteLn(Format('Call to WSARecv() failed with error %d', [WSAGetLastError]));
         Result := 0;
         Exit
      end;
    end;
```

```
      end;
    end;
    LeaveCriticalSection(CriticalSection);
  end;

  procedure CleanUp(S : String);
  begin
    WriteLn('Call to ' + S + ' failed with error: ' + SysErrorMessage(WSAGetLastError));
    WSACleanUp;
    Halt;
  end;

begin
 if WSAStartUp($0202, WSAData) = 0 then
 try
   InitializeCriticalSection(CriticalSection);
// Set up I/O completion port ...
   CompletionPort := CreateIOCompletionPort(INVALID_HANDLE_VALUE, 0, 0,0);
   if CompletionPort = 0 then
   begin
     WriteLn(Format('Call to CreateIOCompletionPort() failed with error:
             %d',[GetLastError]));
     WSACleanUp;
     Exit;
   end;
// Determine how many processors on the system ...
   GetSystemInfo(SystemInfo);
// Create worker threads based on the number of processors.
// Create 2 worker threads for each processor ..
   for i := 0 to (SystemInfo.dwNumberOfProcessors * 2) - 1 do
   begin
   // Create a handle for a thread ...
     ThrdHandle := CreateThread(NIL, 0, @WorkerThread, @CompletionPort, 0, ThreadID);
     if ThrdHandle = 0 then
     begin
       WriteLn(Format('Call to CreateThread() failed with error: %d',[GetLastError]));
       WSACleanUp;
       Exit;
     end;
     CloseHandle(ThrdHandle);
   end;
// Create a listening socket ...
   sktListen := WSASocket(AF_INET, SOCK_STREAM, 0, NIL, 0, WSA_FLAG_OVERLAPPED);
   if sktListen = INVALID_SOCKET then
    CleanUp('WSASocket()');
   HostAddr.sin_family := AF_INET;
   HostAddr.sin_port := htons(ECHO_PORT);
   HostAddr.sin_addr.S_addr := htonl(INADDR_ANY);
   Res := bind(sktListen, @HostAddr, SizeOf(HostAddr));
   if Res = SOCKET_ERROR then
    CleanUp('bind()');
// Prepare the socket for listening ...
   Res := listen(sktListen,5);
   if Res = SOCKET_ERROR then
    CleanUp('listen()');
// Enter a while loop to accept connections and assign to the completion port ...
   while TRUE do
   begin
     sktAccept :=  WSAAccept(sktListen, NIL, NIL, NIL, 0);
     if sktAccept = SOCKET_ERROR then
```

```
      begin
        WriteLn(Format('Call to WSAAccept() failed with error %d', [WSAGetLastError]));
        closesocket(sktListen);
        WSACleanUp;
        Exit;
      end;
// Create a socket information structure to associate with the socket
      PerHandleData := PPerHandleData(GlobalAlloc(GPTR,SizeOf(TPerHandleData)));
      if PerHandleData = NIL then
      begin
        WriteLn(Format('Call to GlobalAlloc() failed with error %d', [GetLastError]));
        closesocket(sktListen);
        closesocket(sktAccept);
        WSACleanUp;
        Exit;
      end;
// Associate the accepted socket with the original completion port.
      WriteLn(Format('Success! Socket number %d connected', [sktAccept]));
      PerHandleData^.skt := sktAccept;
      Res := CreateIoCompletionPort(THANDLE(sktAccept), CompletionPort,
             DWORD(PerHandleData),0);
      if Res = 0 then
      begin
        WriteLn(Format('Call to CreateIoCompletionPort failed with error %d',
               [GetLastError]));
        closesocket(sktListen);
        closesocket(sktAccept);
        WSACleanUp;
        Exit;
      end;
// Create per I/O socket information structure to associate with the
// WSARecv call below.
      PerIoData := PPerioOperationData(GlobalAlloc(GPTR, sizeof(TPERIOOPERATIONDATA)));
      if PerIoData = NIL then
      begin
        WriteLn(Format('Call to GlobalAlloc() failed with error %d', [WSAGetLastError]));
        closesocket(sktListen);
        closesocket(sktAccept);
        WSACleanUp;
        Exit;
      end;
      ZeroMemory(@PerIoData^.Overlapped, sizeof(TOVERLAPPED));
      PerIoData^.BytesSend := 0;
      PerIoData^.BytesRecv := 0;
      PerIoData^.DataBuffer.len := DataBuffSize;
      PerIoData^.DataBuffer.buf := PerIoData^.Buffer;
      Flags := 0;
      Res := WSARecv(sktAccept, @PerIoData^.DataBuffer, 1, RecvBytes, Flags,
         @PerIoData^.Overlapped, NIL);
      if Res = SOCKET_ERROR then
      begin
        if WSAGetLastError <> ERROR_IO_PENDING then
        begin
          WriteLn(Format('Call to WSARecv() failed with error %d', [WSAGetLastError]));
          closesocket(sktListen);
          closesocket(sktAccept);
          WSACleanUp;
          Exit;
        end;
      end;
```

```
    end;
    closesocket(sktListen);
  finally
   WSACleanUp;
  end
  else WriteLn('Failed to load Winsock...');
end.
```

function select Winsock2.pas

Syntax

select(nfds: Integer; readfds, writefds, exceptfds: PFdSet; timeout: PTimeVal): Integer; stdcall;

Description

This function determines the status of one or more sockets.

Parameters

nfds: This argument is ignored and included only for the sake of compatibility.

readfds: An optional pointer to a set of sockets to be checked for reading

writefds: An optional pointer to a set of sockets to be checked for writing

exceptfds: An optional pointer to a set of sockets to be checked for errors

timeout: The maximum time for select() to wait, or NIL for blocking operation

Return Value

If the function succeeds, it will return the number of descriptors that are ready. The function will return zero if the time limit has expired. If the connection has been closed gracefully and all data received, the return value will be zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAEFAULT, WSAENETDOWN, WSAEINVAL, WSAEINTR, WSAEINPROGRESS, and WSAENOTSOCK.

See Appendix B for a detailed description of the error codes.

See Also

accept, connect, recv, recvfrom, send, WSAAsyncSelect, WSAEventSelect

Example

See Listing 5-4 (program EX55).

function WSAAsyncSelect Winsock2.pas

Syntax

WSAAsyncSelect(s: TSocket; hWnd: HWND; wMsg: u_int; lEvent: Longint): Integer; stdcall;

Description

This function requests a Windows message-based notification of network events for a socket.

Parameters

s: A descriptor identifying the socket for which event notification is required

hWnd: A handle identifying the window that should receive a message when a network event occurs

wMsg: The message to be received when a network event occurs

lEvent: A bit mask that specifies a combination of network events in which the application is interested

Return Value

If the function succeeds, it will return zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are for the following events:

FD_CONNECT: WSAEAFNOSUPPORT, WSAECONNREFUSED, WSAENETUNREACH, WSAEFAULT, WSAEUINVAL, WSAEISCONN, WSAEMFILE, WSAENOBUFS, WSAENOTCONN, and WSAETIMEDOUT

FD_CLOSE: WSAENETDOWN, WSAECONNRESET, and WSAECONNABORTED

FD_READ, FD_WRITE, FD_OOB, FD_ACCEPT, FD_QOS, FD_GROUP_QOS, and FD_ADDRESS_LIST_CHANGE: WSAENETDOWN

FD_ROUTING_INTERFACE_CHANGE: WSAENETUNREACH and WSAENETDOWN

See Appendix B for a detailed description of the error codes.

See Also

select, WSAEventSelect

Example

See Listing 5-6 (program EX58).

Listing 5-6: An asynchronous echo server that uses two different protocols, TCP and UDP

```
program EX58
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, Buttons,
  Winsock2;

const
```

```
 SOCK_EVENT = WM_USER + 500;
 NULL : Char   = #0;

 CRLF : array[0..2] of char = #13#10#0;

 MaxBufferSize  = MAXGETHOSTSTRUCT;

 ECHO_PORT   = 9000;

type

 TCharArray    = array[0..MAXGETHOSTSTRUCT - 1] of char;

 TConditions  = (Success, Failure, None);

 TTransport   = (TCP, UDP);



  TfrmMain = class(TForm)
    gbStatusMsg: TGroupBox;
    memStatusMsg: TMemo;
    gbOptions: TGroupBox;
    gbPortNo: TGroupBox;
    edPortNo: TEdit;
    pnButtons: TPanel;
    btnStart: TBitBtn;
    btnStop: TBitBtn;
    btnClose: TBitBtn;
    rgbTransportProtocol: TRadioGroup;
    gbWSVersion: TGroupBox;
    edWSVersion: TEdit;
    procedure btnStartClick(Sender: TObject);
    procedure btnStopClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure rgTransportClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
   FStatus            : TConditions;
   FMsg               : String;
   wsaData            : TWSADATA;
   FEchoPortNo,
   FSocketNo,
   FSkt               : TSocket;
   FWnd               : HWND;
   FCount             : Integer;
   FProtocol          : PProtoEnt;
   FService           : PServent;
   FSockAddrIn        : TSockAddrIn;
   FTransport         : TTransport;
   FRC                : Integer;
   FMsgBuff           : TCharArray;
   WSRunning: Boolean;
   procedure GetServer;
   procedure EchoEvent(var Mess : TMessage); message SOCK_EVENT;
   procedure Start;
```

5 Chapter

```
    procedure Stop;
    function  GetDatagram : TCharArray;
    procedure SetDatagram(DataReqd : TCharArray);
  end;

var
  frmMain: TfrmMain;

implementation
{$R *.DFM}

procedure TfrmMain.btnStartClick(Sender: TObject);
begin
 WSRunning := WSAStartUp($0202, wsaData) = 0;
 if not WSRunning then
 begin
  btnStart.Enabled := FALSE;
  btnStop.Enabled  := FALSE;
  memStatusMsg.Lines.Add('Cannot load Winsock ' + edWSVersion.Text);
  Exit;
 end;
 case rgbTransportProtocol.ItemIndex of
  0 : FTransport   := TCP;
  1 : FTransport   := UDP;
 end;// case
  FWnd             := AllocateHwnd(EchoEvent);
  btnStart.Enabled := FALSE;
  btnStop.Enabled  := TRUE;
  Start;
end;

procedure TfrmMain.btnStopClick(Sender: TObject);
begin
 btnStart.Enabled  := TRUE;
 btnStop.Enabled   := FALSE;
 Stop;
end;

procedure TfrmMain.FormCreate(Sender: TObject);
begin
 btnStop.Enabled := FALSE;
 memStatusMsg.Clear;
end;

procedure TfrmMain.btnCloseClick(Sender: TObject);
begin
 Close;
end;

procedure TfrmMain.rgTransportClick(Sender: TObject);
begin
 if rgbTransportProtocol.ItemIndex = 0 then
  FTransport := TCP
 else
  FTransport := UDP;
end;

procedure TfrmMain.GetServer;
begin
 // Create a socket
```

```
 case FTransport of
  UDP : FSocketNo := socket(AF_INET, SOCK_DGRAM,  IPPROTO_IP);
  TCP : FSocketNo := socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 end;
 if FSocketNo <> INVALID_SOCKET then
 begin
  FMsg := Concat('Socket ',IntToStr(FSocketNo),' created...');
  memStatusMsg.Lines.Add(FMsg);
  FMsg := '';
  case FTransport of
   UDP : FProtocol := getprotobyname('udp');
   TCP : FProtocol := getprotobyname('tcp');
  end;
  if FProtocol <> NIL then
  begin
   if FTransport = UDP then
    Fmsg := Concat('udp',' protocol present...')
   else
    Fmsg := Concat('tcp',' protocol present...');
   memStatusMsg.Lines.Add(FMsg);
   FMsg := '';
   case FTransport of
    UDP :FService := getservbyname('echo','udp');
    TCP :FService := getservbyname('echo','tcp');
   end;
   if FService <> NIL then
   begin
    Fmsg := Concat('echo',' service present...');
    memStatusMsg.Lines.Add(FMsg);
    FMsg := '';
    FStatus := Success;
    FSockAddrIn.sin_family           := AF_INET;
    FSockAddrIn.sin_port             := FService^.s_port;
    FSockAddrIn.sin_addr.s_addr      := htonl(INADDR_ANY);
   end else
   begin
    FStatus := Failure;
    Fmsg := Concat('Failure: ',SysErrorMessage(WSAGetLastError));
    memStatusMsg.Lines.Add(FMsg);
    FMsg := '';
    Exit;
   end;
  end;
 end;
end;

procedure TfrmMain.EchoEvent(var Mess : TMessage);
var
 MsgStr: TCharArray;
begin
 case WSAGetSelectEvent(Mess.LParam) of
  FD_ACCEPT  : begin
                FMsg := 'Accepting ...';
                memStatusMsg.Lines.Add(Fmsg);
               end;
  FD_READ    : begin
               inc(FCount);
               frmMain.memStatusMsg.Lines.Add('FD_READ ' + IntToStr(FCount));
// process the message ...
```

```
                FMsg := Concat('Message ', FMsg, ' received from
                               ',StrPas(inet_ntoa(FSockAddrIn.sin_addr)));
// send the message back ...
                memStatusMsg.Lines.Add(FMsg);
                FMsg := '';
                MsgStr := GetDatagram;
                memStatusMsg.Lines.Add(MsgStr);
                SetDatagram(MsgStr);
                end;
  FD_WRITE   : begin
//                 memStatusmsg.Lines.Add('FD_WRITE...');
                end;
   end;
end;

procedure TfrmMain.Stop;
begin
 if FTransport = TCP then
 begin
  shutdown(FSkt,1);
  closesocket(Fskt);
 end else
 begin
  shutdown(FSocketNo,1);
  CloseSocket(FSocketNo);
 end;
 WSACleanUp;
end;

procedure TfrmMain.Start;
var
 AddrSize,
 Res: Integer;
 ServerAddr: TSockAddrIn;
begin
 GetServer;
 if FStatus <> Success then
 begin
  Exit;
 end;
 if bind(FSocketNo, @FSockAddrIn, SizeOf(TSockAddrIn)) = Integer(SOCKET_ERROR) then
 begin
  FMsg := Concat('Failed to bind : ', SysErrorMessage(WSAGetLastError));
  memStatusMsg.Lines.Add(FMsg);
  memStatusMsg.Lines.Add('If this happens, this is likely to be caused by an echo server
                         already running on your machine.');
  memStatusMsg.Lines.Add('To cure this problem, you must abort the service before you run
                         this server.');
  FMsg := '';
  FStatus := Failure;
  Exit;
 end;
{Now to determine port no. This should be in SockInfo }
 AddrSize := SizeOf(TSockAddrIn);
 if getsockname(FSocketNo, @ServerAddr, AddrSize) = SOCKET_ERROR then
 begin
  FMsg := Concat('Failed to get port : ',SysErrorMessage(WSAGetLastError));
  memStatusMsg.Lines.Add(FMsg);
  FMsg := '';
  FStatus := Failure;
```

```
  Exit;
 end else
 begin{success!}
  FEchoPortNo := ntohs(ServerAddr.sin_port);
  FMsg := Concat('Successful. Now listening on port ', IntToStr(FEchoPortNo));
  memStatusMsg.Lines.Add(FMsg);
  FMsg := '';
  FCount := 0;
 end;
 if FTransport = UDP then
 begin
  if WSAAsyncSelect(FSocketNo, FWnd, SOCK_EVENT,  FD_READ or FD_WRITE or FD_CONNECT
                    or FD_CLOSE)
                    = SOCKET_ERROR then {handle}
  begin
   FMsg := Concat('Error : ',SysErrorMessage(WSAGetLastError));
   memStatusMsg.Lines.Add(FMsg);
   FMsg := '';
   FStatus := Failure;
   Exit;
  end;
 end;
 if FTransport = TCP then
 begin
// listen ...
  Res := listen(FSocketNo,5);
  if Res = SOCKET_ERROR then
  begin
   memStatusMsg.Lines.Add(SysErrorMessage(WSAGetLastError));
   Exit;
  end;
  AddrSize := SizeOf(FSockAddrIn);
  Fskt := accept(FSocketNo, @FSockAddrIn, @AddrSize);
  if WSAAsyncSelect(FSkt, FWnd, SOCK_EVENT,  FD_READ or FD_WRITE or FD_CONNECT
                    or FD_CLOSE)
                    = SOCKET_ERROR then {handle}
  begin
   FMsg := Concat('Error : ',SysErrorMessage(WSAGetLastError));
   memStatusMsg.Lines.Add(FMsg);
   FMsg := '';
   FStatus := Failure;
   Exit;
  end;
 end;
end;

function  TfrmMain.GetDatagram : TCharArray;
var
 Size,
 Response: Integer;
begin
 Size := SizeOf(TSockAddrIn);
 Response := 0;
 case FTransport of
  UDP : Response := recvfrom(FSocketNo, FMsgBuff, SizeOf(FMsgBuff), 0,
                            @FSockAddrIn, Size);
  TCP : Response := recv(FSkt, FMsgBuff, SizeOf(FMsgBuff),0);
 end;
 if Response = SOCKET_ERROR then
 begin { Error receiving data from remote host }
```

```
  if WSAGetLastError <> WSAEWOULDBLOCK then{this is a real error!}
  begin
   FStatus := Failure;
   FMsg := Concat('Error reading data : ',SysErrorMessage(WSAGetLastError));
   Result := '';
   memStatusMsg.Lines.Add(Fmsg);
   FMsg := '';
   Exit;
  end
 end;
 Result := FMsgBuff;
end;

procedure TfrmMain.SetDatagram(DataReqd : TCharArray);
var
 Response: Integer;
begin
 Response := 0;
 case FTransport of
  UDP : Response := sendto(FSocketNo, DataReqd, SizeOf(DataReqd), MSG_DONTROUTE,
                          @FSockAddrIn, SizeOf(TSockAddrIn));
  TCP : Response := send(Fskt, DataReqd, SizeOf(DataReqd), 0);
 end;
 if Response = SOCKET_ERROR then
 begin { Error sending data to remote host }
  if WSAGetLastError <> WSAEWOULDBLOCK then{this is a real error!}
  begin
   FMsg := SysErrorMessage(WSAGetLastError);
   memStatusMsg.Lines.Add(FMsg);
   FStatus := Failure;
   Exit;
  end
 end else
 begin
  FStatus := Success;
 end;
end;

procedure TfrmMain.FormDestroy(Sender: TObject);
begin
 DeallocateHwnd(FWnd);
 if WSACleanUp = SOCKET_ERROR then // this should not happen!
 begin
  MessageDlg('Failed to close down WinSock!', mtError,
            [mbOk], 0)
 end;
end;

end.
```

function WSACreateEvent Winsock2.pas

Syntax

WSACreateEvent: WSAEVENT; stdcall;

Description

This function creates a new event object whose initial state is non-signaled.

Parameters

None

Return Value

If the function succeeds, it will return the handle of the new event object. Otherwise, it will return the value WSA_INVALID_HANDLE. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSA_NOT_ENOUGH_MEMORY.

See Appendix B for a detailed description of the error codes.

See Also

WSACloseEvent, WSAEnumNetworkEvents, WSAEventSelect, WSAGetOverlappedResult, WSARecv, WSARecvFrom, WSAResetEvent, WSASend, WSASendTo, WSASetEvent, WSAWaitForMultipleEvents

Example

See Listing 5-7 (program EX52).

Listing 5-7: A generic echo server that uses overlapped I/O with a callback routine

```
program EX52;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows,
  WinSock2;

const
 MaxEchoes = 10;
 DataBuffSize = 8192;
 S = 'Hello';

type

 PSocketInfo = ^TSocketInfo;
 TSocketInfo = record
               Overlapped : TOverlapped;// WSAOverlapped
               skt : TSocket;
               Buffer : array[0..DataBuffSize - 1] of char;
               DataBuffer : TWSABuf;
               BytesSend,
               BytesRecv : DWORD;
  end;

var
  WSAData: TWSAData;
  DummyAddr: PSockAddrIn;
  HostAddr: TSockAddrIn;
  sktListen,
  sktAccept: TSocket;
  Size: PInteger;
```

```
  ThrdHandle: THandle;
  ThreadID: DWORD;
  AcceptEvent: WSAEvent;
  Res: Integer;

// Callback routine ...
 procedure Worker(Error, BytesTransferred : DWORD; Overlapped : PWSAOverlapped; InFlags :
                  DWORD); stdcall;
 var
  SendBytes,
  Recvbytes,
  Flags: DWORD;
  Sktinfo : PSocketInfo;
 begin
// Typecast the WSAOverlapped structure as a TSocketInfo structure ...
    sktInfo := PSocketInfo(Overlapped);
    if Error <> 0 then
      WriteLn(Format('I/O operation failed with error %d',[Error]));
    if BytesTransferred = 0 then
      WriteLn(Format('Closing socket %d',[sktInfo^.skt]));
     if (Error <> 0) or (BytesTransferred = 0) then
     begin
      closesocket(SktInfo^.skt);
      GlobalFree(Cardinal(Sktinfo));
      Exit;
    end;
// Check to see if the BytesRecv = 0. If this is so,
    if SktInfo^.BytesRecv = 0 then
    begin
     SktInfo^.BytesRecv := BytesTransferred;
     SktInfo^.BytesSend := 0;
    end
    else
    begin
      SktInfo^.BytesSend := sktInfo^.BytesSend + BytesTransferred;
    end;
    if SktInfo^.BytesRecv > SktInfo^.BytesSend then
    begin
// Post another WSASend() request ...
     ZeroMemory(@SktInfo^.Overlapped, SizeOf(TOverlapped));
     SktInfo^.DataBuffer.buf := SktInfo^.Buffer + Sktinfo^.BytesSend;
     SktInfo^.DataBuffer.len := SktInfo^.BytesRecv - SktInfo^.BytesSend;
     Res := WSASend(SktInfo^.skt, @SktInfo^.DataBuffer, 1, SendBytes, 0,
                    @SktInfo^.Overlapped, @Worker);
     if Res = SOCKET_ERROR then
      if WSAGetLastError <> WSA_IO_PENDING then
      begin
       WriteLn(Format('Call to WSASend() failed with error:
                      %s',[SysErrorMessage(WSAGetLastError)]));
       Exit;
      end;
    end
    else
    begin
      Sktinfo^.BytesRecv := 0;
// No more bytes to send so stop calling WSASend(), so
// post another WSARecv() request ...
      Flags := 0;
      ZeroMemory(@SktInfo^.Overlapped, SizeOf(TOverlapped));
      SktInfo^.DataBuffer.len := DataBuffSize;
```

```
      SktInfo^.DataBuffer.buf := SktInfo^.Buffer;
      Res := WSARecv(SktInfo^.skt, @SktInfo^.DataBuffer, 1, RecvBytes,
                     Flags, @SktInfo^.Overlapped, @Worker);
      if Res = SOCKET_ERROR then
       if WSAGetlastError <> WSA_IO_PENDING then
       begin
         WriteLn(Format('Call to WSARecv() failed with error:
                        %s',[SysErrorMessage(WSAGetLastError)]));
         Exit;
       end;
    end;
 end;

 function WorkerThread(lpParameter : Pointer) : DWORD; stdcall;
 var
  Flags,
  Index,
  RecvBytes: DWORD;
  SktInfo: PSocketInfo;
  EventArray : array[0..0] of WSAEvent;
 begin
// save the Accept event in the array ...
  EventArray[0] := WSAEvent(lpParameter^);
  Index := 0;
   while TRUE do
   begin
     Index := WSAWaitForMultipleEvents(1, @EventArray, FALSE, WSA_INFINITE, TRUE);
     if Index = WSA_WAIT_FAILED then
     begin
       WriteLn('call to WSARecv() failed with error: ' + SysErrorMessage(WSAGetLastError));
       Result := 0;
       Exit;
     end;
     if Index <> WAIT_IO_COMPLETION then
      break;// we have an accept() call already, so break out of the loop ...
   end;// while
   WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
// Now create a socket information structure to associate with the accepted socket ...
   SktInfo := PSocketInfo(GlobalAlloc(GPTR, SizeOf(TSocketInfo)));
   if Sktinfo = NIL then
   begin
    WriteLn('Call to GlobalAlloc() failed with error: ' + SysErrorMessage(GetLastError));
    Result := 0;
    Exit;
   end;
// Populate the SktInfo structure ...
   Sktinfo.skt := sktAccept;
   ZeroMemory(@SktInfo^.Overlapped, SizeOf(TOverlapped));
   SktInfo^.BytesSend := 0;
   sktInfo^.BytesRecv := 0;
   sktInfo^.DataBuffer.len := DataBuffSize;
   SktInfo^.DataBuffer.buf := SktInfo^.Buffer;
   Flags := 0;
   Res := WSARecv(SktInfo^.skt, @SktInfo^.DataBuffer, 1, RecvBytes, Flags,
                  @SktInfo^.Overlapped, @Worker);
   if Res = SOCKET_ERROR then
    if WSAGetLastError <> WSA_IO_PENDING then
    begin
      WriteLn('Call to WSARecv() failed with error: ' + SysErrorMessage(WSAGetLastError));
      Result := 0;
```

```
      Exit;
    end;
// Success, there is a connection ...
    WriteLn(Format('Socket %d connected...',[sktAccept]));
  // JCP 080202 end;
 end;

  procedure CleanUp(S : String);
  begin
    WriteLn('Call to ' + S + ' failed with error: ' + SysErrorMessage(WSAGetLastError));
    WSACleanUp;
    Halt;
  end;

begin
 if WSAStartUp($0202, WSAData) = 0 then
 try
  sktListen := WSASocket(AF_INET, SOCK_STREAM, 0, NIL, 0, WSA_FLAG_OVERLAPPED);
  if sktListen = INVALID_SOCKET then
   CleanUp('WSASocket()');
  HostAddr.sin_family := AF_INET;
  HostAddr.sin_port := htons(IPPORT_ECHO);
  HostAddr.sin_addr.S_addr := htonl(INADDR_ANY);
  Res := bind(sktListen, @HostAddr, SizeOf(HostAddr));
  if Res = SOCKET_ERROR then
   CleanUp('bind()');
  Res := listen(sktListen,5);
  if Res = SOCKET_ERROR then
   CleanUp('listen()');
// Create an event object ...
  AcceptEvent := WSACreateEvent;
  if AcceptEvent = WSA_INVALID_EVENT then
   CleanUp('WSACreateEvent()');
// Create a worker thread to servuce completed I/O requests ...
  ThrdHandle := CreateThread(NIL, 0, @WorkerThread, @AcceptEvent, 0, ThreadID);
  if ThrdHandle = 0 then
  begin
   WriteLn('call to CreateThread failed with error: '+ SysErrorMessage(GetLastError));
   closesocket(sktListen);
   WSACleanUp;
   halt;
  end;
  DummyAddr:= AllocMem(SizeOf(TSockAddrIn));
  try
   DummyAddr.sin_family := AF_INET;
   DummyAddr.sin_port := htons(IPPORT_ECHO);
   DummyAddr.sin_addr.S_addr := INADDR_ANY;
   Size:= AllocMem(SizeOf(TSockAddrIn));
   try
    Size^ := SizeOf(DummyAddr);
// Enter an infinite loop ...
    while TRUE do
    begin
      sktAccept := accept(sktListen, @DummyAddr, Size);
      if not WSASetEvent(AcceptEvent) then
       CleanUp('accept()');
    end;
   finally
     Freemem(Size);
   end;
```

```
    finally
      Freemem(DummyAddr);
    end;
   finally
    WSACleanUp;
   end
   else WriteLn('Failed to load Winsock...');
 end.
```

function WSAWaitForMultipleEvents Winsock2.pas

Syntax

WSAWaitForMultipleEvents(cEvents: DWORD; lphEvents: PWSAEVENT; fWaitAll: BOOL; dwTimeout: DWORD; fAlertable: BOOL): DWORD; stdcall;

Description

This function returns when one or all of the specified event objects are in the signaled state or when the timeout interval specified by *dwTimeout* expires.

Parameters

cEvents: Specifies the number of event object handles in the array pointed to by *lphEvents*. The maximum number of event object handles is WSA_MAXIMUM_WAIT_EVENTS. One or more events must be specified.

lphEvents: Points to an array of event object handles

fWaitAll: Specifies the wait type. If *fWaitAll* is TRUE, the function returns when all event objects in the *lphEvents* array are signaled at the same time. If FALSE, the function returns when any one of the event objects is signaled. In the latter case, the return value indicates the event object whose state caused the function to return.

dwTimeout: Specifies the timeout interval, in milliseconds. The function returns if the interval expires, even if conditions specified by the *fWaitAll* parameter are not satisfied. If *dwTimeout* is zero, the function tests the state of the specified event objects and returns immediately. If *dwTimeout* is WSA_INFINITE, the function's timeout interval never expires.

fAlertable: Specifies whether the function returns when the system queues an I/O completion routine for execution by the calling thread. If *fAlertable* is TRUE, the completion routine is executed and the function returns. If FALSE, the completion routine is not executed when the function returns.

Return Value

If the function fails, the return value will be WSA_WAIT_FAILED. To obtain extended error information, call WSAGetLastError(). The return value upon success is one of the values in Table 5-9.

Table 5-9: Return values for WSAWaitForMultipleEvents()

Value	Meaning
WSA_WAIT_EVENT_0 to (WSA_WAIT_EVENT_0 + cEvents – 1)	If fWaitAll is TRUE, the return value indicates that the state of all specified event objects is signaled. If fWaitAll is FALSE, the return value minus WSA_WAIT_EVENT_0 indicates the lphEvents array index of the object that satisfied the wait.
WAIT_IO_COMPLETION	One or more I/O completion routines are queued for execution.
WSA_WAIT_TIMEOUT	The timeout interval elapsed and the conditions specified by the fWaitAll parameter are not satisfied.
WSANOTINITIALISED	A successful WSAStartup must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock 1.1 call is in progress or the service provider is still processing a callback function.
WSA_NOT_ENOUGH_MEMORY	There is not enough free memory available to complete the operation.
WSA_INVALID_HANDLE	One or more of the values in the lphEvents array is not a valid event object handle.
WSA_INVALID_PARAMETER	The cEvents parameter does not contain a valid handle count.

See Also

WSACloseEvent, WSACreateEvent

Example

See Listing 5-8 (program EX57).

Listing 5-8: A generic echo server that uses the WSAEventSelect() model

```
program EX57;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows,
  WinSock2;

const
 MaxEchoes = 10;
 DataBuffSize = 8192;
 S = 'Hello';
 EventTotal: DWORD = 0;
type

 PSocketInfo = ^TSocketInfo;
 TSocketInfo = record
              Buffer: array[0..DataBuffSize - 1] of char;
              DataBuffer: WSABuf;
              skt: TSocket;
              BytesSend,
              BytesRecv : DWORD;
              end;

var
  WSAData: TWSAData;
```

```
  EventArray: array[0..WSA_MAXIMUM_WAIT_EVENTS - 1] of WSAEVENT;
  SocketArray: array[0..WSA_MAXIMUM_WAIT_EVENTS - 1] of PSocketInfo;
  HostAddr: TSockAddrIn;
  sktListen,
  sktAccept: TSocket;
  Flags,
  RecvBytes,
  SendBytes: DWORD;
  Event: WSAEvent;
  Res: Integer;
  NetworkEvents: WSANETWORKEVENTS;
  SocketInfo : PSocketInfo;

  function CreateSocketInfo(skt: TSocket) : BOOLEAN;
  var
   SI: PSocketInfo;
  begin
    EventArray[EventTotal] := WSACreateEvent;
    if EventArray[EventTotal] = WSA_INVALID_EVENT then
    begin
      WriteLn(Format('Call to WSACreateEvent() failed with error %d',[WSAGetLastError]));
      Result := FALSE;
      Exit;
    end;
    SI := PSocketInfo(GlobalAlloc(GPTR, SizeOf(TSocketInfo)));
    if SI = NIL then
    begin
      WriteLn(Format('Call to GlobalAlloc() failed with error %d',[GetLastError]));
      Result := FALSE;
      Exit;
    end;
// Now prepare the TSocketInfo record for use ...
    SI^.skt := skt;
    SI^.BytesSend := 0;
    SI^.BytesRecv := 0;
    SocketArray[EventTotal] := Si;
    inc(EventTotal);
    Result := TRUE;
  end;

  procedure FreeSocketInfo(Event: DWORD);
  var
    SI: PSocketInfo;
    i: DWORD;
  begin
    SI := SocketArray[Event];
    closesocket(SI^.skt);
    GlobalFree(Cardinal(SI));
    WSACloseEvent(EventArray[Event]);
// Close up some space ...
    for i := Event {- 1} to EventTotal {- 1} do
    begin
      EventArray[i] := EventArray[i+1];
      SocketArray[i] := SocketArray[i+1];
    end;
    dec(EventTotal);
  end;

  procedure CleanUp(S : String);
  begin
```

```
     WriteLn('Call to ' + S + ' failed with error: ' + SysErrorMessage(WSAGetLastError));
     WSACleanUp;
     Halt;
   end;

begin
 if WSAStartUp($0202, WSAData) = 0 then
 try
// Create a listening socket ...
   sktListen := WSASocket(AF_INET, SOCK_STREAM, 0, NIL, 0, WSA_FLAG_OVERLAPPED);
   if sktListen = INVALID_SOCKET then
    CleanUp('WSASocket()');
   CreateSocketInfo(sktListen);
   Res := WSAEventSelect(sktListen,EventArray[EventTotal - 1],FD_ACCEPT or FD_CLOSE);
   if Res = SOCKET_ERROR then
    CleanUp('WSAEventSelect()');
   HostAddr.sin_family := AF_INET;
   HostAddr.sin_port := htons(IPPORT_ECHO);
   HostAddr.sin_addr.S_addr := htonl(INADDR_ANY);
   Res := bind(sktListen, @HostAddr, SizeOf(HostAddr));
   if Res = SOCKET_ERROR then
    CleanUp('bind()');
// Prepare the socket for listening ...
   Res := listen(sktListen,5);
   if Res = SOCKET_ERROR then
    CleanUp('listen()');
// Enter a while loop to accept connections ...
    while TRUE do
    begin
      Event := WSAWaitForMultipleEvents(EventTotal,@EventArray,FALSE,WSA_INFINITE,FALSE);
      if Event = WSA_WAIT_FAILED then
        CleanUp('WSAWaitForMultipleEvents()');
      Res := WSAEnumNetworkEvents(SocketArray[Event - WSA_WAIT_EVENT_0]^.skt,EventArray
                                        [Event - WSA_WAIT_EVENT_0], @NetworkEvents);
      if Res = SOCKET_ERROR then
         CleanUp('WSAEnumNetworkEvents()');
      if (NetworkEvents.lNetworkEvents and FD_ACCEPT) = FD_ACCEPT then
      begin
        if NetworkEvents.iErrorCode[FD_ACCEPT_BIT] <> 0 then
        begin
          WriteLn(Format('FD_ACCEPT failed with error %d',
                         [NetworkEvents.iErrorCode[FD_ACCEPT_BIT]]));
          break;
        end;
        sktAccept := WSAaccept(SocketArray[Event - WSA_WAIT_EVENT_0]^.skt, NIL, NIL,NIL,0);
        if sktAccept = INVALID_SOCKET then
        begin
          WriteLn(Format('Call to accept() failed with error %d', [WSAGetLastError]));
          break;
        end;
         if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS) then
         begin
           WriteLn('Too many connections - closing socket.');
           closesocket(sktAccept);
           break;
         end;
         CreateSocketInfo(sktAccept);
         if WSAEventSelect(sktAccept, EventArray[EventTotal - 1], FD_READ or FD_WRITE or
                          FD_CLOSE) = SOCKET_ERROR then
         begin
```

```
      WriteLn(Format('WSAEventSelect() failed with error %d', [WSAGetLastError]));
      Exit;
    end;
    WriteLn(Format('Socket %d connected', [sktAccept]));
  end;
  // Try to read and write data to and from the data buffer if read and write events
    occur.
  if (NetworkEvents.lNetworkEvents and FD_READ = FD_READ) or
    (NetworkEvents.lNetworkEvents and FD_WRITE = FD_WRITE) then
  begin
    if (NetworkEvents.lNetworkEvents and FD_READ = FD_READ) and
      (NetworkEvents.iErrorCode[FD_READ_BIT] <> 0) then
    begin
      WriteLn(Format('FD_READ failed with error %d',
             [NetworkEvents.iErrorCode[FD_READ_BIT]]));
      break;
    end;
    if (NetworkEvents.lNetworkEvents and FD_WRITE = FD_WRITE{READ}) and
      (NetworkEvents.iErrorCode[FD_WRITE_BIT] <> 0) then
    begin
      WriteLn(Format('FD_WRITE failed with error %d',
             [NetworkEvents.iErrorCode[FD_WRITE_BIT]]));
      break;
    end;
    SocketInfo := PSocketInfo(SocketArray[Event - WSA_WAIT_EVENT_0]);
    // Read data only if the receive buffer is empty.
    if SocketInfo^.BytesRECV = 0 then
    begin
      SocketInfo^.DataBuffer.buf := SocketInfo^.Buffer;
      SocketInfo^.DataBuffer.len := DATABUFFSIZE;
      Flags := 0;
      if WSARecv(SocketInfo^.skt, @SocketInfo^.DataBuffer, 1, RecvBytes,
        Flags, NIL, NIL) = SOCKET_ERROR then
      begin
        if WSAGetLastError <> WSAEWOULDBLOCK then
        begin
          FreeSocketInfo(Event - WSA_WAIT_EVENT_0);
          Exit;//return;
        end;
      end
      else
      begin
        SocketInfo^.BytesRecv := RecvBytes;
      end
    end;

    // Write buffer data if it is available.
    if SocketInfo^.BytesRecv > SocketInfo^.BytesSend then
    begin
      SocketInfo^.DataBuffer.buf := SocketInfo^.Buffer + SocketInfo^.BytesSEND;
      SocketInfo^.DataBuffer.len := SocketInfo^.BytesRecv - SocketInfo^.BytesSEND;
      if WSASend(SocketInfo^.skt, @SocketInfo^.DataBuffer, 1, SendBytes, 0,
             NIL, NIL) = SOCKET_ERROR then
      begin
        if WSAGetLastError <> WSAEWOULDBLOCK then
        begin
          WriteLn(Format('WSASend() failed with error %d', [WSAGetLastError]));
          FreeSocketInfo(Event - WSA_WAIT_EVENT_0);
          Exit;
        end;
```

```
          // A WSAEWOULDBLOCK error has occured. An FD_WRITE event will be posted
          // when more buffer space becomes available
        end
        else
        begin
           SocketInfo^.BytesSEND := SocketInfo^.BytesSEND + SendBytes;
           if SocketInfo^.BytesSEND = SocketInfo^.BytesRECV then
           begin
             SocketInfo^.BytesSEND := 0;
             SocketInfo^.BytesRECV := 0;
           end
        end
      end
    end;
    if (NetworkEvents.lNetworkEvents and FD_CLOSE) = FD_CLOSE then
    begin
      if NetworkEvents.iErrorCode[FD_CLOSE_BIT] <> 0 then
      begin
        WriteLn(Format('FD_CLOSE failed with error %d',
               [NetworkEvents.iErrorCode[FD_CLOSE_BIT]]));
        break;
      end;
      WriteLn(Format('Closing socket information %d', [SocketArray[Event -
                      WSA_WAIT_EVENT_0]^.skt]));
      FreeSocketInfo(Event - WSA_WAIT_EVENT_0);
    end;
 end;// while ...
 closesocket(sktListen);
 finally
  WSACleanUp;
 end
 else WriteLn('Failed to load Winsock...');
end.
```

function WSAEnumNetworkEvents   Winsock2.pas

Syntax

WSAEnumNetworkEvents(s: TSocket; hEventObject: WSAEVENT; lpNetworkEvents: LPWSANETWORKEVENTS): Integer; stdcall;

Description

The function performs three tasks: (1) records network events for the selected socket, (2) clears the internal network events record, and (3) optionally resets event objects.

WSAEnumNetworkEvents() works with WSAEventSelect(), which associates an event object with one or more network events.

Parameters

s: A descriptor identifying the socket

hEventObject: An optional handle identifying an associated event object to be reset *lpNetworkEvents*: A pointer to a _WSANETWORKEVENTS record that is filled with a record of occurred network events and any associated error codes

Return Value

If the function succeeds, it will return zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes for each of these events are:

FD_CONNECT: WSAEAFNOSUPPORT, WSAECONNREFUSED, WSAENET-UNREACH, WSAENOBUFS, and WSAETIMEDOUT

FD_CLOSE: WSAENETDOWN, WSAECONNRESET, and WSAECONN-ABORTED

FD_READ, FD_WRITE, FD_OOB, FD_ACCEPT, FD_QOS, FD_GROUP_QOS, and FD_ADDRESS_LIST_CHANGE: WSAENETDOWN

FD_ROUTING_INTERFACE_CHANGE: WSAENETUNREACH and WSAENETDOWN

See Appendix B for a detailed description of the error codes.

See Also

WSAEventSelect

Example

See Listing 5-8 (program EX57).

function WSAEventSelect   Winsock2.pas

Syntax

WSAEventSelect(s: TSocket; hEventObject: WSAEVENT; lNetworkEvents: Longint): Integer; stdcall;

Description

This function associates an event with the supplied set of network events.

Parameters

s: A descriptor identifying the socket

hEventObject: A handle identifying the event object to be associated with the supplied set of FD_XXX network events

lNetworkEvents: A bit mask that specifies the combination of FD_XXX network events in which the application has interest

Return Value

If the function succeeds, it will return zero. If the function fails, it will return a value of SOCKET_ERROR(). To retrieve the error code, call the function WSAGetLastError. Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINVAL, WSAEINPROGRESS, and WSAENOTSOCK.

See Also

WSAAsyncSelect, WSACloseEvent, WSACreateEvent, WSAEnumNetworkEvents, WSAWaitForMultipleEvents

Example

See Listing 5-8 (program EX57).

function WSACloseEvent Winsock2.pas

Syntax

```
WSACloseEvent(hEvent: WSAEVENT): BOOL; stdcall;
```

Description

This function closes an open event object handle.

Parameters

hEvent: Identifies an open event object handle

Return Value

If the function succeeds, it will return TRUE. If the function fails, it will return FALSE. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSA_INVALID_HANDLE.

See Appendix B for a detailed description of the error codes.

See Also

WSACreateEvent, WSAEnumNetworkEvents, WSAEventSelect, WSAGetOverlappedResult, WSARecv, WSARecvFrom, WSAResetEvent, WSASend, WSASendTo, WSASetEvent, WSAWaitForMultipleEvents

Example

See Listing 5-3 (program EX53).

function WSAResetEvent Winsock2.pas

Syntax

```
WSAResetEvent(hEvent: WSAEVENT): BOOL; stdcall;
```

Description

This function resets the state of the specified event object to non-signaled.

Parameters

hEvent: Identifies an open event object handle

Return Value

If the function succeeds, the return value will be TRUE. If the function fails, the return value will be FALSE. To get extended error information, call WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSA_INVALID_HANDLE.

See Appendix B for a detailed description of the error codes.

See Also

WSACloseEvent, WSACreateEvent, WSASetEvent

Example

See Listing 5-7 (program EX52).

function WSASetEvent   Winsock2.pas

Syntax

```
WSASetEvent(hEvent: WSAEVENT): BOOL; stdcall;
```

Description

This function sets the state of the specified event object to be signaled.

Parameters

hEvent: Identifies an open event object handle.

Return Value

If the function succeeds, the return value will be TRUE. If the function fails, the return value will be FALSE. To get extended error information, call WSAGetLastError(). Possible errors are WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, and WSA_INVALID_HANDLE.

See Appendix B for a detailed description of the error codes.

See Also

WSACloseEvent, WSACreateEvent, WSAResetEvent

Example

See Listing 5-7 (program EX52).

function WSAGetOverlappedResult Winsock2.pas

Syntax

WSAGetOverlappedResult(s: TSocket; lpOverlapped: LPWSAOVERLAPPED; var lpcbTransfer: DWORD; fWait: BOOL; lpdwFlags: DWORD): BOOL; stdcall;

Description

This function returns the results of an overlapped operation on the specified socket.

Parameters

s: Identifies the socket. This is the same socket that was specified when the overlapped operation was started by a call to WSARecv(), WSARecvFrom(), WSASend(), WSASendTo(), or WSAIoctl().

lpOverlapped: Points to a WSAOVERLAPPED record that was specified when the overlapped operation was started

lpcbTransfer: Points to a 35-bit variable that receives the number of bytes that were actually transferred by a send or receive operation or by WSAIoctl()

fWait: Specifies whether the function should wait for the pending overlapped operation to complete. If TRUE, the function does not return until the operation has been completed. If FALSE and the operation is still pending, the function returns FALSE and the WSAGetLastError() function returns WSA_IO_INCOMPLETE. The *fWait* parameter may be set to TRUE only if the overlapped operation selected event-based completion notification.

lpdwFlags: Points to a variable that will receive one or more flags that supplement the completion status. If the overlapped operation was initiated via WSARecv() or WSARecvFrom(), this parameter will contain the results value for the *lpFlags* parameter.

Return Value

If the function succeeds, it will return TRUE, indicating the overlapped operation has completed successfully. If the function fails, it will return FALSE. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAENOTSOCK, WSA_INVALID_HANDLE, WSA_INVALID_PARAMETER, WSA_IO_INCOMPLETE, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

WSAAccept, WSAConnect, WSACreateEvent, WSAIoctl, WSARecv, WSARecvFrom, WSASend, WSASendTo, WSAWaitForMultipleEvents

Example

See Listing 5-3 (program EX53).

Raw Sockets

In this short section, we will expose raw sockets. Raw sockets are based on the SOCK_RAW socket type in the AF_INET and AF_ATM address families. Unlike other socket types, such as SOCK_STREAM and SOCK_DGRAM, support for SOCK_RAW is purely optional in the AF_INET address family. That is, it is an optional feature in the Winsock hierarchy that not all vendors support. Fortunately for network developers, Microsoft supports this socket type for both address families, but there is a sting in the tail. For one thing, there are restrictions on the use of raw sockets in the AF_INET address family. Not so with the AF_ATM address family, as SOCK_RAW is the only socket type to use with the AF_ATM address family. But we will not concern ourselves with ATM in this book. (For more information on ATM, consult Appendix C.) So what is this restriction, and why do we have it? Perhaps the best way to answer these questions is to answer the following questions first: What are raw sockets, and why do we have them? Raw sockets work intimately with the IP and ICMP protocols, which underpin the process of message delivery and error reporting mechanisms, respectively. That is, sockets of the type SOCK_RAW provide access to the link layer of the IP layer of the TCP/IP network. Familiar applications, such as ping, traceroute, and other low-level programs, use this intimacy provided by raw sockets. It has been known for unscrupulous (and klutz) hackers to hijack raw sockets to perform denial of service attacks on servers. Because of this easy access to the link layer, it can pose a serious network security problem. To overcome this hurdle without making it impossible to program with raw socket, Microsoft imposes the following restriction on Windows NT 4.0, Windows 2000, and Windows XP: You have to have administrative privileges. The following passage is from the Microsoft MSDN Platform SDK:

> "On Windows NT/Windows 2000, raw socket support requires administrative privileges. Users running Winsock applications that make use of raw sockets must have administrative privileges on the computer, otherwise raw socket calls fail with an error code of WSAEACCESS."

How does a ping application work? To put it simply: A ping application uses raw sockets to send and receive ICMP messages in IP datagrams. Actually, this is an oversimplification of a tricky process. Each ICMP message that the ping application sends is prefaced with an IP header. Figure 5-1 shows this IP header.

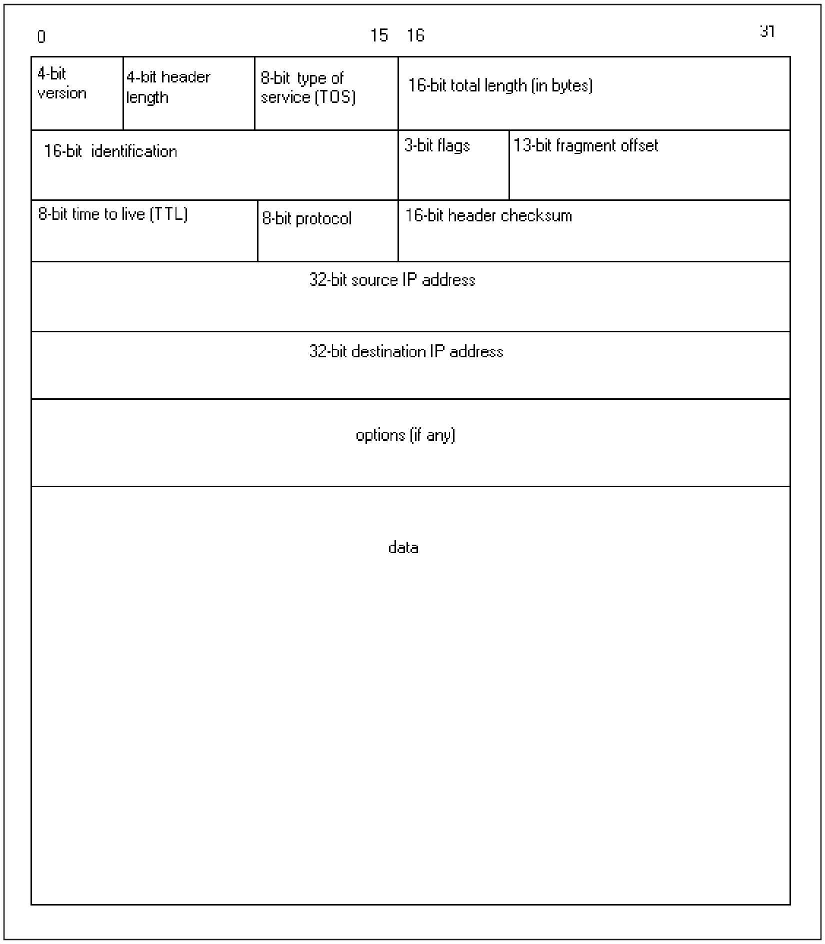


Figure 5-l: IP header

This is a complex header, but suffice it to be aware that TCP/IP protocols, including ICMP, use this header. We won't say any more about this header except that it is used for IP routing, a topic not discussed further in this book. We will refer to this header as we explain how ping works.

As we have found out, an ICMP message is encapsulated as part of the IP datagram. Figure 5-2 shows graphically the structure of the IP datagram.

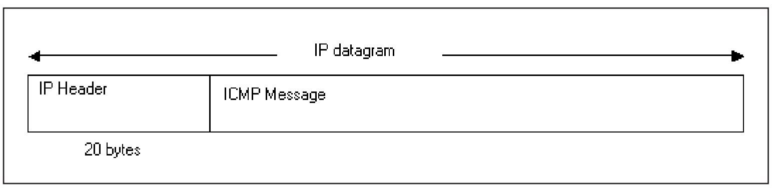


Figure 5-2: The structure of the IP datagram

ICMP is part of the IP layer that is responsible for the communication of errors and conditions that require attention. IP and other higher protocols, usually TCP and UDP, interrogate ICMP for error conditions. Occasionally, ICMP will voluntarily report errors to user processes when necessary. Figure 5-3 shows the structure of the ICMP message that is used for echo request and echo reply employed by the ping program. Again, this is not discussed further in this book.

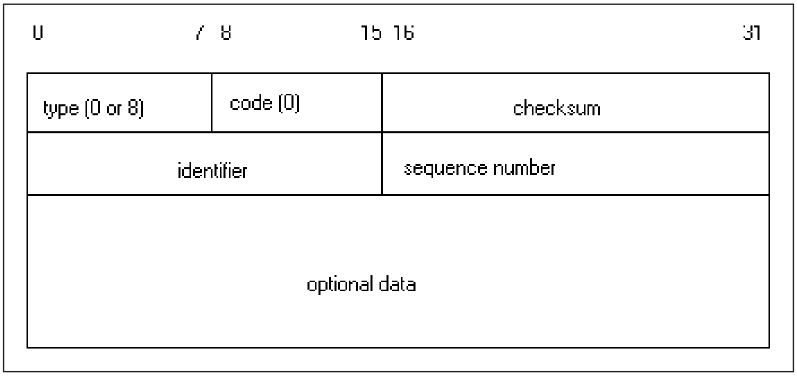


Figure 5-3: The structure of the ICMP message

There are many different types of messages generated by ICMP in response, ranging from fatal error conditions to information reporting. Table 5-10 displays these ICMP messages.

Table 5-10: ICMP messages

Type	Code	Description
0	0	Echo reply
3		Destination unreachable
	0	Network unreachable
	1	Host unreachable
	2	Protocol unreachable
	3	Port unreachable
	4	Fragmentation needed but defragmentation bit is set
	5	Source route failed
	6	Destination network unknown
	7	Destination host unknown
	8	Source host isolated (obsolete)
	9	Destination network administratively prohibited
	10	Host network administratively prohibited
	11	Network unreachable for TOS
	12	Host unreachable for TOS
	13	Communication administratively prohibited
	14	Host precedence violation
	15	Precedence cutoff in effect

Type	Code	Description
4	0	Source quench
5		Redirect:
	0	Redirect for network
	1	Redirect for host
	2	Redirect for type-of-service and network
	3	Redirect for type-of-service and host
8	0	Echo request
9	0	Router advertisement
10	0	Router solicitation
11		Time exceeded:
	0	TTL equals 0 in transit
	1	TTL equals 0 during reassembly
12		Parameter problem:
	0	IP header bad (catchall error)
	1	Required option missing
13	0	Timestamp request
14	0	Timestamp reply
15	0	Information request (obsolete)
16	0	Information request (obsolete)
17	0	Address mask request
18	0	Address mask request

As you can see from the table, there are 15 different types of messages. For the ping application, there are only two messages that are of interest to us: type 0 and type 8.

The ping program was developed by Mike Muuss to check if a host was reachable. The ping program sends an echo request (type 8) to the host. If the host is "switched on," it will send back an ICMP echo reply (type 0). Because of the echoing behavior, its behavior is similar to sonar, so it is popularly called ping. In the days before security became a serious concern, you were always guaranteed to receive an ICMP echo reply when you sent an echo reply request to the target host. This is no longer strictly true as firewalls tend to block out strobes such as pings. In such situations, the host may be hidden behind the firewall from the ping program but still "visible" to other applications using permitted services such as FTP, SMTP, HTTP, and many others. (For followers of *Star Trek*, it is similar to the cloaking device that hides a Romulan vessel.) In spite of this apparent shortcoming, ping is still a useful network-debugging tool.

Like any client-server application pair, you use the ping program as an ICMP client to send an ICMP echo request to a ping server. However, there is a difference between this client-server system and the other client-server systems, such as FTP, SMTP, and many others. The difference is that the ping server is not a user process like FTP; instead, it is a kernel process. In other words, the ping server is part of the kernel and is "switched on" all the time.

Notice that the ping application in Listing 5-9 proceeds as follows:

1. Creates a socket of type SOCK_RAW with the protocol set to ICMP
2. Calls the setsockopt() function to set the timeout, which in this example is two seconds
3. Resolves the name or IP address of the target host
4. Creates a pointer to TICMPHdr and populates the fields in steps 5 to 9
5. Sets the type field to ICMP_ECHOREQ
6. Sets the ID field to the current process ID by a call to GetCurrent-ProcessId()
7. Sets the sequence number
8. Fills the buffer field to any value
9. Calculates the checksum and stores this value in TICMPHdr
10. Calls the sendto() function to send the datagram
11. Decodes the reply and display the results

The ping application repeats steps 5 to 11 inclusive as required. The checksum that is calculated in step 9 is known as the IP checksum. Though we call this checksum the IP checksum, it is used by other protocols such as UDP and TCP. Why do we need a checksum? The checksum is used as a measure to detect data corruption that may have occurred between the sender and the receiver.

If you cast your mind back to the beginning of this chapter, we stated that, unlike TCP, UDP doesn't provide a virtual circuit in which data is transported in a well-behaved queue. Since UDP sits on top of IP, it inherits this behavior from IP. And so it is with ICMP. The implication is that datagrams can arrive in any order, be duplicated or simply be swallowed in a cyberspace black hole. This is the reason for having steps 6 and 7. Raw sockets operate in promiscuous mode. In other words, raw sockets will accept any datagrams that come down on the wire irrespective of their source. To avoid receiving datagrams that come from hosts not targeted by the ping program, you need to add some means of identifying each datagram that you send out. The easiest way to do this is to call GetCurrentProcessId() to get the identifier of your process, which in this case is your ping program. So when your ping program receives a datagram, it checks if the datagram returned by the target host contains the same process identifier. Such a check is as simple as the following snippet of code:

```
if IcmpHeader^.Id <> GetCurrentProcessId then
begin
  WriteLn('someone else''s packet!');
  Exit;
end;
```

On receipt of a datagram, the ping server reflects back the datagram. Since datagrams can come in any order, you need to add a sequence identifier to each datagram. Having this will allow you to detect which:

- datagrams have been dropped.
- datagrams are out of order.
- datagrams have died.

Your ping program must also check the code type returned by the ping server. It should be type 0 for an echo reply. Sometimes, though, the ping server can return types other than 0, so it is necessary for your ping application to check this type code, as the following snippet of code from Listing 5-9 shows:

```
if IcmpHeader._Type <> ICMP_ECHOREPLY then
begin
  WriteLn(Format('Non-echo type %d recvd',[IcmpHeader^._type]));
  Exit;
end;
```

Listing 5-9: The ping program

```
program EX59;

{$APPTYPE CONSOLE}

uses
  Dialogs, SysUtils, Windows,  Winsock2,
  Protocol;

const
  DEF_PACKET_SIZE = 32;
  MAX_PACKET_SIZE = 1024;
  ICMP_MIN = 8;
  ICMP_ECHOREPLY  =  0; // ICMP type: echo reply
  ICMP_ECHOREQ    =  8; // ICMP type: echo request

type

  TCharBuf = array[1..MAX_PACKET_SIZE] of char;

  PICMPHdr = ^TICMPHdr;
  TICMPHdr = packed record
                      Type: Byte;      // Type
                      Code: Byte;      // Code
                      Checksum: WORD;  // Checksum
                      ID: WORD;        // Identification
                      Seq: WORD;       // Sequence
                      Data: LongWord;  // Data
              end;

var
 bufIcmp: TCharBuf;
 iDataSize: Integer = 44;
 Res: Smallint;     //DWORD;
 I: Integer;
 sktRaw: TSocket = INVALID_SOCKET;
 DestAddr,FromAddr: TSockAddrIn;
```

```
 Host: PHostent;
 BRead: Integer;
 FromLen: Integer = SizeOf(FromAddr);
 TimeOut: Integer = 2000;
 IcmpData: PChar;
 RecvBuf: TCharBuf;  //PChar;
 Addr: Cardinal = 0;
 icmp: PIcmpHdr;
 SeqNo: Integer = 0;
 wsaData: TWSADATA;
 nCount: Integer = 0;
 BWrote: Integer = 0;
 FAddr: PChar;
 HostName : String = 'localhost';
 Forever: Boolean = FALSE;
 Position: Integer;

 procedure CopOut(Msg: String);
 begin
   WriteLn(Msg);
   closesocket(sktRaw);
   WSACleanUp;
   Halt;
 end;

{
 The response is an IP packet. We must decode the IP header to locate
 the ICMP data
}

 procedure DecodeResponse(Buffer: TCharBuf; Bytes: Integer; var FromAddr: TSockAddrIn);
 var
  iphdr: PIpHeader;
  IcmpHeader: PICMPHdr;
  iphdrlen: Integer;
 begin
   iphdr := PIpHeader(@Buffer);
   iphdrlen := (iphdr.x and $0F)* 4 ; // number of 32-bit words *4 = bytes
   if Bytes  < (iphdrlen + ICMP_MIN) then
     WriteLn(Format('Too few bytes from %s',[inet_ntoa(FromAddr.sin_addr)]));
   IcmpHeader := PIcmpHdr(@Buffer[iphdrlen + 1]);
   if IcmpHeader._Type <> ICMP_ECHOREPLY then
   begin
     WriteLn(Format('Non-echo type %d recvd',[IcmpHeader^._type]));
     Exit;
   end;
   if IcmpHeader^.Id <> GetCurrentProcessId then
   begin
     WriteLn('someone else''s packet!');
     Exit;
   end;
   WriteLn(Format('%d bytes from %s:',[bytes, inet_ntoa(fromAddr.sin_addr)]));
   WriteLn(Format(' icmp_seq = %d',[IcmpHeader^.Seq]));
   WriteLn(Format(' time: %d ms ',[GetTickCount - LongWord(IcmpHeader^.Data)]));// timestamp
end;

{
 This checksum is taken from Indy's IdICMPClient component. Grateful thanks to the
 makers of Indy components.
}
```

```
function CalcCheckSum: word;
type
  PWordArray = ^TWordArray;
  TWordArray = array[1..512] of word;
var
  pwa: PWordarray;
  dwChecksum: longword;
  i, icWords, iRemainder: integer;
begin
  icWords      := iDataSize div 2;
  iRemainder   := iDatasize mod 2;
  pwa          := PWordArray(@bufIcmp);
  dwChecksum   := 0;
  for i        := 1 to icWords do
  begin
    dwChecksum := dwChecksum + pwa^[i];
  end;
  if (iRemainder <> 0) then
  begin
    dwChecksum := dwChecksum + byte(bufIcmp[iDataSize]);
  end;
  dwCheckSum := (dwCheckSum shr 16) + (dwCheckSum and $FFFF);
  dwCheckSum := dwCheckSum + (dwCheckSum shr 16);
  Result := word(not dwChecksum);
end;

begin
  if ParamCount >= 1 then
  begin
   HostName := ParamStr(1);
   Forever := ParamStr(2) = '-t' ; // we loop forever!
  end;
  if WSAStartUp($0202,wsaData) = 0 then
  begin
   try
{
 Set up for sending and receiving pings
}

  sktRaw := WSASocket (AF_INET, SOCK_RAW, IPPROTO_ICMP, NIL, 0, WSA_FLAG_OVERLAPPED);
  if sktRaw = INVALID_SOCKET then
   CopOut(Format('Call to WSASocket() failed: %d',[WSAGetLastError]));
  Res := setsockopt(sktRaw,SOL_SOCKET,SO_RCVTIMEO,PChar(@Timeout), SizeOf(timeout));
  if Res = SOCKET_ERROR then
   CopOut(Format('Call to setsockopt(SO_RCVTIMEO) failed: %d',[WSAGetLastError]));
  TimeOut := 2000;
  Res := setsockopt(sktRaw,SOL_SOCKET,SO_SNDTIMEO,PChar(@timeout), SizeOf(timeout));
  if Res = SOCKET_ERROR then
   CopOut(Format('Call to setsockopt(SO_SNDTIMEO) failed: %d',[WSAGetLastError]));
   FillChar(DestAddr,SizeOf(DestAddr),0);
  DestAddr.sin_family := AF_INET;
  DestAddr.sin_addr.s_addr := inet_addr(PChar(HostName));
  if DestAddr.sin_addr.s_addr = INADDR_NONE then
  begin
    Host := gethostbyname(PChar(HostName));
    if Host <> NIL then
    begin
      Move(Host.h_addr^, FAddr, Host.h_length);
      DestAddr.sin_addr.S_un_b.s_b1 := Byte(FAddr[0]);
      DestAddr.sin_addr.S_un_b.s_b2 := Byte(FAddr[1]);
```

```
      DestAddr.sin_addr.S_un_b.s_b3 := Byte(FAddr[2]);
      DestAddr.sin_addr.S_un_b.s_b4 := Byte(FAddr[3]);
      DestAddr.sin_family := host.h_addrtype;
    end
    else
     CopOut(Format('Call to gethostbyname() failed: %d',[WSAGetLastError]));
  end;
  while TRUE do
  begin
   if not Forever then
   begin
    inc(nCount);
    if nCount = 4 then
     break;
   end;
{
 Set up for sending and receiving pings
}
    iDataSize := DEF_PACKET_SIZE + sizeof(TIcmpHdr);
    FillChar(bufIcmp, sizeof(bufIcmp), 0);
    icmp := PIcmpHdr(@bufIcmp);
    with icmp^ do
    begin
     _type := ICMP_ECHOREQ;
     code := 0;
     CheckSum := 0;
     id := word(GetCurrentProcessId);
     seq := SeqNo;
{
     Position := SizeOf(ICMP_ECHOREQ) + SizeOf(Code) + SizeOf(CheckSum) + SizeOf(id) +
                 SizeOf(SeqNo);
     WriteLn('Position = ' + IntToStr(Position));
     Move(Windows.GetTickCount, Data, SizeOf(LongWord)); // Not working either!!!!
}
     Data := Windows.GetTickCount;// <<<< original code - doesn't work properly >>>>
{ Fill the buffer with junk after the initialized elements}
     i := Succ(sizeof(TIcmpHdr));
     while i <= iDataSize do
     begin
       bufIcmp[i] := 'E';
       Inc(i);
     end;
     CheckSum := CalcCheckSum;
     inc(SeqNo);
    end;
   BWrote := sendto(sktRaw, bufIcmp, idatasize, 0,@DestAddr, SizeOf(DestAddr));
   if BWrote = SOCKET_ERROR then
   begin
    if WSAGetLastError =  WSAETIMEDOUT then
    begin
      WriteLn('timed out');
      continue;
    end;
    CopOut(Format('Call to sendto() failed: %d',[WSAGetLastError]));
    end;
   if BWrote < idatasize then
    WriteLn(Format('Wrote %d bytes',[BWrote]));
   BRead := recvfrom(sktRaw, RecvBuf, MAX_PACKET_SIZE,0,@FromAddr, FromLen);
   if BRead = SOCKET_ERROR then
   begin
```

```
      if WSAGetLastError = WSAETIMEDOUT then
      begin
        WriteLn('timed out');
        continue;
      end;
      CopOut(Format('Call to recvfrom() failed: %d',[WSAGetLastError]));
    end;
    DecodeResponse(RecvBuf,BRead,FromAddr);
    sleep(2000); { give it a break, man...}
   end;//
   finally
     WSACleanUp;
   end;
  end else
   ShowMessage('Unable to load Winsock 2!');
 end.
```

The traceroute program is another well-known network debugger, devised by Van Jacobson, that also uses the ICMP protocol. The entire traceroute program is presented in Listing 5-10.

The principle of the traceroute program is that it allows us to track the route that IP datagrams take between the sender and the receiver. Although it is not always guaranteed that IP datagrams will always follow the same route for each trace, most of the time they do. Unlike the client-server systems, including ping, a traceroute application does not require a server in the client-server context. Traceroute uses the ICMP message header and the TTL (time to live) field in the IP header to perform "hops." The TTL is a byte field that the sender (your traceroute application) initializes to some value. If you set TTL to 10, this represents a maximum of ten hops or the traversal of up to ten routers that the datagrams can traverse. The algorithm for the traceroute application is best shown as steps, which we present below:

1. Sets the TTL to 1.
2. Sends the IP datagram to the destination host.
3. The router on the route sends back the ICMP header message "time exceeded."
4. Increments the TTL by 1.
5. Repeats steps 2 through 4 until the destination host is reached or until the TTL is equal to an arbitrary figure. When the destination host is reached, the host sends back the ICMP message "port unreachable."

How does the traceroute application know that it has reached the destination host? Although the algorithm is simple, the devil is in the details. We present the traceroute application in Listing 5-10. When you examine the listing, you will appreciate that the traceroute application shares the same code as the ping application. The obvious one is both applications use the same checksum routine.

This sums up a brief introduction to raw sockets, which will allow you to build your own low-level networking applications. However, there is one salient fact to remember: You must have administrator privileges on Windows NT, Windows 2000, or Windows XP before you can develop, debug, and run applications that use raw sockets.

Listing 5-10: The traceroute application

```
program EX510;

{$APPTYPE CONSOLE}

uses
  Dialogs,
  SysUtils,
  Windows,
  Winsock2,
  Protocol,
  WS2tcpip;

const
  DEF_PACKET_SIZE = 32;
  MAX_PACKET_SIZE = 1024;
  ICMP_MIN        = 8; { Minimum size of ICMP header...}
  ICMP_ECHOREPLY  = 0; // ICMP type: echo reply
  ICMP_ECHOREQ    = 8; // ICMP type: echo request
{
 Constants for ICMP message types ...
}
  ICMP_DESTUNREACH =  3;
  ICMP_SRCQUENCH   =  4;
  ICMP_REDIRECT    =  5;
  ICMP_TIMEOUT     = 11;
  ICMP_PARMERR     = 12;

type

  TCharBuffer = array[1..MAX_PACKET_SIZE] of char;

  PICMPHdr = ^TICMPHdr;
  TICMPHdr = record
              Type: Byte;      // Type
              Code: Byte;      // Code
              Checksum: WORD;  // Checksum
              ID: WORD;        // Identification
              Seq: WORD;       // Sequence
              Data: LongWord;  // Data
             end;

var
 BufIcmp,
 RecvBuf: TCharBuffer;
 iDataSize: Integer = 44;
 Res: Integer;
 I: Integer;
 sktRaw: TSocket = INVALID_SOCKET;
 DestAddr,FromAddr: TSockAddrIn;
 Host: PHostent;
 BRead: Integer;
```

```
 FromLen: Integer = SizeOf(FromAddr);
 TimeOut: Integer = 10000;
 Addr: Cardinal = 0;
 icmp: PIcmpHdr;
 SeqNo: Integer = 0;
 wsaData: TWSADATA;
 nCount: Integer = 0;
 BWrote: Integer = 0;
 FAddr: PChar;
 HostName : String = 'localhost';
 Forever: Boolean = FALSE;
 bOption: Boolean = TRUE;
 Done: Boolean = FALSE;
 MaxHops: Byte = 255;
 TTLCount: Byte;

 procedure CopOut(Msg: String);
 begin
   WriteLn(Msg);
   closesocket(sktRaw);
   WSACleanUp;
   Halt;
 end;

{
  Set a TTL for tracing ...
}
 function SetTTL(skt: TSocket; TimeToLive: Integer) : Integer;
 begin
   Result := setsockopt(skt, IPPROTO_IP, IP_TTL, PChar(@TimeToLive), SizeOf(Integer));
   if Result = SOCKET_ERROR then
    CopOut(Format('Call to setsockopt(IP_TTL) failed: %d',[WSAGetLastError]));
  end;

{
 The response is an IP packet. We must decode the IP header to locate
 the ICMP data
}

 function DecodeResponse(Buffer: TCharBuffer; Bytes: Integer; FromAddr: TSockAddrIn; TTL:
Integer): Boolean;
 var
  iphdr: PIpHeader;
  IcmpHeader: PICMPHdr;
  iphdrlen: Integer;
  Host: PHostent;
  FinalDestAddr: TSockAddrIn; //  struct in_addr  inaddr = from->sin_addr;
  P: Pointer;
  Address: Longint;
 begin
   Result := FALSE;
   iphdr := PIpHeader(@Buffer);
   iphdrlen := (iphdr.x and $0F)* 4 ; // number of 32-bit words *4 = bytes
   if Bytes  < (iphdrlen + ICMP_MIN) then
     WriteLn(Format('Too few bytes from %s',[inet_ntoa(FromAddr.sin_addr)]));
   IcmpHeader := PIcmpHdr(@Buffer[iphdrlen + 1]);
   case IcmpHeader._Type of
     ICMP_ECHOREPLY: begin    // Response from destination
                       Address := FromAddr.sin_addr.S_addr;
                       P := system.addr(Address);
```

```
                        Host := gethostbyaddr(P, 4, AF_INET);
                        if Host <> NIL then
                         WriteLn(Format('Host reached => %2d  %s (%s) %d ms', [ttl,
                                 Host^.h_name, inet_ntoa(FromAddr.sin_addr),GetTickCount -
                                 LongWord(ICmpHeader.Data)]));
                        Result := TRUE;
                      end;
      ICMP_TIMEOUT: begin     // Response from router along the way
                        Address := FromAddr.sin_addr.S_addr;
                        P := system.addr(Address);
                        Host := gethostbyaddr(P, 4, AF_INET);
                        if Host <> NIL then
                         WriteLn(Format('%2d  %s (%s)', [ttl, Host^.h_name,
                                 inet_ntoa(FromAddr.sin_addr)]))
                        else
                         WriteLn(Format('%2d  No host name (%s)', [ttl,
                                 inet_ntoa(FromAddr.sin_addr)]));
                       Result := FALSE;
                    end;
      ICMP_DESTUNREACH: begin // Can't reach the destination at all
                          WriteLn(Format('%2d  %s  reports: Host is unreachable', [ttl,
                                         inet_ntoa(FromAddr.sin_addr)]));
                          Result := TRUE;
                        end
        else
        begin
          WriteLn(Format('non-echo type %d received', [IcmpHeader^._type]));
          Result := TRUE;
        end;
    end; // case
end;

function CalcCheckSum: word;
type
  PWordArray = ^TWordArray;
  TWordArray = array[1..512] of word;
var
  pwa: PWordarray;
  dwChecksum: longword;
  i, icWords, iRemainder: integer;
begin
  icWords := iDataSize div 2;
  iRemainder := iDatasize mod 2;
  pwa := PWordArray(@bufIcmp);
  dwChecksum := 0;
  for i := 1 to icWords do
  begin
    dwChecksum := dwChecksum + pwa^[i];
  end;
  if (iRemainder <> 0) then
  begin
    dwChecksum := dwChecksum + byte(bufIcmp[iDataSize]);
  end;
  dwCheckSum := (dwCheckSum shr 16) + (dwCheckSum and $FFFF);
  dwCheckSum := dwCheckSum + (dwCheckSum shr 16);
  Result := word(not dwChecksum);
end;
begin
  if ParamCount >= 1 then
   HostName := ParamStr(1);
```

```
  if WSAStartUp($0202,wsaData) = 0 then
  begin
   try
{
 Set up for sending and receiving pings
}

  sktRaw := WSASocket (AF_INET, SOCK_RAW, IPPROTO_ICMP, NIL, 0, WSA_FLAG_OVERLAPPED);
  if sktRaw = INVALID_SOCKET then
   CopOut(Format('Call to WSASocket() failed: %d',[WSAGetLastError]));
  Res := setsockopt(sktRaw,SOL_SOCKET,SO_RCVTIMEO,PChar(@Timeout), SizeOf(timeout));
  if Res = SOCKET_ERROR then
   CopOut(Format('Call to setsockopt(SO_RCVTIMEO) failed: %d',[WSAGetLastError]));
  TimeOut := 1000;
  Res := setsockopt(sktRaw,SOL_SOCKET,SO_SNDTIMEO,PChar(@timeout), SizeOf(timeout));
  if Res = SOCKET_ERROR then
   CopOut(Format('Call to setsockopt(SO_SNDTIMEO) failed: %d',[WSAGetLastError]));
   FillChar(DestAddr,SizeOf(DestAddr),0);
{
   Set the socket to bypass the standard routing mechanisms
   i.e. use the local protocol stack to the appropriate network interface
}

  if setsockopt(sktRaw, SOL_SOCKET, SO_DONTROUTE, PChar(@bOption), SizeOf(BOOLEAN)) =
              SOCKET_ERROR then
   CopOut(Format('Call to setsockopt(SO_DONTROUTE) failed: %d', [WSAGetLastError]));
  DestAddr.sin_family := AF_INET;
  DestAddr.sin_addr.s_addr := inet_addr(PChar(HostName));
  if DestAddr.sin_addr.s_addr = INADDR_NONE then
  begin
    Host := gethostbyname(PChar(HostName));
    if Host <> NIL then
    begin
      Move(Host.h_addr^, FAddr, Host.h_length);
      DestAddr.sin_addr.S_un_b.s_b1 := Byte(FAddr[0]);
      DestAddr.sin_addr.S_un_b.s_b2 := Byte(FAddr[1]);
      DestAddr.sin_addr.S_un_b.s_b3 := Byte(FAddr[2]);
      DestAddr.sin_addr.S_un_b.s_b4 := Byte(FAddr[3]);
      DestAddr.sin_family := host.h_addrtype;
    end
    else
     CopOut(Format('Call to gethostbyname() failed: %d',[WSAGetLastError]));
  end;
  WriteLn(Format('Tracing route to %s [%s] over a maximum of %d hops: ', [ParamStr(1),
          inet_ntoa(DestAddr.sin_addr), maxhops]));
  TTLCount := 1;
  while (TTLCount <= MaxHops) and (not Done) do
  begin
{
 Set up for sending and receiving pings
}
    setTTL(sktRaw,TTLCount);
    iDataSize := DEF_PACKET_SIZE + sizeof(TIcmpHdr);
    FillChar(bufIcmp, sizeof(bufIcmp), 0);
    icmp := PIcmpHdr(@bufIcmp);
    with icmp^ do
    begin
     _type := ICMP_ECHOREQ;
     code := 0;
     CheckSum := 0;
```

```
      id := word(GetCurrentProcessId);
      seq := SeqNo;
      Data := Windows.GetTickCount;
{ Fill the buffer with junk after the initialized elements}
      i := Succ(sizeof(TIcmpHdr));
      while i <= iDataSize do
      begin
        bufIcmp[i] := 'E';
        Inc(i);
      end;
      CheckSum := CalcCheckSum;
      inc(SeqNo);
     end;
    BWrote := sendto(sktRaw, bufIcmp, idatasize, 0,@DestAddr, SizeOf(DestAddr));
    if BWrote = SOCKET_ERROR then
    begin
     if WSAGetLastError =  WSAETIMEDOUT then
     begin
       WriteLn(Format('%2d *timed out',[SeqNo]));
       continue;
     end;
     CopOut(Format('Call to sendto() failed: %d',[WSAGetLastError]));
     end;
    if BWrote < idatasize then
     WriteLn(Format('Wrote %d bytes',[BWrote]));
    BRead := recvfrom(sktRaw, RecvBuf, MAX_PACKET_SIZE,0,@FromAddr, FromLen);
    if BRead = SOCKET_ERROR then
    begin
      if WSAGetLastError = WSAETIMEDOUT then
      begin
        WriteLn(Format('%2d *timed out',[SeqNo]));
        continue;
      end;
      CopOut(Format('Call to recvfrom() failed: %d',[WSAGetLastError]));
    end;
    Done := DecodeResponse(RecvBuf,BRead,FromAddr, TTLCount);
    sleep(2000); { give it a break, man...}
    inc(TTLCount);
   end;//
   finally
     WSACleanUp;
   end;
  end else
   ShowMessage('Unable to load Winsock 2!');
end.
```


Microsoft Extensions to Winsock 2

In this section we will briefly explore Microsoft extensions to Winsock 2. The extensions are:

- AcceptEx()
- GetAcceptExSockaddrs()
- TransmitFile()
- WSARecvEx()

As you would expect, like the accept() function we examined earlier, AcceptEx() is intended to be used by a server application.

The AcceptEx() function combines several socket functions into a single operation. It performs three tasks:

- Accepts a new connection
- Returns both the local and remote addresses for the connection
- Receives the first block of data sent by the remote

To parse the first data that is accepted by AcceptEx(), you must use the GetAcceptExSockaddrs() function to extract the first data into local and remote addresses. No other function can do this because AcceptEx() writes the data in a special format (called TDI) that only GetAcceptExSockaddrs() can parse. You also need GetAcceptExSockaddrs() to find the sockaddr structures in the buffer accepted by AcceptEx().

The TransmitFile() function uses the operating system's cache manager to transmit file data over a connected socket handle as a high-performance operation. Because of its high-performance file transfer capability, the function is best suited for use on servers running Windows Server versions of operating systems.

The WSARecvEx() function is similar to recv(), except the *flags* parameter in WSARecvEx() is a variable parameter. Use this variable parameter to check whether a partial or complete message has been received using a message-oriented protocol. As with recv(), you can use WSARecvEx() to receive data streams on stream-oriented protocols (TCP).

Although you can use WSARecvEx() with stream protocols, it is pointless to do so because recv() can perform the task equally well, as it is designed to handle data streams. Instead, you should use WSARecv() in situations where you are likely to get partial messages on message-based protocols. When a partial message is received (because the message is larger than the application's buffer, it arrives in several pieces), the MSG_PARTIAL bit is set in the *flags* parameter to indicate to the application that a partial message has been received. When your application receives the whole message at once, the MSG_PARTIAL bit is not set. Contrast this behavior with recv(); recv() does not have a mechanism to detect partial messages when they arrive. Theoretically, you could get away with it by using recv() with a very large buffer to receive the data, but this is rather expensive in terms of resources. Rather, it is more efficient to use WSARecvEx(), which is designed to cope with partial messages.

Let's wrap up this introductory section with a formal definition of these functions.

function AcceptEx *MSWSock.pas*

Syntax

AcceptEx(sListenSocket, sAcceptSocket: TSocket; lpOutputBuffer: LPVOID; dwReceiveDataLength, dwLocalAddressLength, dwRemoteAddressLength: DWORD; var lpdwBytesReceived: DWORD; lpOverlapped: POVERLAPPED): BOOL; stdcall;

Description

The function accepts a new connection, returns the local and remote address, and receives the first block of data sent by the client application. Be aware that this function is not supported on Windows 95/98/Me.

Parameters

sListenSocket: This is a descriptor identifying a socket that has already been called with the listen() function. A server application waits for attempts to connect on this socket.

sAcceptSocket: This is a descriptor identifying a socket on which to accept an incoming connection. This socket must not be bound or connected.

lpOutputBuffer: A pointer to a buffer that receives the first block of data sent on a new connection, the local address of the server, and the remote address of the client. The receive data is written to the first part of the buffer starting at offset zero, while the addresses are written to the latter part of the buffer. This parameter must be specified on operating systems prior to Windows 2000 and can be set to NIL on Windows 2000 or later. If this parameter is set to NIL, no receive operation will be performed, nor will local or remote addresses be available through the use of GetAcceptEx-Sockaddrs() calls.

dwReceiveDataLength: This is the number of bytes in *lpOutputBuffer* that will be used for the data at the start of the buffer. This size should not include the size of the local address of the server, nor the remote address of the client; they are appended to the output buffer. If *dwReceiveDataLength* is zero, accepting the connection will not result in a receive operation. Instead, AcceptEx() completes as soon as a connection arrives, without waiting for any data.

dwLocalAddressLength: This is the number of bytes reserved for the local address information. This value must be at least 16 bytes more than the maximum address length for the transport protocol in use.

dwRemoteAddressLength: This is the number of bytes reserved for the remote address information. This value must be at least 16 bytes more than the maximum address length for the transport protocol in use. It must not be zero.

lpdwBytesReceived: This stores the number of bytes received. This parameter is set only if the operation completes synchronously. If it returns ERROR_IO_PENDING and is completed later, this parameter is never set and you must obtain the number of bytes read from the completion notification mechanism.

lpOverlapped: An overlapped structure that is used to process the request. This parameter must be specified; it cannot be NIL.

Return Value

If no error occurs, the function will return TRUE. If the function fails, AcceptEx() will return FALSE. Use the WSAGetLastError() function to retrieve the error information. If, however, WSAGetLastError() returns the code ERROR_IO_PENDING, the operation was successfully initiated and is still in progress.

See Appendix B for a detailed description of the error codes.

See Also

WSAAccept, WSAConnect, WSACreateEvent, WSAIoctl, WSARecv, WSARecvFrom, WSASend, WSASendTo, WSAWaitForMultipleEvents

procedure GetAcceptExSockaddrs MSWSock.pas

Syntax

```
GetAcceptExSockaddrs(lpOutputBuffer: LPVOID; dwReceiveDataLength,
dwLocalAddressLength, dwRemoteAddressLength: DWORD; var LocalSockaddr:
LPSOCKADDR; var LocalSockaddrLength: Integer; RemoteSockaddr:
LPSOCKADDR; var RemoteSockaddrLength: Integer); stdcall;
```

Description

The procedure parses the data obtained from a call to the AcceptEx() function and passes the local and remote addresses to a sockaddr structure.

Parameters

lpOutputBuffer: A pointer to a buffer that will receive the first block of data sent on a connection resulting from an AcceptEx() call. It must be the same *lpOutputBuffer* parameter that was passed to AcceptEx().

dwReceiveDataLength: The number of bytes in the buffer used for receiving the first data. This value must be equal to the *dwReceiveDataLength* parameter that was passed to the AcceptEx() function.

dwLocalAddressLength: This is the number of bytes reserved for the local address information, which must be equal to the *dwLocalAddressLength* parameter that was passed to the AcceptEx() function.

dwRemoteAddressLength: This is the number of bytes reserved for the remote address information, which must be equal to the *dwRemoteAddressLength* parameter that was passed to the AcceptEx() function.

LocalSockaddr: This is a pointer to the sockaddr structure that will receive the local address of the connection, which is the same information that would be returned by getsockname(). This parameter must be specified.

LocalSockaddrLength: This is the size of the local address and must be specified.

RemoteSockaddr: A pointer to the sockaddr structure that will receive the remote address of the connection, which is the same information that would be returned by the getpeername() function. This parameter must be specified.

RemoteSockaddrLength: This is the size of the local address, which must be specified.

Return Value

This function does not return a value.

See Also

accept, getpeername, getsockname

function TransmitFile MSWSock.pas

Syntax

TransmitFile(hSocket: TSocket; hFile: HANDLE; nNumberOfBytesToWrite, nNumberOfBytesPerSend: DWORD; lpOverlapped: POVERLAPPED; lpTransmit-Buffers: LPTRANSMIT_FILE_BUFFERS; dwReserved: DWORD): BOOL; stdcall;

Description

The function transmits file data over a connected socket handle. This function uses the operating system's cache manager to retrieve the file data and provides high-performance file data transfer over sockets.

Parameters

hSocket: This is a handle to a connected socket over which the function will transmit the file data. The socket specified by *hSocket* must be a connection-oriented socket. The function does not support datagram sockets. Sockets of type SOCK_STREAM, SOCK_SEQPACKET or SOCK_RDM are connection-oriented sockets.

hFile: This is a handle to the open file that the function transmits. Since the operating system reads the file data sequentially, you can improve caching performance by opening the handle with FILE_FLAG_SEQUENTIAL_SCAN. The *hFile* parameter is optional; if the *hFile* parameter is NIL, only data in the header and/or the tail buffer is transmitted, and any additional action, such as socket disconnect or reuse, is performed as specified by the *dwFlags* parameter.

nNumberOfBytesToWrite: A number of file bytes to transmit. The function will complete when it has sent the specified number of bytes or when an error occurs, whichever occurs first. Set *nNumberOfBytesToWrite* to zero in order to transmit the entire file.

nNumberOfBytesPerSend: This is the size of each block of data that will be sent in each send operation, in bytes. Windows' sockets layer uses this specification. To select the default send size, set *nNumberOfBytesPerSend* to zero. The *nNumberOfBytesPerSend* parameter is useful for message protocols that have limitations on the size of individual send requests.

lpOverlapped: A pointer to an overlapped structure. If the socket handle has been opened as overlapped, you must specify this parameter to achieve an overlapped (asynchronous) I/O operation. By default, socket handles are opened as overlapped.

lpTransmitBuffers: A pointer to a TRANSMIT_FILE_BUFFERS data structure that contains pointers to data to send before and after the file data is sent. Set the *lpTransmitBuffers* parameter to NIL if you want to transmit only the file data. The structure is defined in MSWSock.pas and is shown below:

```
_TRANSMIT_FILE_BUFFERS = record
  Head: LPVOID;
  HeadLength: DWORD;
  Tail: LPVOID;
  TailLength: DWORD;
end;
TRANSMIT_FILE_BUFFERS = _TRANSMIT_FILE_BUFFERS;
```

dwReserved: The *dwReserved* parameter has six settings:

- TF_DISCONNECT — Starts a transport-level disconnect after all the file data has been queued for transmission
- TF_REUSE_SOCKET — Prepares the socket handle to be reused. When the TransmitFile() request completes, the socket handle can be passed to the AcceptEx() function. It is only valid if TF_DISCONNECT is also specified.
- TF_USE_DEFAULT_WORKER — Directs the Windows sockets service provider to use the system's default thread to process long TransmitFile() requests. The system default thread can be adjusted using the following registry parameter as a REG_DWORD: CurrentControlSet\Services\afd\Parameters\TransmitWorker.

- TF_USE_SYSTEM_THREAD — Directs the Windows sockets service provider to use system threads to process long TransmitFile() requests
- TF_USE_KERNEL_APC — Directs the driver to use kernel Asynchronous Procedure Calls (APCs) instead of worker threads to process long TransmitFile() requests. Long TransmitFile() requests are defined as requests that require more than a single read from the file or a cache; the request therefore depends on the size of the file and the specified length of the send packet.

  Use of TF_USE_KERNEL_APC can deliver significant performance benefits. It is possible (though unlikely), however, that the thread in which context TransmitFile() is initiated is being used for heavy computations; this situation may prevent APCs from launching. Note that the Windows sockets kernel mode driver uses normal kernel APCs, which launch whenever a thread is in a wait state, which differs from user-mode APCs, which launch whenever a thread is in an alertable wait state initiated in user mode.
- TF_WRITE_BEHIND — Completes the TransmitFile() request immediately, without pending. If this flag is specified and TransmitFile() succeeds, then the data has been accepted by the system but not necessarily acknowledged by the remote end. Do not use this setting with the TF_DISCONNECT and TF_REUSE_SOCKET flags.

Return Value

If the function succeeds, the return value will be TRUE. Otherwise, the return value will be FALSE. To get extended error information, call WSAGetLastError(). The function returns FALSE if an overlapped I/O operation is not complete before TransmitFile() returns. In that case, WSAGetLastError() returns ERROR_ IO_PENDING or WSA_IO_PENDING. Applications should handle either ERROR_IO_PENDING or WSA_IO_PENDING.

See Appendix B for a detailed description of the error codes.

See Also

AcceptEx, WSAGetLastError

function WSARecvEx MSWSock.pas

Syntax

```
WSARecvEx(s: TSocket; buf: PChar; len: Integer; var flags: Integer): Integer; stdcall;
```

Description

The function is identical to the recv() function, except that the *flags* parameter is a variable parameter. When a partial message is received while using the datagram protocol, the MSG_PARTIAL bit is set in the *flags* parameter on return from the function.

Parameters

s: A descriptor identifying a connected socket

buf: A buffer to receive the incoming data

len: The size of *buf*

flags: An indicator specifying whether the message is fully or partially received for datagram sockets

Return Value

If no error occurs, the function will return the number of bytes received. If the connection has been closed, it will return a value of zero. Additionally, if a partial message was received, the MSG_PARTIAL bit is set in the *flags* parameter. If a complete message was received, MSG_PARTIAL is not set in *flags*.

Otherwise, a value of SOCKET_ERROR is returned. You should call WSAGetLastError() to retrieve the specific error code that can be retrieved by calling code.

See Appendix B for a detailed description of the error codes.

See Also

recvfrom, select, send, socket, WSAAsyncSelect

Microsoft Extensions to Winsock 2 for Windows XP and Windows .NET Server

In this section we will briefly explore new functions for Windows XP and Windows .NET Server.

Microsoft added several new functions to the Winsock 2 stable for Windows XP and Windows .NET Server. These functions are specific to Microsoft's implementation of Windows Sockets 2, and there is no sure-fire guarantee that other vendors will support these functions.

These functions, like those we explored briefly in the last chapter, such as getaddrinfo(), are designed for Windows XP and offer support for IPv6.

These functions are listed below.

- ConnectEx()
- DisconnectEx()
- TransmitPackets()
- WSANSPIoctl()
- WSARecvMsg()

Like the getaddrinfo() function that we examined in Chapter 4, these new functions are designed to simplify network programming by replacing some calls with one call. In the case of the ConnectEx() function, you can open the connection on a specified socket and immediately send the first block of data if you elect to do so. Contrast that feature with the current way to call connect() and then send() or WSASend() in a loop to send the data. As you would expect, the ConnectEx() function will only work with stream protocols like SOCK_STREAM, SOCK_RDM, and SOCK_SEQPACKET. The big plus with ConnectEx() is that it uses overlapped I/O (see the section "Using Overlapped Routines") and WSAConnect() doesn't. Potentially, because of its ability to use overlapped I/O, ConnectEx() can handle a large number of clients using a few threads. This is simply not possible with WSAConnect(). Another useful feature is that under certain error conditions, this function is able to reuse the socket.

The DisconnectEx() function closes the stream connection and allows the socket handle to be reused. The DisconnectEx() function does not use datagram sockets. You can use this function with an overlapped structure. To use DisconnectEx(), you will need to call WSAIoctl() with SIO_GET_EXTENSION_FUNCTION_POINTER to obtain a function pointer to it. We will discuss WSAIoctl() in the next chapter.

The TransmitPackets() function is a cost-effective way to send data that is held in memory or a data file on a connected socket because it uses the operating system cache manager to retrieve the data, locking the memory for the shortest time possible to send the data. You can see why this is so if you cast your mind back to how you send data using a function to read the data and then call the send() or WSASend() functions. As with DisconnectEx(), you will need to call WSAIoctl() to create a function pointer to TransmitPackets(). This function, unlike ConnectEx(), can be used with both connected and non-connected sockets.

The WSANSPIoctl() function is used to set or retrieve operating parameters associated with a name space query handle. You can use this either as a blocking or non-blocking function, depending on the application. To make WSANSPIoctl() non-blocking, you would use an overlapped structure in its parameter list; otherwise, you would pass it as a pointer to nothing. The final function is WSARecvMsg(), which receives data as well as optional control information from connected and unconnected sockets. You can use this function instead of WSARecv() and WSARecvFrom().

function ConnectEx MSWSock.pas

Syntax

LPFN_CONNECTEX = function (s: TSocket; name: PSockAddr; namelen: Integer; lpSendBuffer: PVOID; dwSendDataLength: DWORD; lpdwBytesSent: LPDWORD; lpOverlapped: LPOVERLAPPED): BOOL; stdcall;

Description

The function establishes a connection to a specified socket and optionally sends data once the connection is established. The function is only supported on connection-oriented sockets.

Parameters

s: Descriptor identifying an unconnected, previously bound socket.

name: Name of the socket of the sockaddr structure to which to connect

namelen: Length of *name*, in bytes

lpSendBuffer: Pointer to the buffer to be transferred upon connection establishment. This parameter is optional.

dwSendDataLength: Size of data in *lpSendBuffer*. Used when *lpSendBuffer* is not NIL.

lpdwBytesSent: Number of bytes sent from *lpSendBuffer*. Used when *lpSendBuffer* is not NIL.

lpOverlapped: An overlapped structure used to process the request, which must be specified and cannot be NIL.

Return Value

If successful, it will return TRUE; otherwise, it will return FALSE. You should use the WSAGetLastError() function to get extended error information. If WSAGetLastError() returns the code ERROR_IO_PENDING, the operation has initiated successfully and is in progress. Under such circumstances, the call may still fail when the overlapped operation completes.

If the error code returned is WSAECONNREFUSED, WSAENETUNREACH, or WSAETIMEDOUT, the application can call ConnectEx(), WSAConnect(), or connect() again on the same socket.

See Appendix B for a detailed description of the error codes.

See Also

AcceptEx, bind, closesocket, connect, getsockopt, ReadFile, send, setsockopt, TransmitFile, WriteFile, WSAConnect, WSARecv, WSASend, WSAStartUp

function DisconnectEx MSWSock.pas

Syntax

```
LPFN_DISCONNECTEX = function (s: TSocket; lpOverlapped: LPOVERLAPPED;
dwFlags: DWORD; dwReserved: DWORD): BOOL; stdcall;
```

Description

The function closes a connection on a socket and allows the socket handle to be reused.

Parameters

s: A handle to a connected, connection-oriented socket

lpOverlapped: A pointer to an overlapped structure. If the socket handle has been opened as overlapped, specifying this parameter will result in overlapped (asynchronous) I/O operation. Socket handles are overlapped by default.

dwFlags: Specifies a flag that customizes processing of the function call. The *dwFlags* parameter has one optional flag, TF_REUSE_SOCKET. This will allow the socket handle to be reused by AcceptEx() or ConnectEx() when DisconnectEx() is done.

dwReserved: Reserved. Must be zero. If nonzero, the error code WSAEINVAL will be returned.

Return Value

If successful, the function will return TRUE; otherwise, it will return FALSE. Use the WSAGetLastError() function to get extended error information. If WSAGetLastError() returns the code ERROR_IO_PENDING, the operation has been initiated successfully and is in progress.

See Appendix B for a detailed description of the error codes.

See Also

AcceptEx, connect, ConnectEx

function TransmitPackets MSWSock.pas

Syntax

```
LPFN_TRANSMITPACKETS = function (Socket: TSocket; lpPacketArray:
LPTRANSMIT_PACKETS_ELEMENT; ElementCount: DWORD; nSendSize:
DWORD; lpOverlapped: LPOVERLAPPED; dwFlags: DWORD): BOOL; stdcall;
```

Description

The function transmits in-memory data or file data over a connected socket. The function uses the operating system cache manager to retrieve file data, locking memory for the minimum time required to transmit and resulting in efficient, high-performance transmission.

Parameters

Socket: A handle to the connected socket to be used in the transmission. Although the socket does not need to be a connection-oriented circuit, the default destination/peer should have been established using the connect(), WSAConnect(), accept(), WSAAccept(), AcceptEx(), or WSAJoinLeaf() functions.

lpPacketArray: An array of type TRANSMIT_PACKETS_ELEMENT, describing the data to be transmitted

ElementCount: The number of elements in *lpPacketArray*

nSendSize: The size of the data block used in the send operation. Set *nSendSize* to zero to let the sockets layer select a default size for sending.

Setting *nSendSize* to $FFFFFFF enables the caller to control the size and content of each send request, achieved by using the TP_ELEMENT_EOP flag in the TRANSMIT_PACKETS_ELEMENT array pointed to in the *lpPacketArray* parameter. This capability is useful for message protocols that place limitations on the size of individual send requests. The structure of TRANSMIT_PACKETS_ELEMENT is defined in MSWSock.pas and is shown below:

```
_TRANSMIT_PACKETS_ELEMENT = record
  dwElFlags: ULONG;
  cLength: ULONG;
  case Integer of
    0: (
      nFileOffset: LARGE_INTEGER;
      hFile: HANDLE);
    1: (
      pBuffer: LPVOID);
end;
TRANSMIT_PACKETS_ELEMENT = _TRANSMIT_PACKETS_ELEMENT;
PTRANSMIT_PACKETS_ELEMENT = ^TRANSMIT_PACKETS_ELEMENT;
LPTRANSMIT_PACKETS_ELEMENT = ^TRANSMIT_PACKETS_ELEMENT;
TTransmitPacketElement = TRANSMIT_PACKETS_ELEMENT;
PTransmitPacketElement = PTRANSMIT_PACKETS_ELEMENT;
```

lpOverlapped: A pointer to an OVERLAPPED structure. If the socket handle specified in the *Socket* parameter has been opened as overlapped, use this parameter to achieve asynchronous (overlapped) I/O operation. Socket handles are opened as overlapped by default.

dwFlags: Flags used to customize processing of the TransmitPackets() function. Table 5-11 outlines the use of the *dwFlags* parameter.

Table 5-II: Possible values for the dwFlags parameter

Value	Description
TF_DISCONNECT	Starts a transport-level disconnect after all the file data has been queued for transmission. This value applies only to connection-oriented sockets. Specifying this flag for datagram sockets results in an error.
TF_REUSE_SOCKET	Prepares the socket handle to be reused. When the TransmitPackets() function completes, the socket handle can be passed to the AcceptEx() function. This value is valid only when a connection-oriented socket and TF_DISCONNECT are specified.

Value	Description
TF_USE_DEFAULT_WORKER	Directs Windows sockets to use the system's default thread to process long TransmitPackets() requests. Long TransmitPackets() requests are defined as requests that require more than a single read from the file or a cache; the long request definition, therefore, depends on the size of the file and the specified length of the send packet. The system default thread can be adjusted using the following registry parameter as a REG_DWORD: CurrentControlSet/Services/AFD/Parameters/TransmitWorker.
TF_USE_SYSTEM_THREAD	Directs Windows sockets to use system threads to process long TransmitPackets() requests. Long TransmitPackets() requests are defined as requests that require more than a single read from the file or a cache; the long request definition, therefore, depends on the size of the file and the specified length of the send packet.
TF_USE_KERNEL_APC	Directs Windows sockets to use kernel Asynchronous Procedure Calls (APCs) instead of worker threads to process long TransmitPackets() requests. Long TransmitPackets() requests are defined as requests that require more than a single read from the file or a cache; the long request definition, therefore, depends on the size of the file and the specified length of the send packet

Return Value

If successful, the function will return TRUE; otherwise, it will return FALSE. Use the WSAGetLastError() function to retrieve extended error information.

See Appendix B for a detailed description of the error codes.

See Also

accept, AcceptEx, Connect, send, TransmitFile, WSAAccept, WSAConnect, WSAGetOverlappedResult, WSAJoinLeaf

function WSANSPIoctl Winsock2.pas

Syntax

```
WSANSPIoctl(hLookup: HANDLE; dwControlCode: DWORD; lpvInBuffer:
LPVOID; cbInBuffer: DWORD; lpvOutBuffer: LPVOID; cbOutBuffer: DWORD;
lpcbBytesReturned: LPDWORD; lpCompletion: LPWSACOMPLETION): Integer;
stdcall;
```

Description

The function enables developers to make I/O control calls to a registered name space.

Parameters:

hLookup: Lookup handle returned from a call to the WSALookupServiceBegin() function.

dwControlCode: Control code of the operation to perform

lpvInBuffer: Pointer to the input buffer

cbInBuffer: Size of the input buffer

lpvOutBuffer: Pointer to the output buffer

cbOutBuffer: Pointer to an integral value for the size of the output buffer

lpcbBytesReturned: Pointer to the number of bytes returned

lpCompletion: Pointer to a WSACompletion structure used for asynchronous processing. Set *lpCompletion* to NIL to force blocking (synchronous) execution.

Return Value

If successful, the function will return the code NO_ERROR. Otherwise, it will return SOCKET_ERROR, and you should call WSAGetLastError() to retrieve a specific error code.

See Appendix B for a detailed description of the error codes.

See Also

WSAGetLastError, WSALookupServiceBegin, WSALookupServiceEnd, WSALookupServiceNext

function WSARecvMsg MSWSock.pas

Syntax

LPFN_WSARECVMSG = function (s: TSocket; lpMsg: LPWSAMSG; lpdwNumberOfBytesRecvd: LPDWORD; lpOverlapped: LPWSAOVERLAPPED; lpCompletionRoutine: LPWSAOVERLAPPED_COMPLETION_ROUTINE): INT; stdcall;

Description

The function receives data and optional control information from connected and unconnected sockets. This function can be used in place of the WSARecv() and WSARecvFrom() functions.

Parameters

s: Descriptor identifying the socket

lpMsg: A _WSAMSG structure based on Posix.1g specification for the msghdr structure. The structure is defined in MSWSock.pas as:

```
_WSAMSG = record
  name: LPSOCKADDR;           // Remote address
  namelen: INT;               // Remote address length
  lpBuffers: LPWSABUF;        // Data buffer array
  dwBufferCount: DWORD;       // Number of elements in the array
  Control: WSABUF;            // Control buffer
  dwFlags: DWORD;             // Flags
end;
WSAMSG = _WSAMSG;
PWSAMSG = ^WSAMSG;
LPWSAMSG = ^WSAMSG;
TWsaMsg = WSAMSG;
```

lpNumberOfBytesRecvd: A pointer to the number of bytes received, which become immediately available when the WSARecvMsg() function call completes

lpOverlapped: A pointer to a WSAOVERLAPPED structure, which is ignored for non-overlapped structures

lpCompletionRoutine: A pointer to the completion routine called when the receive operation completes, which is ignored for non-overlapped structures

Return Value

On success and immediate completion, the function will return zero. When zero is returned, the specified completion routine is called once the calling thread is in the alertable state. On failure, the function will return a value of SOCKET_ERROR. If a call to WSAGetLastError() returns the code WSA_IO_PENDING, the overlapped operation has been successfully initiated, and completion will be indicated using either events or completion ports.

See Appendix B for a detailed description of the error codes.

See Also

WSAMSG, WSAOverlapped, WSARecv, WSARecvFrom

IP Multicast

This section provides a brief but concise introduction to IP Multicast. This is a fascinating topic in its own right and is the sole subject matter of many networking tomes on the market. No wonder, since it is the communication technology of the future. What we will cover here barely does justice to the topic, but hopefully it will give a taste of what you can do with it in the future. To set out our brief exploration of this topic, we ask the following three questions:

- What is IP Multicast?
- What can you do with it?
- How do you develop a simple IP Multicast application?

What is IP Multicast?

Up to now, we have been looking at one type of IP address: *unicast*. You are forgiven if you thought that this was the only type of IP address. IP supports two other types of addresses: *broadcast* and *multicast*. Strictly speaking, multicast is IP Multicast in this book because there is a form of multicast for the AF_ATM address family. We will not discuss multicast for the AF_ATM address family nor will we cover broadcast in this book. Instead, we will focus on IP Multicast, which is the prevalent form of multicast on the Internet.

What is IP Multicast? The simplest answer to this question is that it is the transfer of IP traffic between a sender and a group of receivers via a special IP address, which, not surprisingly, is called an *IP Multicast address*. It is through this special address that receivers, irrespective of their location on the network, can, by listening to that address, receive all packets from the sender.

Because of this feature, a sender need only send one copy of the data to that special address for delivery to all receivers listening on that address. As you can imagine, IP Multicast (from now on when we refer to multicast we mean IP Multicast) is a very efficient way of sending or "pushing" information to many receivers. By no stretch of the imagination, it is certainly more efficient than TCP, a protocol that can only offer a one-to-one communication circuit.

An analogy to the one-to-many delivery of data would be a radio station (the sender) broadcasting music to anyone (the receiver) who tunes in to listen.

The properties of IP Multicast are:

- A collection of hosts (receivers) that listen on an IP Multicast address is called a *host group*.
- Membership of the host group is dynamic. That is, any host can leave and join the group at any time.
- There is no limitation to the number of hosts in a host group.
- A host group can consist of hosts that are spread across the Internet. That is, the hosts need not be confined to a network segment.
- A sender need not be a member of the host group.

We have been discussing the issue of a special IP Multicast address as though it were one address. Not so! There is a range of addresses, designated as Class D, that are solely for IP Multicast. This class of addresses has a range from 224.0.0.0 to 239.255.255.255. Not all of these addresses are available for use by all. Some of these addresses are reserved for special functions. Table 5-12 enumerates these reserved IP Multicast addresses. You are free to use any other IP Multicast addresses in the range 224.0.1.0 to 238.255.255.255 inclusive but you should be aware of a little caveat: Other IP Multicast applications might be using your very own IP Multicast address for a very different purpose from what you had in mind for your multicasting application. As this coverage is brief, we will not discuss the ramifications of IP Multicast address collisions.

Table 5-12: Reserved IP Multicast addresses

Address	Function
224.0.0.1	All hosts on this subnet
224.0.0.2	All routers on this subnet
224.0.0.5	Open Shortest Path First (OSPF) Version 2, designed to reach all OSPF routers on a network
224.0.0.6	OSPF Version 2, designed to reach all OSPF designated routers on a network

Address	Function
224.0.0.9	Routing Information Protocol (RIP) Version 2
224.0.1.1	Network Time Protocol

Now that we have established what multicast is, how does it actually work in practice? As with other things in life, we have to start at the bottom: the hardware layer. Let's first consider a one-to-one operation (unicast). Every networked PC on a network (usually an Ethernet) has an Ethernet card (a.k.a. NIC, network interface card), which has a unique 48-bit address. Data that is sent between NICs are encapsulated as frames. Each frame has a destination address of the NIC hosted by the target PC. Every NIC on the LAN will receive this frame. However, all NICs, except for the target NIC, will reject this frame, as its destination address will not match with their address. The target NIC accepts this frame and the encapsulated data percolates up from the hardware layer to the TCP/IP stack and then the data is received by the Winsock application.

What Can You Do with IP Multicast?

Unfortunately, the Internet remains a vast ocean of unicast addresses with islands of multicast addresses. Due to this fact, there have been relatively few applications that use multicast. In spite of the slow uptake of multicast applications, it is one of the delivery mechanisms of the future. One reason for this situation is that the majority of routers were designed for unicast routing. This is changing, however, with the replacement of existing routers by those that can handle multicast routing. However, multicasting can occur between these islands through a concept called *IP Tunneling*. IP Tunneling is simply a technique of wrapping IP Multicast datagrams as unicast datagrams. MBone (Internet Multicast Backbone) uses this concept successfully to exchange data between islands of multicast addresses. MBone is heavily used for audio and video multicasts of Internet Engineering Task Force (IETF) meetings, and communications and meetings of NASA, the U.S. House of Representatives, and the Senate. We will not dive into the topic of IP Tunneling, as it is beyond the scope of this book.

Those multicast applications that have appeared so far cater to the following tasks:

- File transfer; file updates
- Transmission of data; live feeds
- Multimedia applications

How Do You Develop a Simple IP Multicast Application?

In spite of the fact that the data propagation on the Internet is still predominantly unicast-based, there is nothing to stop you from developing a multicast application for use on your LAN or company's Intranet. Unlike routers on the Internet, LANs are equipped to handle multicast because Ethernet cards are preconfigured for multicast. The only problem you would have is that a router sitting between your LANs may not support multicast routing. Without further ado, let's jump to it.

In fact, you would need to develop two Winsock 2 multicast applications; one is the sender application that sits on one machine, and the other is the receiver application that sits on PCs on the same LAN. Let's discuss the server application first.

> **NOTE:** The Winsock 1 version of IP Multicasting is implemented differently, but we will not discuss the Winsock 1 implementation in this book.

Before the server can send any data, it has to perform several tasks, including initializing special data structures and binding the multicast address that your clients' applications will tune in to listen. The following steps outline a typical multicast sender using Winsock 2:

1. Call WSASocket() to create a UDP socket. You should call this function with the *dwFlags* parameter set to WSA_FLAG_MULTIPOINT_C_LEAF, WSA_FLAG_MULTIPOINT_D_LEAF, or WSA_FLAG_OVERLAPPED. This is to indicate to Winsock that the socket is to be used for multicast.
2. Set up the socket address for the local interface and call bind().
3. Set up the socket address for the remote address (i.e., the IP Multicast address to which the sender application will send the data.) For example, the IP address would be something like 224.1.2.3.4.
4. By default, the TTL is set to 1. To send the data to the host group beyond the local network, you will need to set the TTL to 8. Do this by calling setsockopt() with the IP_MULTICAST_TTL option.
5. To disable loopback of datagrams, call setsockopt() with the IP_MULTICAST_LOOP option.
6. Call WSAJoinLeaf() with the JL_BOTH option to join the host group. This is not strictly necessary for a sender, but it is an absolute must for a receiver.
7. Call sendto() to send the data until complete.

Steps for running a multicast receiver are essentially the same as the sender, except in step 7 where the receiver receives the datagrams as they arrive at the IP Multicast address. Listings 5-11 and 5-12 give the source code for the sender and receiver applications. Although we haven't discussed IP Multicast from the perspective of the Winsock 1.1 developer, we have included the Winsock 1.1 version as EX513 on the companion CD. This application does not use WSAJoinLeaf(), as it is a Winsock 2 function.

Naturally a full-fledged multicast sender and receiver would be more complex than the steps described above. For example, in a file transfer using multicast, the receiver would have to reassemble the datagrams to build the file. That is, if the receiver finds any datagrams missing, corrupted, or duplicated, it would need to notify the sender of this fact. This requires additional and complex algorithms to solve this particular problem, which is beyond the scope of this book. To join an IP Multicast session, you should call WSAJoinLeaf(). A sender does not need to join the host group, but the receiver must in order to tune in to the datagrams. In a simple multicast application as we have described above, we call WSAJoinLeaf() like this:

```
WSAJoinLeaf(skt, @RemoteAddr, SizeOf(RemoteAddr), NIL, NIL, NIL, NIL, JL_BOTH);
```

The first parameter is the socket that we use to join the host group. The second parameter is the socket address at which the receivers receive the data. The third parameter specifies the size of the socket address. The following four parameters, *lpCallerData*, *lpCalleeData*, *lpSQOS*, and *lpGQOS*, are set to NIL. The *lpCallerData* and *lpCalleeData* parameters specify the exchange of user data. The *lpSQOS* parameter specifies a pointer to a special structure that is used for Quality of Service (QOS) mechanisms, which is beyond the scope of this book. The *lpGQOS* parameter, which is not implemented in the current version of Winsock 2, specifies the socket groups to be used with the structure for QOS. The last parameter, *dwFlags*, specifies how the socket should be used. If the socket is acting as a sender, use the JL_SENDER_ONLY flag. If the socket is acting as a receiver, use JL_RECEIVER_ONLY. If you want the socket to send and receive data, use the JL_BOTH flag. This raises an interesting thought: If the sender can also act as a receiver, you could have a many-to-many multicast session. For example, you could develop a many-to-many chat application.

To conclude this short section, we will give a formal definition of the WSAJoinLeaf() function.

function WSAJoinLeaf Winsock2.pas

Syntax

WSAJoinLeaf(s: TSocket; name: PSockAddr; namelen: Integer; lpCallerData: LPWSABUF; lpCalleeData: LPWSABUF; lpSQOS, lpGQOS: LPQOS; dwFlags: DWORD): TSocket; stdcall;

Description

The function joins a leaf node into a multipoint session, exchanges connect data, and specifies quality of service based on the specified FLOWSPEC structures.

Parameters

s: The descriptor identifying a multipoint socket

name: The name of the peer to which the socket is to be joined

namelen: The length of *name*

lpCallerData: A pointer to the user data that is to be transferred to the peer during multipoint session establishment

lpCalleeData: A pointer to the user data that is to be transferred back from the peer during multipoint session establishment

lpSQOS: A pointer to the FLOWSPEC structures for socket *s*, one for each direction

lpGQOS: Reserved for future use with socket groups; a pointer to the FLOWSPEC structures for the socket group (if applicable)

dwFlags: Flags to indicate that the socket is acting as a sender (JL_SENDER_ONLY), receiver (JL_RECEIVER_ONLY), or both (JL_BOTH)

Return Value

If no error occurs, the function will return a socket descriptor for the newly created multicast socket. Otherwise, the function will return a value of INVALID_SOCKET. To retrieve a specific error code, you should call WSAGetLastError(). On a blocking socket, the return value will indicate success or failure of the join operation. On the other hand, with a non-blocking socket, the function will indicate a successful initiation of a join operation by returning a valid socket descriptor. When you use this function with WSAAsyncSelect() or WSAEventSelect(), and a network event (FD_CONNECT) occurs, an indication will be given on the original socket *s* when the join operation completes, either successfully or otherwise. If WSAGetLastError() returns one of these codes—WSAECONNREFUSED, WSAENETUNREACH or WSAETIMEDOUT—you can call WSAJoinLeaf() on the same socket.

See Appendix B for a detailed description of the error codes.

See Also

accept, bind, select, WSAAccept, WSAAsyncSelect, WSAEventSelect, WSASocket

Example

Listing 5-11 (EX511) demonstrates how to use WSAJoinLeaf() when an IP Multicast application sends data to the IP Multicast address 234.5.6.7 to which receivers listen. Take a look at program EX512 (available on the companion CD), which also demonstrates the use of WSAJoinLeaf(), as well as how to receive data from program EX511.

Listing 5-11: A simple IP Multicast sender application

```
program EX511;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows,
  Winsock2,
  WS2tcpip;

const

 MCASTADDR     = '234.5.6.7';
 MCASTPORT     = 25000;
 BUFSIZE       = 1024;
 DEFAULT_COUNT = 500;

var
 Sender: Boolean = TRUE;      // Act as a sender?
 LoopBack: Boolean = FALSE;   // Disable loopback?
 dwInterface,                 // Local interface to bind to
 dwMulticastGroup,            // Multicast group to join
 dwCount: DWORD;              // Number of messages to send/receive
 iPort: WORD;                 // Port number to use
 wsaData: TWSADATA;
 LocalAddr,
 RemoteAddr,
 FromAddr: TSockAddrIn;
 skt,
 sktMC: TSocket;
 recvbuff,
 sendbuff: array[0..BUFSIZE - 1] of char;
 Len: Integer = SizeOf(TSockAddrIn);
 optval,
 Res: Integer;
 i: DWORD;

begin
  if WSAStartup($0202,wsaData) = 0 then
  begin
    try
      dwInterface := INADDR_ANY;
      dwMulticastGroup := inet_addr(MCASTADDR);
      iPort := MCASTPORT;
```

```
      dwCount := DEFAULT_COUNT;
// Create a socket ...
      skt := WSASocket(AF_INET, SOCK_DGRAM, 0, NIL, 0, WSA_FLAG_MULTIPOINT_C_LEAF or
                 WSA_FLAG_MULTIPOINT_D_LEAF or WSA_FLAG_OVERLAPPED);
      if skt = INVALID_SOCKET then
      begin
        WriteLn(Format('Call to WsaSocket() failed with: %d', [WSAGetLastError]));
        WSACleanup;
        Halt;
      end;
// Bind to the local interface. This is done to receive data.
      LocalAddr.sin_family := AF_INET;
      LocalAddr.sin_port   := htons(iPort);
      LocalAddr.sin_addr.s_addr := dwInterface;
      Res := bind(skt, @LocalAddr, SizeOf(LocalAddr));
      if Res = SOCKET_ERROR then
      begin
        WriteLn(Format('Call to bind() failed with: %d', [WSAGetLastError]));
        closesocket(skt);
        WSACleanup;
        Halt;
      end;
// Setup the SOCKADDR_IN structure describing the multicast group we want to join
      RemoteAddr.sin_family      := AF_INET;
      RemoteAddr.sin_port        := htons(iPort);
      RemoteAddr.sin_addr.s_addr := dwMulticastGroup;
// Change the TTL to something more appropriate
      optval := 8;
      Res := setsockopt(skt, IPPROTO_IP, IP_MULTICAST_TTL, PChar(@optval), SizeOf(Integer));
      if Res = SOCKET_ERROR then
      begin
        WriteLn(Format('Call to setsockopt(IP_MULTICAST_TTL) failed: %d',
               [WSAGetLastError]));
        closesocket(skt);
        WSACleanup;
        Halt;
      end;
// Disable loopback ...
      optval := 0;
      Res := setsockopt(skt, IPPROTO_IP, IP_MULTICAST_LOOP, PChar(@optval), SizeOf(optval));
      if Res = SOCKET_ERROR then
      begin
        WriteLn(Format('Call to setsockopt(IP_MULTICAST_LOOP) failed: %d',
                      [WSAGetLastError]));
        closesocket(skt);
        WSACleanup;
        Halt;
      end;
// Join the multicast group. Note that sockM is not used
// to send or receive data. It is used when you want to
// leave the multicast group. You simply call closesocket()
// on it.
      sktMC := WSAJoinLeaf(skt, @RemoteAddr, SizeOf(RemoteAddr), NIL, NIL, NIL, NIL,
                     JL_BOTH);
      if sktMC = INVALID_SOCKET then
      begin
        WriteLn(Format('Call to WSAJoinLeaf() failed: %d', [WSAGetLastError]));
        closesocket(skt);
        WSACleanup;
        Halt;
```

```
      end;
// Now send data
      while TRUE do
      begin
         StrPCopy(sendbuff,Format('Server 1: This is a test: %d', [i+1]));
         inc(i);
         Res := sendto(skt, sendbuff, StrLen(sendbuff), 0, @RemoteAddr, SizeOf(RemoteAddr));
         if Res = SOCKET_ERROR then
         begin
           WriteLn(Format('Call to sendto() failed with: %d',[WSAGetLastError]));
           closesocket(sktMC);
           closesocket(skt);
           WSACleanup;
           Halt;
         end;
         Sleep(250);
      end;
// Leave the multicast group by closing sock
// For non-rooted control and data plane schemes, WSAJoinLeaf
// returns the same socket handle that you pass into it.
//
     closesocket(skt);
    finally
      WSACleanup;
    end;
  end else
   WriteLn('Unable to load Winsock 2!');
end.
```

Obsolete Functions

In this section, we will discuss obsolete functions to complete our coverage. These functions are specific to Winsock 1.1, but we include these here for completeness. These functions manage Winsock 1.1 blocking functions.

NOTE: Winsock 2 applications should not use any of the functions in this section.

function WSACancelBlockingCall Winsock2.pas

Syntax

WSACancelBlockingCall : integer;

Description

This function cancels a blocking call that is in progress and any outstanding blocking operation for this thread. You would use this function in two cases:

- In the first case, suppose our application is processing a message that has been received while a blocking call is in progress. In this case, WSAIsBlocking() will be TRUE.

- In the second case, suppose that a blocking call is in progress, and Winsock has called back to the application's blocking hook function as established by WSASetBlockingHook().

In each case, the original blocking call will terminate as soon as possible with the error WSAEINTR. In the first case, the termination will not take place until Windows message scheduling has caused control to revert to the blocking routine in Winsock. In the second case, the blocking call will terminate as soon as the blocking hook function completes. Now we will consider the effects of calling WSACancelBlockingCall() on blocking operations, such as connect(), accept(), and select().

When you call WSACancelBlockingCall() to cancel a connect operation, Winsock will terminate the blocking call as soon as possible. However, it may not be possible to release the socket resources until the connection has completed (and then been reset) or timed out. This is likely to be noticeable only if the application immediately tries to open a new socket (if no sockets are available) or connect to the same peer.

Canceling an accept() or select() call does not affect the sockets passed to these calls, but the blocking call will fail. Canceling any other blocking operation other than accept() and select() can leave the socket in an indeterminate state. Therefore, to be on the safe side, you must always call closesocket() after canceling a blocking operation on a socket.

Return Value

If the function succeeds, it returns zero, indicating that the overlapped operation has completed successfully. If the function fails, it returns the value of SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, and WSAEOPNOTSUPP.

See Appendix B for a detailed description of the error codes.

See Also

WSAIsBlocking, WSASetBlockingHook, WSAUnhookBlockingHook

function WSAIsBlocking Winsock2.pas

Syntax

```
WSAIsBlocking: Bool;
```

Description

In a 16-bit environment like Windows 3.1, this function allows a Winsock 1.1 application to determine if it is executing while waiting for a previous blocking call to complete. In other words, you can use the WSAIsBlocking function to check if the task has been re-entered while waiting for an outstanding blocking call to complete.

Return Value

The return value is TRUE if there is an outstanding blocking function awaiting completion in the current thread. Otherwise, it is FALSE. Call WSAGetLast-Error() to retrieve the error code.

See Appendix B for a detailed description of the error codes.

See Also

WSACancelBlockingCall, WSASetBlockingHook, WSAUnhookBlockingHook

function WSASetBlockingHook Winsock2.pas

Syntax

WSASetBlockingHook(lpBlockFunc: TFarProc): TFarProc; stdcall;

Description

This function establishes a blocking hook function supplied by your application. A Winsock implementation includes a default mechanism by which blocking socket functions are implemented. This function gives the application the ability to execute its own function at "blocking" time in place of the default function.

Use the WSASetBlockingHook() function to create your own blocking hook function to handle more complex message processing that the default blocking mechanism cannot handle adequately. The only caveat here is that, with the exception of WSACancelBlockingCall(), you cannot call other Winsock 1.1 functions. Calling WSACancelBlockingCall() will, of course, cause the blocking loop to terminate.

Parameters

lpBlockFunc: A pointer to the blocking function to be installed

Return Value

The return value is a pointer to the previously installed blocking function. An application that calls the WSASetBlockingHook() function should save this return value so that the application can restore it if necessary. (If "nesting" is not important, the application may simply discard the value returned by WSASetBlockingHook() and eventually use WSAUnhookBlockingHook() to restore the default mechanism.) If the operation fails, a NIL pointer is returned, and a specific error number may be retrieved by calling WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS, WSAEFAULT, and WSAEOPNOTSUPP.

See Appendix B for a detailed description of the error codes.

See Also

WSACancelBlockingCall, WSAIsBlocking, WSAUnhookBlockingHook

function WSAUnhookBlockingHook Winsock2.pas

Syntax

WSAUnhookBlockingHook;

Description

This function restores the default blocking hook function. Calling this removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism. That is, WSAUnhookBlockingHook() will always install the default mechanism, and never the previous mechanism.

Return Value

If the function succeeds, it returns zero. If the function fails, it returns a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAEINPROGRESS, and WSAEOPNOTSUPP.

See Appendix B for a detailed description of the error codes.

See Also

WSACancelBlockingCall, WSAIsBlocking, WSASetBlockingHook

Summary

We have reached the end of a particularly long chapter. In this chapter, we learned the techniques of opening a connection, managing data exchange, and breaking the connection. We also learned how to use the I/O schemes to manage the data exchange and to select which I/O scheme is appropriate for a server or client application.

In the next chapter, which will be considerably shorter, we will examine ways of modifying attributes of a socket, which will modify the way our application handles the data transfer. Just as important, we will also discuss how to retrieve an attribute of a socket.

Chapter 6

Socket Options

In the last chapter, we exposed the full spectrum of opening a connection, managing data exchange, and closing a connection. In that discussion, we touched upon the topic of setting and querying the attributes of a socket. In this chapter, the last on Winsock 2, we will explore those functions that query and modify the attributes of a socket. We will also explore the functions that control the I/O behavior of a socket.

Querying and Modifying Attributes

In this section, we will learn how to use getsockopt() and setsockopt() to query and modify the attributes of a socket, respectively.

Why would we want to query the attributes, or *options*, of a socket? And why would we want to set the options? The answer to both of these questions is not just to obtain the information, but to use the information gleaned from getsockopt(), if we wish, to fine-tune the behavior of the socket. To fine-tune a socket's attributes, you should use setsockopt(). Let's take an example from a real-life situation. Very often, you may want to increase the timeout on a receiving socket from 2 to 20 seconds on an extremely slow network. You would use the getsockopt() function to verify that the receiving socket's timeout is, indeed, 2 seconds. You would call it like the following:

```
// Retrieve the value to verify what we set ...
   Res := getsockopt(skt, SOL_SOCKET, SO_RCVTIMEO, PChar(@Value),Size);
   if Res = SOCKET_ERROR then

{rest of code}
```

Don't worry about the parameters in getsockopt(), as we will explain these shortly.

Having satisfied yourself that the timeout value is 2 seconds, call setsockopt() to set a new timeout value of 20 seconds, as the following snippet of code illustrates:

```
// Now set the time-out value to 20 ...
   Value := 200;
   Size := SizeOf(Value);
   Res := setsockopt(skt, SOL_SOCKET, SO_RCVTIMEO, PChar(@Value), Size);
```

```
    if Res = SOCKET_ERROR then

{rest of code}
```

Now that we have demonstrated how to use the getsockopt() and setsockopt() functions (admittedly contrived), it's time for us to examine these prototypes, which are defined in Winsock2.pas. We will start with the getsockopt() function:

```
function getsockopt(s: TSocket; level, optname: Integer; optval: PChar; var optlen: Integer):
Integer; stdcall;
```

The first parameter, *s*, is the socket with options that you wish to query. The second parameter, *level*, defines the level of the socket options. We will discuss this parameter in detail shortly. The third parameter, *optname*, is the name of the socket option that you wish to discover. The fourth parameter, *optval*, contains the options set of that *level* for that socket. Note that this parameter is a PChar type, so you always typecast this as a PChar variable.

In the case of the SO_RCVTIMEO option in the preceding code fragment, typecast the time in seconds as a PChar variable. (This typecasting also applies to setsockopt(), by the way.) The last parameter, *optlen*, defines the length of the result. For example, when you call getsockopt() with the SO_LINGER option as the *optname* parameter, *optlen* will be the size of the TLinger record (see the definition of TLinger record in Winsock2.pas). For the majority of socket options, the size of the socket option is usually the size of an integer. If a socket option was never set with setsockopt(), getsockopt() returns the default value for the socket option.

Remember from our discussion on WSAStartup() in Chapter 2 that it is not possible to get full details of Winsock 2's properties. You can get over this hurdle by calling getsockopt() to retrieve the details. For example, to retrieve information on the maximum message size (from the *iMaxUdpDg* field in the TWSAData record), you would call getsockopt() with the SO_MAX_MSG_SIZE option.

When you want to modify the behavior of a socket, you should call setsockopt() to set the attributes, or options, for that socket. We show its prototype, which is also defined in Winsock2.pas:

```
function setsockopt(s: TSocket; level, optname: Integer; optval: PChar; optlen: Integer):
Integer; stdcall;
```

Since the parameters for setsockopt() are similar to those for getsockopt(), we will not describe them again. However, there are two types of socket options that you must bear in mind, which are as follows:

- **Boolean options** — Enables or disables a feature or behavior. To enable a Boolean option, you should set the *optval* parameter to a nonzero integer. Conversely, to disable the option, you should set the *optval* parameter to zero. The field *optlen* must always be equal to the size of an integer.

- **Integer options** — Require an integer value or record. For other options, *optval* points to an integer or record that contains the desired value for the option, and *optlen* is the length of the integer or record.

By now, you must be wondering about the mysterious second parameter, *level,* that is common to both functions. The explanation is that the *level* parameter refers to a particular grouping of socket options. We group these options into units or, more often in Winsock parlance, into *levels*. Winsock 2 supports a number of levels, such as SOL_SOCKET, SOL_APPLETALK, and many others. However, unlike Winsock 2, Winsock 1.1 provides support for only two levels of socket options, SOL_SOCKET and IPPROTO_TCP. Some implementations of Winsock 1.1 may support the IPPROTO_IP level. Both versions of Winsock (1 and 2) always support the SOL_SOCKET level, which is not protocol dependent. Table 6-1 tabulates the options in both SOL_SOCKET and IPPROTO_TCP levels that are common to both versions of Winsock.

As the focus in the rest of this chapter is on Microsoft's implementation of Winsock on Windows platforms, we will not cover levels that are relevant to Novell's IPX/SPX or Apple's AppleTalk or ATM protocols. The levels that we will cover here are SOL_SOCKET, IPPROTO_TCP, and IPPROTO_IP. Although Microsoft recently added a new level, SOL_IRLMP for infrared devices, we will not discuss SOL_IRLMP in this tome.

Table 6-1: Base levels

Level = SOL_SOCKET			
Value	**Type**	**Meaning**	**Default**
SO_ACCEPTCONN	BOOL	If TRUE, socket is listening.	FALSE
SO_BROADCAST	BOOL	If TRUE, socket is configured for the transmission of broadcast messages.	FALSE
SO_DEBUG	BOOL	If TRUE, debugging is enabled.	FALSE
SO_DONTLINGER	BOOL	If TRUE, the SO_LINGER option is disabled.	TRUE
SO_DONTROUTE	BOOL	If TRUE, routing is disabled.	FALSE
SO_ERROR	Integer	Retrieves error status and clear.	0
SO_KEEPALIVE	BOOL	Keepalives are being sent.	FALSE
SO_LINGER	TLinger	Returns the current linger options.	l_onoff is 0
SO_MAX_MSG_SIZE	Unsigned integer	Maximum outbound (send) size of a message for message-oriented socket types (e.g., SOCK_DGRAM). There is no provision for finding out the maximum inbound message size. This has no meaning for stream-oriented sockets.	Implementation dependent
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream.	FALSE
SO_PROTOCOL_INFO	WSAPROTOCOL_INFO	Description of protocol info for protocol that is bound to this socket.	Protocol dependent

Value	Type	Meaning	Default
SO_RCVBUF	Integer	Total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.	Implementation dependent
SO_REUSEADDR	BOOL	The socket may be bound to an address that is already in use.	FALSE
SO_SNDBUF	Integer	Total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.	Implementation dependent
SO_TYPE	Integer	The type of the socket (e.g., SOCK_STREAM).	As created via socket API
PVD_CONFIG	Service provider dependent	An "opaque" data structure object from the service provider associated with socket s. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.	Implementation dependent

Level = IPPROTO_TCP			
Value	**Type**	**Meaning**	**Default**
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.	Implementation dependent

TIP: Unlike Winsock 1.1, retrieving configuration information for Winsock 2 is not easy, if not impossible, because of Winsock 2's more elaborate architecture to support multiple protocols. However, you can retrieve some of this information by calling getsockopt() with the socket option PVD_CONFIG, providing that you know the data structure of the record provided by the Winsock vendor.

The socket options that all versions of Winsock support are a subset of the BSD socket options. For those Delphi developers with a UNIX and Linux background, Tables 6-2 and 6-3 list those BSD socket options that getsockopt() and setsockopt() under Winsock do not support.

Table 6-2: BSD socket options not supported by getsockopt()

Value	Type	Meaning
SO_RCVLOWAT	Integer	Receive low water mark
SO_SNDLOWAT	Integer	Send low water mark
TCP_MAXSEG	Integer	Get TCP maximum segment size

Table 6-3: BSD socket options not supported by setsockopt()

Value	Type	Meaning
SO_ACCEPTCONN	BOOL	Socket is listening
SO_RCVLOWAT	Integer	Receive low water mark
SO_SNDLOWAT	Integer	Send low water mark
SO_TYPE	Integer	Type of socket

Table 6-4 shows a complete list of levels and their corresponding grouping of options that getsockopt() can use under Winsock 1.1 and Winsock 2.

Table 6-4: Levels and options that getsockopt() can use

Level = SOL_SOCKET		
Value	**Type**	**Meaning**
SO_ACCEPTCONN	BOOL	FALSE unless a WSPListen()* has been performed.
SO_BROADCAST	BOOL	Allow transmission of broadcast messages on the socket.
SO_DEBUG	BOOL	Record debugging information.
SO_DONTLINGER	BOOL	Don't block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with l_onoff set to zero.
SO_DONTROUTE	BOOL	Do not route, but send directly to interface.
SO_KEEPALIVE	BOOL	Send keepalives.
SO_LINGER	TLinger	Linger on close if unsent data is present
SO_OOBINLINE	BOOL	Receive out-of-band data in the normal data stream.
SO_RCVBUF	Integer	Specify the total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.
SO_REUSEADDR	BOOL	Allow the socket to be bound to an address that is already in use. (See bind().)
SO_SNDBUF	Integer	Specify the total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.

Level = IPPROTO_TCP**		
Value	**Type**	**Meaning**
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

* This is the listen() function in the Winsock 2 Service Provider API, which is not discussed in this tome.

**Included for backward compatibility with WinSock 1.1

There are constraints and traps that you should be aware of when using the getsockopt() and setsockopt() functions that are easily overlooked during the development process. I am often guilty of these lapses, so I will describe these traps for both the wary and the not-so-wary.

A factor that you must remember is that some of these socket options are platform specific. For example, the socket option SO_EXCLUSIVEADDRUSE is only available in Winsock 2 on Windows 2000 and later. So when you use either the getsockopt() or setsockopt() functions with an unsupported socket option, Winsock will return an error code of WSAENOPROTOOPT.

Another important constraint you should remember is that some of the socket options are only available for inspection after the socket is connected. For example, the option SO_CONNECT_TIME that you would use with getsockopt() to return the time (in seconds) that the socket has been connected would return an invalid value of $FFFFFFFF for a non-connected socket.

Another factor to consider is that some options only make sense with the correct socket type(s). For example, the option SO_DONTLINGER only applies to sockets of the SOCK_STREAM type.

In the rest of this chapter, we will focus on options for each level that interest us, namely, SOL_SOCKET, IPPROTO_TCP, and IPPROTO_IP. There is one level that we will not be covering in this book, SOL_IRLMP, which deals with infrared sockets. This option first appeared in Windows CE.

Option Level = SOL_SOCKET

Option = SO_DEBUG

Use this option to enable output of debugging information from a Winsock implementation. The mechanism for generating the debug information and the form it takes are beyond the scope of this book.

Option = SO_KEEPALIVE

Use this option on a stream socket (SOCK_STREAM) to enable an application to keep a data stream connected by sending acknowledgment requests at set intervals to the peer. The properties of this option are defined by RFC 1122. One such property is the minimum period between transmissions of keep-alive packets. This period is two hours, which is not always suitable for some applications. However, you can devise a keep-alive scheme in a client-server application pair. Be aware that this option does not work with sockets of the SOCK_DGRAM type. The option is off by default.

> **TIP:** On Windows 95 and 98 platforms, you can configure the duration and frequency in the registry under the KeepAliveInterval and KeepAliveTime keys. However, be aware that these keys are a global setting and will affect all sockets.

CAUTION: When dealing with the registry, as I have found to my chagrin, it is very easy to corrupt the registry, making your system unusable. So be very careful when you edit the registry.

Option = SO_LINGER

Use this option to control what action is required when you call closesocket() on a socket that has data queued waiting to be sent. (To refresh your memory on closesocket() and the SO_LINGER settings, please refer to Chapter 5.) To set the desired behavior on the socket, you would create a record of the type TLinger, which is defined in Winsock2.pas, and point to this record by the *optval* parameter in setsockopt().

```
linger = record
  l_onoff: u_short;              // option on/off
  l_linger: u_short;             // linger time
end;

TLinger = linger;
PLinger = ^linger;
```

To enable SO_LINGER, you should set the *l_onoff* field to a value greater than zero, and then set the *l_linger* field to zero or the desired timeout (in seconds) and call setsockopt().

To enable SO_DONTLINGER (i.e., disable SO_LINGER), you should set the *l_onoff* field to zero and call setsockopt().

TIP: Enabling SO_LINGER with a nonzero timeout on a non-blocking socket is not recommended.

Option = SO_REUSEADDR

By default, you cannot bind a socket (see Chapter 5 for the bind() function) to a local address that is already in use. Sometimes, however, you may want to reuse an address in this way. By calling this option with setsockopt(), you will be able to bind a socket to an existing address. How can this be useful, you ask? Consider a possible scenario. A server has crashed and is terminated by the network administrator. It needs to be restarted immediately, but it cannot do so because the port that was bound to the socket (bound by the bind() function) that the server was using prior to the crash is no longer available, thus causing a loss of service. (To dive into the reasons for this behavior would require us to examine TCP in detail, which is beyond the scope of this book. If you are interested, see Appendix C.) You can avoid this aberrant behavior by using the SO_REUSEADDR option. There is one caveat, however, which is that the remote address must be different from the remote address being used by the previous socket that is using the same local address.

Option = SO_RCVBUF and SO_SNDBUF

By calling this option (SO_RCVBUF and SO_SNDBUF), you can adjust the size of the buffers that the TCP/IP stack uses for receiving or sending data on a socket, respectively. For a stream socket, SO_RCVBUF is the same as the maximum TCP window size.

NOTE: Not all implementations support these options.

Option Level = IPPROTO_TCP

Option = TCP_NODELAY

Use this option to disable the TCP Nagle algorithm and vice versa. The TCP Nagle algorithm, when enabled, reduces the number of small packets sent by a host by buffering the data if there is unacknowledged data already "in flight" or until a full-size packet can be sent. Using this algorithm enhances delivery of data. However, for some applications (like a networking game or simulation), this algorithm can impede performance, and you need to use the TCP_NODELAY option to disable the algorithm. For some background on the Nagle algorithm, consult RFC 896 (see Appendix C).

TIP: Setting this option unwisely can have a significant negative impact on network and application performance. For this reason, unless you know what you are doing, it is usually discouraged.

Option Level = IPPROTO_IP

This level is for use with the IP protocol. You should use this level when you want to either modify the IP header or add a socket to an IP Multicast group. Winsock 1.1 and Winsock 2 both support this level. However, some options in the IPPROTO_IP level in Winsock 1.1 differ from Winsock 2.

Option = IP_OPTIONS

You should use this option if you wish to modify some of the fields in the IP header. For example, you could modify some of these fields to affect the following:

- Security
- Record route
- Time-stamp
- Loose source routing
- Strict source routing

NOTE: Be aware that not all hosts and routers support all of these modifications.

The prototype for the IP header, which is defined in ICMP.PAS, is as follows:

```
PIpHdr = ^TipHdr;
 TIpHdr = packed record // {
            ip_hl;                    // header length
            ip_v;                     // version
            ip_tos : u_char;          // type of service
            ip_len : short;           // total length
            ip_id : u_short;          // identification
            ip_off : short;           // fragment offset field
            ip_ttl;                   // time to live
            ip_p : u_char;            // protocol
            ip_cksum : u_short;       // checksum
            ip_src;                   // source address
            ip_dst : TInAddr;         // destination address
 end;//} IP_HDR, *PIP_HDR, *LPIP_HDR;
```

Option = IP_HDRINCL

If you set this option to TRUE, it will force the sending function, such as send(), to send an IP header ahead of the data that it is sending and will cause the receiving function, such as recv(), to accept the IP header ahead of the data. However, to make this option work, you must fill in the fields of the IP header correctly. This option is only available on Windows 2000 and later. Like raw sockets, the use of this option requires administrative privileges.

Option = IP_TOS

Use this option to indicate the type of service that specifies certain properties of the packet.

Option = IP_TTL

You should use this option to specify the time to live in the TTL field in the IP header. In other words, your goal is to limit the number of routers that the packet can traverse before it is discarded. How does this work? As the router receives the packet, it examines the header and decrements the TTL field by one. When the field becomes zero, the router discards the packet. For example, setting the option to two means that the packet can only do three hops (remember, you start the count from zero) before it dies.

Option = IP_MULTICAST_IF

This option sets the local interface from which you can send multicast data on the local machine. This only makes sense if your machine has more than one network card. We call this machine *multi-homed*.

Option = IP_MULTICAST_TTL

This option has the same effect on data packets as IP_TTL, except it acts on multicast data only.

Option = IP_MULTICAST_LOOP

To prevent loopback of data that you send, set this option to FALSE. The option is TRUE by default.

Option = IP_ADD_MEMBERSHIP

For Winsock 1.1 applications, this is an option you use to add a socket to an IP Multicast group.

Option = IP_DROP_MEMBERSHIP

Call this option to remove the socket from an IP Multicast group.

Option = IP_DONTFRAGMENT

When you set this option to TRUE, it tells the network not to fragment the IP packet. However, if the size of the IP datagram exceeds the maximum transmission unit (MTU), the datagram will die. If the "don't fragment" field in the IP header is set, the network will generate an ICMP error message.

Modifying I/O Behavior

So far, we have described how to query and set the attributes of a socket using options. In the rest of this chapter, we will consider how you might modify the I/O behavior of a socket. There are two functions with which you can modify the I/O behavior—ioctlsocket() and WSAIoctl(). These functions are defined in Winsock2.pas, and their prototypes are listed as follows:

```
function ioctlsocket(s: TSocket; cmd: Longint; var argp: u_long): Integer; stdcall;

function WSAIoctl(s: TSocket; dwIoControlCode: DWORD; lpvInBuffer: LPVOID; cbInBuffer: DWORD;
lpvOutBuffer: LPVOID; cbOutBuffer: DWORD; var lpcbBytesReturned: DWORD; lpOverlapped:
LPWSAOVERLAPPED; lpCompletionRoutine: LPWSAOVERLAPPED_COMPLETION_ROUTINE): Integer; stdcall;
```

As you can see, the WSAIoctl() function, which is part of the Winsock 2 implementation, packs more power and functionality than ioctlsocket(), but we will consider ioctlsocket() first as an introduction.

The first parameter refers to the socket, *s*, with which you want to work. The second parameter, *cmd*, is the command that the function is to execute. The third parameter, *argp*, stores the result of the operation on that socket.

The function ioctlsocket() supports the following commands: FIONBIO, FIONREAD, and SIOCATMARK. These commands are present in all versions of Winsock. Calling ioctlsocket() with the FIONBIO command enables or disables non-blocking mode on a socket (refer to Chapter 5).

The FIONREAD command determines the amount of data that can be read from socket *s*. On a stream socket (e.g., SOCK_STREAM), *argp* points to an unsigned long integer in which ioctlsocket() will store the result. The function returns the size of the data that may be read in a single receive operation, which may not be the same as the total amount of data queued on the socket. On a datagram socket (SOCK_DGRAM), the function returns the size of the first datagram queued on the socket, which might not be the same size as subsequent datagrams.

You should use FIONREAD on a socket that has the socket option SO_OOBINLINE already set. That is, the socket has been set to receive out-of-band data (refer to Chapter 5). When you call ioctlsocket() with this command, the function determines if the data queued on the socket is out-of-band data or normal data. If data to be read is out-of-band, the function will return a TRUE value in the *argp* parameter. (For WSAIoctl(), the Boolean result points to the *lpvOutBuffer* parameter, which we will describe later.)

As we have seen from the implementation, WSAIoctl() is more complex to use than ioctlsocket(). However, at the price of complexity, not only can you use WSAIoctl() either to set or retrieve operating parameters for the specified socket, you can use it to set or retrieve the underlying transport protocol.

The first parameter, *s*, is the socket. The second parameter, *dwIoControlCode,* defines the operational code to execute. For a listing of these commands, refer to Table 6-5. The third and fourth parameters, *lpvInBuffer* and *cbInBuffer*, are input buffers. The first is a pointer to the value to which you pass, and the second is a pointer to the size of the first buffer. Similarly, the fifth parameter, *lpvOutBuffer,* is a pointer to an output buffer that receives the data, and the sixth parameter, *cbOutBuffer*, is the size of the output buffer. *lpcbBytesReturned* indicates the number of bytes returned. Finally, if you set the *lpOverlapped* and *lpCompletionRoutine* parameters to NIL, the function will treat the socket, *s*, as a non-overlapped socket. You should use WSAIoctl() for overlapped I/O operations, in which case the parameters *lpOverlapped* and *lpCompletionRoutine* must point to a valid overlapped structure and a callback routine, respectively.

In addition to the tasks described so far, you can use the ioctlsocket() and WSAIoctl() functions for QOS and multicast applications, which are beyond the scope of this tome.

Table 6-5: Commands for ioctlsocket() and WSAIoctl()

Command	Platform	Function	Input	Output	Winsock Version	Description
SIO_ENABLE_ CIRCULAR_ QUEUEING	Windows 2000 and NT 4.0	WSAIoctl	Boolean	Boolean	2 and later	If the incoming buffer overflows, discard oldest message first.
SIO_FIND_ ROUTE	Not supported	WSAIoctl	TSockAddrIn	Boolean	2 and later	Verifies that a route to the given address exists.
SIO_FLUSH	Windows 2000 and NT 4.0	WSAIoctl	None	None	2 and later	Determines whether OOB data has been read.
SIO_GET_ BROADCAST_ ADDRESS	Windows 2000 and NT 4.0	WSAIoctl	None	TSockAddrIn	2 and later	Returns a broadcast address for the address family of the socket.
SIO_GET_ EXTENSION_ FUNCTION_ POINTER	All Win32 platforms	WSAIoctl	Tguid	Function pointer	2 and later	Retrieves a function pointer specific to the underlying provider.
SIO_CHK_QOS	Windows 2000	WSAIoctl	DWORD	DWORD	2 and later	Sets the QOS attributes for the socket.
SIO_GET_QOS	Windows 2000 and Windows 98	WSAIoctl	None	QOS structure	2 and later	Returns the QOS data structure for the socket.
SIO_SET_QOS	Windows 2000 and Windows 98	WSAIoctl	QOS structure	None	2 and later	Sets the QOS properties for the socket.
SIO_MULTI-POINT_LOOP-BACK	All Win32 platforms	WSAIoctl	Boolean	Boolean	2 and later	Sets or gets whether the multicast data will be looped back to the socket.
SIO_MULTI-CAST_SCOPE	All Win32 platforms	WSAIoctl	Integer	Integer	2 and later	Gets or sets the time to live (TTL) value for multicast data.
SIO_KEEP-ALIVE_VALS	Windows 2000	WSAIoctl	tcp_keepalive structure	tcp_keepalive structure	2 and later	Sets the TCP keepalives active on each connection.
SIO_RCVALL	Windows 2000	WSAIoctl	Unsigned integer	None	2 and later	Receives all packets on the network.
SIO_RCVALL_M CAST	Windows 2000	WSAIoctl	Unsigned integer	None	2 and later	Receives all multicast packets on the network.
SIO_RCVALL_ IGMPMCAST	Windows 2000	WSAIoctl	Unsigned integer	None	2 and later	Receives all IGMP packets on the network.

Command	Platform	Function	Input	Output	Winsock Version	Description
SIO_ROUTING_INTERFACE_QUERY	Windows 2000	WSAIoctl	TSockAddrIn	None	2 and later	Determines whether OOB data has been read.
SIO_ROUTING_INTERFACE_CHANGE	Windows 2000	WSAIoctl	TSockAddrIn	None	2 and later	Sends notification when an interface to a remote socket has changed.
SIO_ADDRESS_LIST_QUERY	All Win32 platforms	WSAIoctl	None	TSOCKET_ADDRESS_LIST structure	2 and later	Returns a list of interfaces to which the socket can bind.
SIO_GET_INTERFACE_LIST	All Win32 platforms	WSAIoctl	None	TINTERFACE_INFO structure	2 and later	Returns a list of local interfaces.
SO_SSL_GET_CAPABILITIES	Windows CE	WSAIoctl	None	DWORD	1.1	Returns the Winsock security provider's capabilities.
SO_SSL_GET_FLAGS	Windows CE	WSAIoctl	None	DWORD	1.1	Returns s-channel-specific flags for the socket.
SO_SSL_SET_FLAGS	Windows CE	WSAIoctl	DWORD	None	1.1	Sets the socket's s-channel-specific flags.
SO_SSL_GET_PROTOCOLS	Windows CE	WSAIoctl	None	SSLPROTOCOLS	1.1	Returns a list of protocols supported by the security provider.
SO_SSL_SET_PROTOCOLS	Windows CE	WSAIoctl	SSLPROTOCOLS	None	1.1	Sets a list of protocols that the underlying provider should support.
SO_SSL_SET_VALIDATE_CERT_HOOK	Windows CE	WSAIoctl	SSLVALIDATE CERTHOOK	None	1.1	Sets the validation function for accepting SSL certificates.
SO_SSL_PERFORM_HANDSHAKE	Windows CE	WSAIoctl	None	None	1.1	Initiates a secure handshake on a connected socket.
SIO_GET_NUMBER_OF_ATM_DEVICES	Windows 2000	WSAIoctl	None	DWORD	2 and later	Returns the number of ATM adapters.
SIO_GET_ATM_ADDRESS	Windows 2000	WSAIoctl	DWORD	TATM_ADDRESS	2 and later	Returns the ATM address for the given device.
SIO_ASSOCIATE_PVC	Windows 2000	WSAIoctl	TATM_PVC_PARAMS structure	None	2 and later	Associates socket with a permanent virtual circuit.
SIO_GET_ATM_CONNECTION_ID	Windows 2000?	WSAIoctl & ioctlsocket	None	TATM_CONNECTION_ID	2 and later	Determines whether OOB data has been read.

Before we give formal definitions of the functions discussed, here is a word about the examples. Unlike the previous chapters where we give examples of using the functions, we have a collection of examples to demonstrate the usage of these functions with different levels and options. Only two levels, SOL_SOCKET and IPPROTO_IP, are used with getsockopt() and setsockopt() functions. For a demonstration on how to use the ioctlsocket() function with the FIONBIO option, please refer to Listing 5-4 in Chapter 5.

function getsockopt *Winsock2.pas*

Syntax

function getsockopt(s: TSocket; level, optname: Integer; optval: PChar; var optlen: Integer): Integer; stdcall;

Description

This function retrieves the current socket option with a socket of any type, in any state, and stores the result in *optval*.

Parameters

s: A descriptor identifying a socket

level: The level at which the option is defined. The supported levels include SOL_SOCKET and IPPROTO_TCP.

optname: The socket option for which the value is to be retrieved

optval: A pointer to the buffer in which the value for the requested option is to be returned

optlen: A pointer to the size of the *optval* buffer

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAEINPROGRESS, WSAEINVAL, WSAENOPROTOOPT, and WSAENOTSOCK.

See Appendix B for a detailed description of the error codes.

See Also

setsockopt, socket, WSAAsyncSelect, WSAConnect, WSAGetLastError, WSASetLastError

Example

See programs EX61, EX62, EX63, and EX64 on the companion CD.

function setsockopt Winsock2.pas

Syntax

function setsockopt(s: TSocket; level, optname: Integer; optval: PChar; optlen: Integer): Integer; stdcall;

Description

This function sets a socket option for the socket.

Parameters

s: A descriptor identifying a socket

level: The level at which the option is defined. The supported levels include SOL_SOCKET and IPPROTO_TCP.

optname: The socket option for which the value is to be set

optval: A pointer to the buffer in which the value for the requested option is supplied

optlen: The size of the *optval* buffer

Return Value

If the function succeeds, it will return a value of zero. Otherwise, it will return SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAEINPROGRESS, WSAEINVAL, WSAENETRESET, WSAENOPROTOOPT, WSAENOTCONN, and WSAENOTSOCK.

See Appendix B for a detailed description of the error codes.

See Also

bind, getsockopt, ioctlsocket, socket, WSAAsyncSelect, WSAEventSelect

Example

See programs EX65, EX66, EX67, and EX68 on the companion CD.

function ioctlsocket Winsock2.pas

Syntax

ioctlsocket(s: TSocket; cmd: Longint; var argp: u_long): Integer; stdcall;

Description

This function retrieves operating parameters associated with the socket, independent of the protocol and communications subsystem.

Parameters

s: A descriptor identifying a socket

cmd: The command to perform on the socket *s*

argp: A pointer to a parameter for *cmd*

Return Value

If the function succeeds, it returns a value of zero. Otherwise, it returns SOCKET_ERROR. To retrieve the specific error code, call the function WSAGetLastError(). Possible error codes are WSANOTINITIALISED, WSAENETDOWN, WSAEINVAL, WSAEINPROGRESS, WSAENOTSOCK, and WSAEFAULT.

See Appendix B for a detailed description of the error codes.

See Also

getsockopt, setsockopt, socket, WSAAsyncSelect, WSAEventSelect, WSAIoctl

Example

See Listing 5-4 in Chapter 5.

function WSAIoctl Winsock2.pas

Syntax

WSAIoctl(s: TSocket; dwIoControlCode: DWORD; lpvInBuffer: LPVOID; cbInBuffer: DWORD; lpvOutBuffer: LPVOID; cbOutBuffer: DWORD; var lpcbBytesReturned: DWORD; lpOverlapped: LPWSAOVERLAPPED; lpCompletionRoutine: LPWSAOVERLAPPED_COMPLETION_ROUTINE): Integer; stdcall;

Description

The function controls the mode of a socket.

Parameters

s: A handle to a socket

dwIoControlCode: The control code of the operation to perform

lpvInBuffer: A pointer to the input buffer

cbInBuffer: The size of the input buffer

lpvOutBuffer: A pointer to the output buffer

cbOutBuffer: The size of the output buffer

lpcbBytesReturned: The number of actual bytes of output

lpOverlapped: An address of the WSAOVERLAPPED record (ignored for non-overlapped sockets) *lpCompletionRoutine*: A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets)

Return Value

If the function succeeds, it will return zero. If the function fails, it will return a value of SOCKET_ERROR. To retrieve the error code, call the function WSAGetLastError(). Possible error codes are WSAENETDOWN, WSAEFAULT, WSAEINVAL, WSAEINPROGRESS, WSAENOTSOCK, WSAEOPNOTSUPP, and WSAEWOULDBLOCK.

See Appendix B for a detailed description of the error codes.

See Also

getsockopt, ioctlsocket, setsockopt, socket, WSASocket

Example

See program EX 69 on the companion CD.

Summary

In this short chapter, you discovered socket options and socket commands that affect the behavior of a socket and I/O operations on a socket, respectively. This chapter concludes our coverage of Winsock 2. It's now time to explore TAPI.

Part 2

Fundamentals of TAPI Programming

by Alan C. Moore

Chapter 7

Introduction to TAPI

Telephony has a magic ring (bad pun), just like telepathy; no wonder a recent television commercial in the United States played with these two similar-sounding words. The same sense of magic that must have struck early telephone users—to be able to talk with someone in another town or country—struck computer users when they were first able to place calls from their desktop machines without lifting the phone handset from its cradle. Of course, this capability depends on certain hardware and software. Before the advent of the computer sound card and the voice modem, such telephony was impossible. With the addition of these hardware components and software to communicate with them, telephony became not only possible, but a standard element of the Windows desktop.

What can you do with telephony under Windows? As we'll see, you can create applications that support a wide range of sophisticated communications features and services using a telephone line. You can provide support for speech and data transmission with a variety of terminal devices. Your applications can also support complex connection types and call management scenarios, such as conference calls, call waiting, and voice mail. Covering all of these is beyond the scope of this book, but we'll cover the most basic ones.

What makes it possible for you to write a Windows application that supports all of these features in various types of calls? In a word, TAPI (the Telephony Application Programming Interface). Described in the Microsoft TAPI Help file (referred to as the TAPI Help file from this point on), TAPI "provides a device-independent interface for carrying out telephony tasks." As such, it is an integral part of Microsoft's Windows Open Systems Architecture (WOSA) just like the Winsock API we discussed earlier. It provides transparent support for a variety of communications hardware.

That Help file states that TAPI "simplifies the development of telephonic applications by hiding the complexities of low-level communications programming." As the documentation points out, TAPI accomplishes this task by abstracting telephony services, making them independent of the underlying telephone network as well as the way the computer is connected to the switch and phone set. As we'll see, these connections to the switch may be established in a variety of ways. They can be connected directly from the user's workstation or through a server on a LAN (local area network). Importantly, as the documentation stresses, "regardless of their nature, telephony devices and connections are handled in a single, consistent manner, allowing developers to apply the same programming techniques to a broad range of communications functions." We'll discuss a number of specific instances in this chapter. First, let's take a superficial look at what you can do using TAPI.

With TAPI you can create full-featured communications applications or add telephony support to database, spreadsheet, word-processing, and personal information management applications. In fact, in any situation when you need to send or receive data through a telephone network, TAPI is usually the answer. Some of the functionality you can provide users of your applications includes the ability to:

- Connect directly to a telephone network instead of having to use a specialized communications application
- Automatically dial telephone numbers
- Send or receive documents such as files, faxes, and electronic mail
- Retrieve data from news or information services
- Place and manage conference calls
- Manage voice mail
- Automate the processing of incoming calls by using caller ID
- Support collaborative computing over telephone lines

In this chapter, we'll outline the development of Windows telephony and examine the evolution of TAPI from its origins. We'll provide information about some of the basic issues involved and lay the basis for understanding the functions, structures, and constants that are part of TAPI. However, we will not be able to discuss many of the more advanced line functions or any of the phone API functions.

An Historical Review

TAPI originated in a manner similar to many other Windows APIs. Various vendors were already developing support for telephony, but using their own particular approaches. Of course, these approaches were proprietary and generally not compatible with each other. That was not the Windows way, following WOSA. In the beginning—in the early 1990s (BT, or Before TAPI)—telephony equipment was expensive, usually DOS-based, and supported by proprietary software. Herman D'Hooge, an Intel engineer, is probably the single most important person responsible for the creation of TAPI. He and his company recognized the need for a single telephony API early on. There was a similar interest in such an API at Microsoft, and the two companies, Microsoft and Intel, decided to work together. Toward that goal, D'Hooge met one of Microsoft's telephony engineers, Toby Nixon, with whom he worked to create the first version of TAPI. That initial TAPI draft was presented to a group representing over 40 companies involved in telephony.

As you might imagine, the initial and limited draft went through major revisions as feedback was received from these interested parties. The first public release, TAPI 1.0, was presented in 1993 at a telephony conference in Dallas, Texas. At the same time, another important piece of the puzzle was being developed at Intel—the Telephony Service Provider Interface (TSPI). What TAPI was for the applications developer, TSPI was for the telephony hardware provider.

Next came the testing phase, beginning in 1994, during which various vendors tested their equipment with TSPI and TAPI. In the meantime, Microsoft had decided that TAPI would be a part of each of its operating systems: TAPI 1.3 was supported by Windows 3.x; the next version, TAPI 1.4, shipped with Windows 95; TAPI 2.0 shipped with Windows NT 4.0. This latest incarnation of the Windows NT family supports TAPI 3.0. This book uses the latest Project JEDI header translation, which supports version 3.0. However, we cover only the most basic functions here.

The World of Telephony Applications

As we've discussed, telephony applications enable people to access telecommunications systems from their computers, allowing them to manage voice calls and data-transfer operations. You can use TAPI to provide such functionality within any application, and it applies to various types of hardware, including voice modems and cable modems, among others.

In short, TAPI allows your application to provide the local machine with access to a telephone network, with all of its features and limitations. As a developer, it is your job to provide the user interface, taking advantage of the functionality in Windows that Delphi and TAPI provide. It also sends messages for many of its events, so, with a little work, you can use a memo control to provide feedback on every step in placing or receiving a call, use drag and drop to let the user send files or faxes over the telephone line, enable the user to initiate a conference call (also using drag and drop to select the names of the participants), and support other sophisticated scenarios.

As you have probably deduced, TAPI provides your application with access to a variety of telephone network services. Although these services may use different technologies to establish calls and transmit voice and data, TAPI makes these service-specific details transparent to applications. That's what WOSA has intended to accomplish. With TAPI you can create applications that can take advantage of any available service without including service-specific code in your application.

Historically, most telephone connections in the world have been of a type referred to as POTS, or Plain Old Telephone Service. Figure 7-1 shows a typical POTS environment. POTS calls are generally transmitted digitally, except while in the local loop. The latter is the portion of the telephone network that exists between the individual telephone and the telephone company's central switching office. It is within this loop that things get a bit complicated. Human speech from a household telephone is generally transmitted in analog format. However, the digital data from a computer must first be converted to analog by a modem. The situation remains complex, but progress is taking place. For example, digital networks are gradually replacing analog in the local loop.

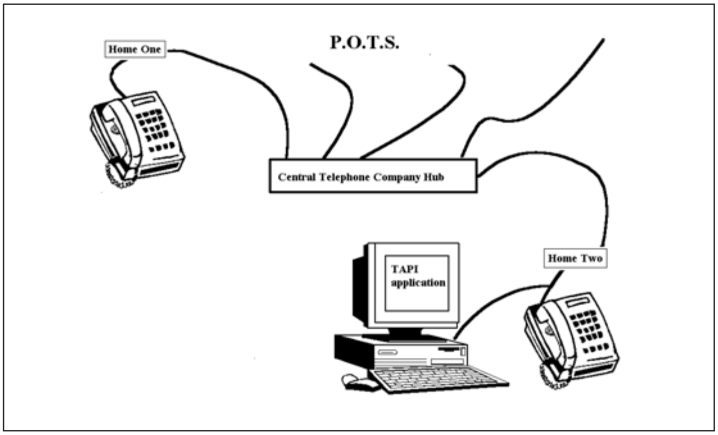


Figure 7-l: A typical POTS environment

Fortunately, using TAPI for POTS is straightforward because of POTS' comparative simplicity. It normally uses only one type of information (such as data or voice) per call, supports one channel per line, and so on. Most current uses for TAPI are related to POTS, and most telephony programmers use TAPI exclusively for POTS applications.

Does this mean that when you use TAPI you are restricted to POTS? Of course not! With TAPI, you can make connections over various types of networks. You'll recall our discussion of Integrated Services Digital Network (ISDN) when we presented Winsock. You can also use TAPI to access ISDN networks. Such networks provide all of the advantages of ISDN over POTS since they:

- Are totally digital
- Are less error prone
- Provide faster data transmission, with speeds up to 128 kilobytes per second (Kbps) on basic service

- Provide from 3 to 32 channels for simultaneous transmission of voice and data
- Are based on a recognized international standard, that of Integrated Services Digital Network, or ISDN

Let's take a more detailed look at these advantages.

ISDN networks are completely digital and do not have to hassle with the analog-to-digital conversions required under POTS with a modem. Because data travels from one end of an ISDN network to the other in digital format, error rates are lower than with the analog transmission that takes place on POTS. It is also faster; at the time of publication, it has up to 128 Kbps on Basic Rate Interface (BRI-ISDN) standard lines and is considerably higher on Primary Rate Interface (PRI-ISDN) standard lines. How does this compare with modems? Today's maximum dial-up modem data rates (as of publication) are generally 56 Kbps or less, depending on the quality of the local loop, which varies with the locality.

As we look to the future, we can foresee many advantageous developments. As ISDN connections become more common, we'll be able to send data to the recipient while simultaneously having a phone conversation with that person. Depending on its transmission rate, each ISDN line can provide a minimum of three channels (two for voice or data and one strictly for data or signaling information) and as many as 32 channels for simultaneous, independently operated transmission of voice and data.

How do BRI-ISDN lines differ from PRI-ISDN lines? According to the specification described in the TAPI Help file, BRI-ISDN lines provide two 64 Kbps "B" channels and one 16 Kbps "D" channel. So-called B channels carry voice or data, while so-called D channels carry signaling information or packet data. PRI-ISDN lines differ by locality. In the United States, Canada, and Japan, the PRI-ISDN lines have 23 64 Kbps B channels and one 64 Kbps D channel. In European countries, the PRI-ISDN lines have 30 B channels and two D channels.

What about other types of networks? You'll be pleased to learn that you can use TAPI with other digital networks, such as T1/E1 and Switched 56 service. The latter enables local and long-distance telephone companies to provide signaling at 56 Kbps over dial-up telephone lines. This service is quickly becoming available throughout the United States and in many other countries. It should be noted that to use it, you must have special equipment. Additionally and not surprisingly, its connection capabilities are limited to calls to other facilities that have the proper equipment. Still, its high speed and reasonable pricing make it a good choice for many data communications needs (Switched 56 is used for data calls only).

TAPI's versatility doesn't end here either! You can use it with other services, such as with CENTREX, with digital Private Branch Exchanges (PBXs), and with key systems. CENTREX provides a number of special network services (such as conferencing) but does not require any special equipment. This is possible because the user pays for the use of telephone company equipment over regular telephone lines. Best of all, programming a CENTREX or PBX application using TAPI is virtually the same as programming a POTS application. In other words, regardless of the environment, there's no need to make changes to an application's source code. Finally, TAPI can be used with various types of hardware, voice modems, cable modems, DSL, and ISDN lines.

The Elements of a Telephony System

To understand the programming structure for TAPI, you need to understand the Windows Open Systems Architecture (WOSA) model that we mentioned above and discussed at length when we introduced Winsock in Chapter 1. We show the main steps in the communication process between the elements in Figure 7-2 and will now explain how it works. First, your application will make one or more function calls to TAPI.PAS to request the desired functionality. TAPI.PAS has the job of providing an interface—or means of communication—with the TAPI dynamic-link library (DLL). That DLL, in turn, will make calls to TAPI32 DLL, which will then forward those application requests to the telephony service for processing. Then, the DLL will communicate with TAPISRV.EXE, which has the task of implementing and managing the TAPI functions. Finally, TAPISRV.EXE will communicate with one or more telephony service providers (drivers) who will control the hardware and do the actual work. These service providers are

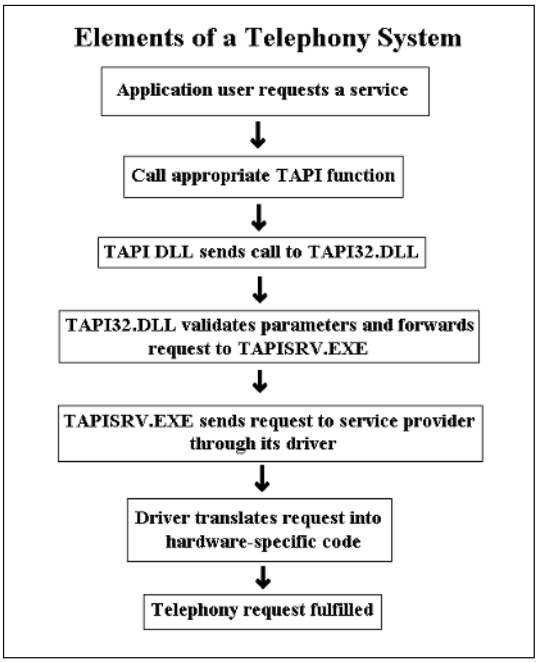


Figure 7-2: The main steps in the communication process between the elements of a telephony system

also DLLs and their task is to carry out low-level, device-specific actions needed to complete telephony tasks through hardware devices, such as fax boards, ISDN cards, telephones, and modems. It's important to note that applications link to and call functions only in the TAPI DLL; they never call the service providers directly.

When an application calls a TAPI function, the TAPI dynamic-link library validates and takes note of the parameters of the function and forwards it to TAPISRV.EXE. This telephony service application processes the call and routes a request to the appropriate service provider. To receive requests from TAPISRV, a service provider must implement the Telephony Service Provider Interface (TSPI) we mentioned earlier. Of course, a service provider has the option of providing different levels of the service provider interface: basic, supplementary, or extended. On the one hand, a simple service provider might provide basic telephony service, such as support for outgoing calls through a Hayes-compatible modem. On the other hand, a custom service provider written by a third-party vendor might provide a full range of support, including advanced features like conference calls. There is a vast array of possibilities.

TIP: There is one golden rule regarding the behavior of service providers: You can install any number of service providers on a computer provided that the service providers do not attempt to access the same hardware device(s) at the same time. The installation program will generally associate specific hardware with a specific service provider.

Some service providers have the ability to access multiple devices. In some instances, a user will need to install a device driver along with the service provider. Most modern computers handle this kind of situation automatically, distributing CD-ROMs or other media that include, install, and register needed drivers and components. Often computer makers also distribute their own telephony applications that take full advantage of the particular hardware (usually a voice modem) and its drivers.

For the developer writing applications to run on various machines, there are TAPI functions that determine which services are available on the given computer; further, TAPI can determine which service providers are available and provide information about their respective capabilities. In this way, any number of applications can request services from the same service provider; TAPI will take care of the job of managing access to the service provider.

As we've seen, service providers have a vital role to play in the world of TAPI; they provide the hardware-independent interface or communications link to various classes of device. This WOSA-based structure simplifies development by letting programmers treat devices with similar properties in a similar manner. What are these device classes? They include such expected items as telephones, modems, and even multimedia devices. Fortunately, your application never needs to know which service provider controls which device.

This device class-centric structure helps make TAPI extensible because the framework is flexible enough to classify and provide support for new equipment. That's good news because both hardware and software are being developed and enhanced continually. Our perception of features changes also. Features that were considered optional just a few years ago quickly become standard through customer demands and vendor competition. As long as an application does not depend on optional features, it can use any of the available services to carry out its telephony tasks. As we've discussed and outlined in Figure 7-2, an application must access the many different services through TAPI alone; TAPI assumes the important responsibility to translate the requests from the application into the required protocols and interfaces.

Nature and Structure of TAPI

TAPI is a huge application programming interface. Take a look at TAPI.pas, developed by Project JEDI. Much of this API involves two device classes: line device and phone device, with the former being paramount. Likewise, the API defines two main sets of functions and messages, one for line devices and one for phone devices. We will concentrate on the line API in this introductory work, explaining all of its basic functions in detail.

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

A service provider can provide some rather sophisticated and powerful functionality to users. For example, it might be possible for an application to request the combination of multiple channels in a single call to give that call wider bandwidth. As we just pointed about, with POTS, your application must generally assign one channel per line. But with ISDN, a line's channels are dynamically allocated when an application makes or answers a call. Because these channels have identical capabilities and are interchangeable, your application need not identify which channel is to be used in a given function call. Channels are owned and assigned by the service provider for the line device in a way that is transparent to applications. The channel management method is abstract and eliminates any need to introduce the naming of channels by TAPI.

We've briefly discussed line devices and will dive into that essential topic with earnest beginning in Chapter 8, "Line Devices and Essential Operations." There is also another telephony device type—the phone device—that we'll mention briefly. Conceptually, just as a line device class is an abstraction of a physical line device, the phone device class represents a device-independent abstraction of a telephone set. There is one important difference: While you can assume that the basic line device functions will always be available to you, you cannot make any such assumption about phone devices. We will not discuss phone devices in detail in this tome.

The TAPI architecture includes some truly beneficial features, not the least of which is the way it treats line and phone devices as being independent of each other. In other words, you can use a phone (device) without using an associated line, and you can use a line (device) without using a phone. As a result, service providers that fully implement this line/phone independence can offer uses for these line and phone devices not defined or even considered by traditional telephony protocols. The TAPI Help file provides several interesting examples. For example, a person can use the handset of the desktop's phone as a waveform audio device for voice recording or playback, perhaps without the switch's knowledge that the phone is in use. In such an implementation, lifting the local phone handset need not automatically send an off-hook signal to the switch. Of course, for some functionality, it might be necessary to also relate to other APIs, such as (in this case, possibly) the Waveform API. For more information on this API, see Alan C. Moore's *The Tomes of Delphi: Win32 Multimedia API*. The capabilities of a service provider are limited by the capabilities of the hardware and software used to interconnect the switch, the phone, and the computer. We'll briefly consider some of those limitations and future possibilities.

Today, computer telephony is characterized by both current limitations and future possibilities. Some of TAPI's more advanced capabilities require that an application be able, for example, to retrieve data from telephones. Even today, most telephones cannot be connected directly to computers to control speech calls and thus are currently incapable of supporting telephony functions beyond the passive role they play in POTS. If some predications come to fruition, future users will be able to install and configure telephone sets like other peripheral computer devices. These telephone sets will no doubt be accompanied by new types of cards that will control the flow of information between the computer and the telephone itself. Other future possibilities include client-server configurations that will allow users to take advantage of telephony services by connecting over a local area network (LAN) to a server that has a specific type of board and associated software installed. Figure 7-3 shows a possible telephony configuration over a LAN.

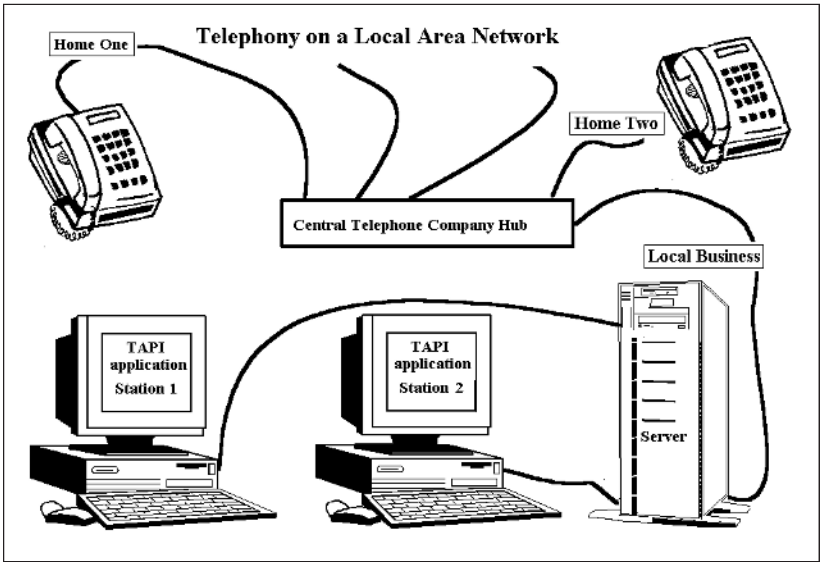


Figure 7-3: A possible telephony configuration over a LAN

Media Stream

You'll recall that when we discussed Winsock, we explained how that technology allowed us to transmit byte streams. In TAPI, we talk about media streams rather than byte streams. A *media stream* is simply the information exchanged over a telephone call. That information can represent a variety of media. While TAPI allows you to control the various line and phone devices available, including discovering the type of media a specific line can handle, it does have one limitation: It does not give you access to the content of the media stream.

How can you get that level of access? To manage a media stream, you must use other Win32 functions from the APIs that support areas such as Communications, Wave Audio, or Media Control Interface (MCI). (The latter two APIs are discussed in *The Tomes of Delphi: Win32 Multimedia API*.) Consider an application that manages fax or data transmission. Such an application would use the TAPI functions to control and monitor the line over which data bits were being sent but would use the Communications functions to transmit the actual data. Similarly, an application that recorded conversations or played greeting messages would need to rely on the Wave API.

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

Varieties of Physical Connections

There's more than one way in which lines and phones can be connected between a desktop computer and a telephone network. The two common paradigms are called *phone-centric connections* and *computer-centric connections*. We'll mention several specific configurations that could be supported by a service provider with the caveat that some of the telephone hardware needed to implement some configurations may not be widely available at the time of publication. Table 7-1 shows the various connection types described in the TAPI Help file, beginning with the two most common ones.

Table 7-I: Three types of connections

Connection Type	Description
Phone-centric	Uses a single POTS line in which the computer is connected to the switch through the desktop phone set. These phone sets typically connect to the computer through one of its serial ports. When an application requests an action, the corresponding service provider sends telephony commands (often based on the Hayes AT command set) over a serial connection to the telephone. Under this limited configuration, there is generally only line control, and the computer does not have access to the media stream.
Computer-centric	Uses a computer add-in card or external box connected to both the telephone network and the phone set. The service provider may easily integrate modem and fax functions. It can also use the telephone as an audio I/O device.
BRI-ISDN	Similar in many ways to the computer-centric connection but allows for using the two B channels in a variety of line configurations

See the TAPI Help file for the various models it suggests and tips on how to work with these models.

We'll discuss some of the other possibilities in more detail here, as they may be applicable to your development needs.

Telephony can also be used on local area network servers. Such a server might have multiple telephone line connections. It would have to be able to support a variety of TAPI operations initiated at any of the client computers connected to it. As usual, these requests would be forwarded over the LAN to the server. The server would support third-party call control between itself and the switch to implement the client's call-control requests. An advantage of this model is that it offers a lower cost per computer for call control if the LAN is already in use, and it also offers a reduced cost for media stream access if shared devices are installed in the server. Those shared devices might include voice digitizers, fax and/or data modems, or interactive voice response cards. Although digitized media streams can be carried over the LAN, real-time transfer of media may be problematic with some LAN technologies due to inconsistent throughput.

A LAN-based host can be connected to the switch using a switch-to-host link. As with other tasks in a LAN environment, TAPI operations invoked at any of the client computers will be forwarded over the LAN to the host. In response, the host would then use a third-party switch-to-host link protocol to implement the client's call-control requests. Note that it is also possible for you to connect a private branch exchange (PBX) directly to a LAN and integrate the server functions into the PBX. Microsoft outlines the following sub-configurations in the TAPI Help file:

- One that provides personal telephony functionality to each by associating the PBX line with the computer (on a desktop) as a single line device with one channel.
- One that allows applications to control calls on other stations by modeling each third-party station as a separate line device. (In a PBX, a station is anything to which a wire leads from the PBX.)
- One that sets all third-party stations as a single line device with one address (phone number) assigned to it per station.

In the first sub-configuration, each client computer would have one line device available. In the second, where one workstation can control calls on other machines, your application must first open each line it wants to manipulate or monitor. This setup is particularly important if you're using a small number of lines; however, it could involve a good deal of overhead if a large number of lines is involved. In the third sub-configuration, only one device would be opened, with that device providing monitoring and control of all addresses (all stations) on the line. In this case, to originate a call on any of these stations, the application must specify only the station's address to the function that makes the call. No extra line-opening operations are required. However, this modeling implies that all stations have the same line device capabilities, although their address capabilities could be different.

For TAPI applications, the computers used need not be desktop computers. Such applications can also run on laptops and other portable computers connected to the telephone network over a wireless connection. In fact, we used a laptop as one of the test computers for the code in this book.

TAPI's capabilities are expanding with every new version, but you must have the right hardware to take full advantage. If you're using a shared telephony connection in which the computer's connection is shared by other telephony equipment, you must ensure that there is some control over the use of that equipment, since neither the application nor the service provider can assume that there are no other active devices on the line.

Levels of Telephony Programming Using TAPI

In the previous sections, we discussed parts of the Telephony Application Programming Interface, with particular emphasis on the Line API and to a lesser extent the Phone API. But we haven't told the entire story. These two APIs represent the low-level services of TAPI. As in other APIs (such as Multimedia), there are also a few high-level services that we'll discuss presently. Similar to those other APIs, the high-level services make the programmer's work easier, while the low-level services provide additional functionality and flexibility. The high-level services comprise the handful of functions that belong to the Assisted Telephony services we'll discuss in Chapter 10, "Placing Outgoing Calls."

What is the best way to conceptualize TAPI's line capabilities? Generally, these capabilities fall into two general groups: Basic Telephony and Supplementary Telephony. There are also Extended Telephony services that are service-provider specific. Basic Telephony services constitute a minimal subset of the Win32 telephony specification, one that corresponds roughly to the features of POTS (Plain Old Telephone Service). The specification requires that these features be available with every TAPI-capable line device. To put it another way, every service provider must support these Basic Telephony services. Of course, this is mainly an issue for hardware manufacturers and the device driver developers. The good news for applications developers is that processes that use only these basic functions should work with any TAPI service provider.

We've mentioned some fairly exotic TAPI implementations that are presented in the documentation. These are all beyond the scope of this book, so we should come back to Earth. Even today, many applications remain within the world of services provided by Basic Telephony. The functions that are part of Basic Telephony are shown in Table 7-2. These functions, which we'll discuss in detail in the remainder of this tome, fall into two categories: synchronous and asynchronous. The former (synchronous) group of functions will always return a result to the application immediately; the latter (asynchronous) functions will indicate their completion in a REPLY message to the application. The functions also belong to various categories depending on the type of task they perform.

Table 7-2: Basic Telephony functions

Function	Meaning	Group
lineInitializeEx	A synchronous function that initializes the TAPI line abstraction for use by the invoking application	TAPI initialization and shutdown
lineShutdown	A synchronous function that shuts down an application's use of the line TAPI connection	TAPI initialization and shutdown
lineNegotiateAPIVersion	A synchronous function that allows an application to negotiate a version of TAPI to use	Line version negotiation

Function	Meaning	Group
lineGetDevCaps	A synchronous function that returns the capabilities of a given line device	Line status and capabilities
lineGetDevConfig	A synchronous function that returns configuration of a media stream device	Line status and capabilities
lineGetLineDevStatus	A synchronous function that returns current status of the specified open line device	Line status and capabilities
lineSetDevConfig	A synchronous function that sets the configuration of the specified media stream device	Line status and capabilities
lineSetStatusMessages	A synchronous function that specifies the status changes for which an application wants to be notified	Line status and capabilities
lineGetStatusMessages	A synchronous function that returns an application's current line and address status message settings	Line status and capabilities
lineGetID	A synchronous function that retrieves a device ID associated with the specified open line, address, or call	Line status and capabilities
lineGetIcon	A synchronous function that allows an application to retrieve an icon for display to the user.	Line status and capabilities
lineConfigDialog	A synchronous function that causes the provider of the specified line device to display a dialog box that allows the user to configure parameters related to the line device	Line status and capabilities
lineConfigDialogEdit	A synchronous function that displays a dialog box allowing the user to change configuration information for a line device (Version 1.4)	Line status and capabilities
lineGetAddressCaps	A synchronous function that returns the telephony capabilities of an address	Addresses
lineGetAddressStatus	A synchronous function that returns current status of a specified address	Addresses
lineGetAddressID	A synchronous function that retrieves the address ID of an address specified using an alternate format	Addresses
lineOpen	A synchronous function that opens a specified line device for providing subsequent monitoring and/or control of the line	Opening and closing line devices
lineClose	A synchronous function that closes a specified opened line device	Opening and closing line devices
lineTranslateAddress	A synchronous function that translates between an address in canonical format and an address in dialable format (See Chapter 10 for an explanation of canonical and dialable formats.)	Address formats
lineSetCurrentLocation	A synchronous function that sets the location used as the context for address translation	Address formats
lineSetTollList	A synchronous function that manipulates the toll list	Address formats
lineGetTranslateCaps	A synchronous function that returns address translation capabilities	Address formats
lineGetCallInfo	A synchronous function that returns mostly constant information about a call	Call states and events
lineGetCallStatus	A synchronous function that returns complete call status information for the specified call	Call states and events
lineSetAppSpecific	A synchronous function that sets an application-specific field of a call's information structure	Call states and events

Function	Meaning	Group
lineRegisterRequestRecipient	A synchronous function that registers or de-registers an application as a request recipient for the specified request mode	Request recipient services. These functions are used only in support of assisted telephony. (See Chapters 10 and 11)
lineGetRequest	A synchronous function that gets the next request from the Telephony DLL	Request recipient services
lineMakeCall	Makes an outbound call and returns a call handle for it—asynchronous	Making calls
lineDial	Dials (parts of one or more) dialable addresses—asynchronous	Making calls
lineAnswer	Answers an inbound call—asynchronous	Answering inbound calls
lineSetNumRings	A synchronous function that indicates the number of rings after which inbound calls are to be answered	Toll saver support
lineGetNumRings	A synchronous function that returns the minimum number of rings requested with lineSetNumRings()	Toll saver support
lineSetCallPrivilege	A synchronous function that sets an application's privilege to the privilege specified	Call privilege control
lineDrop	Disconnects a call or abandons a call attempt in progress—asynchronous	Call drop
lineDeallocateCall	A synchronous function that deallocates the specified call handle	Call drop
lineHandoff	A synchronous function that hands off call ownership and/or changes an application's privileges to a call	Call handle manipulation
lineGetNewCalls	A synchronous function that returns call handles to calls on a specified line or address for which an application does not yet have handles	Call handle manipulation
lineGetConfRelatedCalls	A synchronous function that returns a list of call handles that are part of the same conference call as the call specified as a parameter	Call handle manipulation
lineTranslateDialog	A synchronous function that displays a dialog box allowing the user to change location and calling card information (Version 1.4)	Location and country information
lineGetCountry	A synchronous function that retrieves dialing rules and other information about a given country (Version 1.4)	Location and country information

What if you're supporting something more sophisticated, such as a company's PBX phone system? Such a system could have internal capabilities greatly exceeding the external (POTS) system to which it is connected for communicating with the outside world. To support the greater functionality of the PBX, you'll need the functions belonging to Supplementary Telephony. These functions are listed in Table 7-3 and deal with issues such as bearer modes, monitoring various call aspects (media, digits, and tones), media control actions, digit and tone manipulations, and advanced call operations. The latter group includes features like call acceptance, rejection, redirecting, holding, forwarding, parking, pickup, and completion. We do not go into a detailed discussion of these extended TAPI functions in this book. However, since we're currently providing an overview, we will provide a complete one with all of the Supplementary Telephony functions in Table 7-3.

Table 7-3: Supplementary Telephony functions

Function	Meaning	Group
lineSetCallParams	A synchronous function that requests a change in the call parameters of an existing call	Bearer mode and rate
lineMonitorMedia	A synchronous function that enables or disables media mode notification on a specified call	Media monitoring
lineMonitorDigits	A synchronous function that enables or disables digit detection notification on a specified call	Digit monitoring and gathering
lineGatherDigits	A synchronous function that performs the buffered gathering of digits on a call	Digit monitoring and gathering
lineMonitorTones	A synchronous function that specifies which tones to detect on a specified call	Tone monitoring
lineSetMediaControl	A synchronous function that sets up a call's media stream for media control	Media control
lineSetMediaMode	A synchronous function that sets the media mode(s) of the specified call in its LINECALLINFO structure	Media control
lineGenerateDigits	A synchronous function that generates inband digits on a call	Generating inband digits and tones
lineGenerateTone	A synchronous function that generates a given set of inband tones on a call	Generating inband digits and tones
lineAccept	An asynchronous function that accepts an offered call and starts alerting both caller (ringback) and called party (ring)	Call accept and redirect
lineRedirect	An asynchronous function that redirects an offering call to another address	Call accept and redirect
lineDrop	An asynchronous function that drops or disconnects the specified call. Your application may specify user-to-user information to be transmitted as part of the call disconnecting process.	Call reject
lineHold	An asynchronous function that places the specified call on hard hold	Call hold
lineUnhold	An asynchronous function that retrieves a held call	Call hold
lineSecureCall	An asynchronous function that secures an existing call from interference by other events such as call waiting beeps on data connections	Making calls
lineSetupTransfer	An asynchronous function that prepares a specified call for transfer to another address	Call transfer
lineCompleteTransfer	An asynchronous function that transfers a call that was set up for transfer to another call, or enters a three-way conference.	Call transfer

Function	Meaning	Group
lineBlindTransfer	An asynchronous function that transfers a call to another party	Call transfer
lineSwapHold	An asynchronous function that swaps the active call with the call currently on consultation hold	Call transfer
lineSetupConference	An asynchronous function that prepares a given call for the addition of another party	Call conference
linePrepareAddToConference	An asynchronous function that prepares to add a party to an existing conference call by allocating a consultation call that can later be added to the conference call that is placed on conference hold	Call conference
lineAddToConference	An asynchronous function that adds a consultation call to an existing conference call	Call conference
lineRemoveFromConference	An asynchronous function that removes a party from a conference call	Call conference
linePark	An asynchronous function that parks a given call at another address	Call park
lineUnpark	An asynchronous function that retrieves a parked call	Call park
lineForward	An asynchronous function that sets or cancels call forwarding requests	Call forwarding
linePickup	An asynchronous function that picks up a call that is alerting at another number or picks up a call alerting at another destination address and returns a call handle for the picked-up call (linePickup can also be used for call waiting)	Call pickup
lineReleaseUserUserInfo	An asynchronous function that releases user-to-user information, permitting the system to overwrite this storage with new information (Version 1.4.)	Sending information to remote party
lineSendUserUserInfo	An asynchronous function that sends user-to-user information to the remote party on the specified call	Sending information to remote party
lineCompleteCall	An asynchronous function that places a call completion request	Call completion
lineUncompleteCall	An asynchronous function that cancels a call completion request	Call completion
lineSetTerminal	An asynchronous function that specifies the terminal device to which the specified line, address events, or call media stream events are routed	Setting a terminal for phone conversations
lineGetAppPriority	A synchronous function that retrieves handoff and/or Assisted Telephony priority information for an application (Version 1.4)	Application priority
lineSetAppPriority	A synchronous function that sets the handoff and/or Assisted Telephony priority for an application (Version 1.4)	Application priority
lineAddProvider	A synchronous function that installs a telephony service provider (Version 1.4)	Service provider management
lineConfigProvider	A synchronous function that displays the configuration dialog box of a service provider (Version 1.4)	Service provider management

Function	Meaning	Group
lineRemoveProvider	A synchronous function that removes an existing telephony service provider (Version 1.4)	Service provider management
lineGetProviderList	A synchronous function that retrieves a list of installed service providers (Version 1.4)	Service provider management
lineAgentSpecific	An asynchronous function that allows the application to access proprietary handler-specific functions of the agent handler associated with the address (Version 2.0)	Agents
lineGetAgentActivityList	An asynchronous function that obtains the list of activities from which an application selects the functions an agent is performing (Version 2.0)	Agents
lineGetAgentCaps	An asynchronous function that obtains the agent-related capabilities supported on the specified line device (Version 2.0)	Agents
lineGetAgentGroupList	An asynchronous function that obtains the list of agent groups into which an agent can log into on the automatic call distributor (Version 2.0)	Agents
lineGetAgentStatus	An asynchronous function that obtains the agent-related status on the specified address (Version 2.0)	Agents
lineSetAgentActivity	An asynchronous function that sets the agent activity code associated with a particular address (Version 2.0)	Agents
lineSetAgentGroup	An asynchronous function that sets the agent groups into which the agent is logged into on a particular address (Version 2.0)	Agents
lineSetAgentState	An asynchronous function that sets the agent state associated with a particular address (Version 2.0)	Agents
lineProxyMessage	A synchronous function that is used by a registered proxy request handler to generate TAPI messages (Version 2.0)	Proxies
lineProxyResponse	A synchronous function that indicates completion of a proxy request by a registered proxy handler (Version 2.0)	Proxies
lineSetCallQualityOfService	An asynchronous function that requests a change of the quality of service parameters for an existing call (Version 2.0)	Quality of service
lineSetCallData	An asynchronous function that sets the CallData member of the LINECALLINFO structure (Version 2.0)	Miscellaneous
lineSetCallTreatment	An asynchronous function that sets the sounds the user hears when a call is unanswered or on hold (Version 2.0)	Miscellaneous
lineSetLineDevStatus	An asynchronous function that sets the line device status (Version 2.0)	Miscellaneous

Table 7-4 lists all of the functions in the Phone API. These functions are not covered in this book.

Table 7-4: Phone functions

Function	Meaning	Group
phoneInitializeEx	A synchronous function that initializes the Telephony API phone abstraction for use by the invoking application	TAPI initialization and shutdown
phoneShutdown	A synchronous function that shuts down the application's use of the phone Telephony API	TAPI initialization and shutdown
phoneNegotiateAPIVersion	A synchronous function that allows an application to negotiate an API version to use	Phone version negotiation
phoneOpen	A synchronous function that opens the specified phone device, giving the application either owner or monitor privileges	Opening and closing phone devices
phoneClose	A synchronous function that closes a specified open phone device	Opening and closing phone devices
phoneGetDevCaps	A synchronous function that returns the capabilities of a given phone device	Phone status and capabilities
phoneGetID	A synchronous function that returns a device ID for the given device class associated with the specified phone device	Phone status and capabilities
phoneGetIcon	A synchronous function that allows an application to retrieve an icon for display to the user	Phone status and capabilities
phoneConfigDialog	A synchronous function that causes the provider of the specified phone device to display a dialog box that allows the user to configure parameters related to the phone device	Phone status and capabilities
phoneSetHookSwitch	An asynchronous function that sets the hookswitch mode of one or more of the hookswitch devices of an open phone device	Hookswitch devices
phoneGetHookSwitch	A synchronous function that queries the hookswitch mode of a hookswitch device of an open phone device	Hookswitch devices
phoneSetVolume	An asynchronous function that sets the volume of a hookswitch device's speaker of an open phone device	Hookswitch devices
phoneGetVolume	A synchronous function that returns the volume setting of a hookswitch device's speaker of an open phone device	Hookswitch devices
phoneSetGain	An asynchronous function that sets the gain of a hookswitch device's mic of an open phone device	Hookswitch devices
phoneGetGain	A synchronous function that returns the gain setting of a hookswitch device's mic of an opened phone device	Hookswitch devices
phoneSetDisplay	An asynchronous function that writes information to the display of an open phone device	Display
phoneGetDisplay	A synchronous function that returns the current contents of a phone's display	Display
phoneSetRing	An asynchronous function that rings an open phone device according to a given ring mode	Ring
phoneGetRing	A synchronous function that returns the current ring mode of an opened phone device	Ring
phoneSetButtonInfo	An asynchronous function that sets the information associated with a button on a phone device	Buttons

Function	Meaning	Group
phoneGetButtonInfo	A synchronous function that returns information associated with a button on a phone device	Buttons
phoneSetLamp	An asynchronous function that illuminates a lamp on a specified open phone device in a given lamp lighting mode	Lamps
phoneGetLamp	A synchronous function that returns the current lamp mode of the specified lamp	Lamps
phoneSetData	An asynchronous function that downloads a buffer of data to a given data area in the phone device	Data areas
phoneGetData	A synchronous function that uploads the contents of a given data area in the phone device to a buffer	Data areas
phoneSetStatusMessages	A synchronous function that specifies the status changes for which the application wants to be notified	Status
phoneGetStatusMessages	A synchronous function that returns the status changes for which the application wants to be notified	Status
phoneGetStatus	A synchronous function that returns the complete status of an open phone device	Status

In the preceding tables we presented the full telephony services, divided into the Line API and the Phone API. These APIs can be used to implement powerful telephonic functionality in applications. On the other hand, TAPI's high-level programming interface, Assisted Telephony, can be used to add minimal (but useful) telephonic functionality to non-telephony applications. In Chapter 10, we'll examine Assisted Telephony along with other more involved ways of placing outgoing telephone calls. However, before we do that, we need to discuss the basic issues of initializing TAPI, configuring TAPI, and dealing with TAPI messages.

Summary

In this chapter, we have examined the history and the definition of TAPI, including the manner in which it fits into WOSA. We have explored the range of TAPI applications, current and future relationships to hardware, media streams, and a variety of other related topics. Finally, we have provided a summary of all of the TAPI functions, many of which will be discussed in the remaining chapters. Now we are ready to discuss specifics and begin working with TAPI code.

Chapter 8

Line Devices and Essential Operations

The largest part of TAPI is the part that deals with so-called "lines." We'll refer to this application programming interface here as the Line API. This part of TAPI is huge and grows larger with each new version. It comprises all of the constants, structures, and functions that begin with "line." We'll devote the remainder of this book to an examination of a major portion of this API. In this chapter, we'll provide an overview followed by a detailed discussion of the essential line operations, such as initializing, opening, closing, and configuring line devices. In the next chapter, we'll discuss the closely related topic of setting up a callback function to handle telephony messages from Windows. In that chapter, we'll also provide a detailed discussion of those messages. We'll devote the final two chapters to information about providing users with the ability to place calls and answer calls, respectively. As we did in Part I in our discussion of Winsock, we'll provide a detailed reference on the line constants, structures, and functions that support this functionality in this and the remaining chapters.

We'll begin by considering an important question: What exactly is a line device? The TAPI Help file defines a line device as "a physical device such as a fax board, a modem, or an ISDN card that is connected to an actual telephone line." Line devices have a crucial function: They provide support for a wide range of telephony functionality, enabling applications to send and/or receive information to/from a telephone or telephone network. A line device is not a physical line; rather, it provides a logical representation or abstraction of such a physical line device.

In this chapter, we'll begin by providing an overview of the various stages in working with telephony devices. Then we'll show how to initialize telephony devices. As indicated above, we'll postpone our discussion of message handling until the next chapter. We'll briefly discuss the different levels of line functionality and provide a detailed discussion on configuring line devices. We'll then discuss the process of opening line devices, determining capabilities, and working with media modes. We'll conclude the chapter with an in-depth reference to the functions and structures that support this and related functionalities.

Stages in Working with Telephony

In terms of the process involved, working with telephony is similar to working with many other computer technologies, including multimedia. First you must establish a connection with the hardware through its driver(s). You accomplish this task by initializing TAPI as we do in the process of initializing our TAPI class. After you've established a connection to TAPI (and its DLL), you must follow the stages in the process that are shown in Figure 8-1. They can be summarized in this manner: First you must check the capabilities of the TAPI devices on a particular computer and properly configure a device to carry out the tasks you want to accomplish. Next, you must open the device you've identified, setting up a callback mechanism so your application can deal with messages sent back from Windows to that application. (Again, we'll discuss the callback mechanism and the specific messages in detail in Chapter 9, "Handling TAPI Line Messages.") Then the real fun begins—you must provide a range of useful telephony tasks for your users. These can range from the simple placing and answering of calls to setting up conference calls (if your system supports this advanced feature). Finally, you must shut things down properly. In this chapter, we'll go through the four stages in some detail. In the second half of the chapter, we'll expose all of the functions and structures that support these steps.

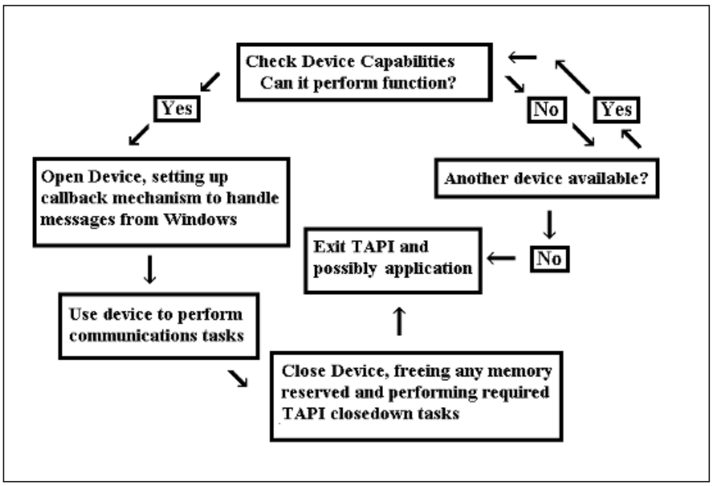


Figure 8-1: Stages in working with a communications device

Before you can discover TAPI capabilities, you must first establish a relationship with TAPI itself. Applications using TAPI versions 1.4 and earlier must use the lineInitialize() function to accomplish this. That function is now obsolete but is still included in newer TAPI versions for backward compatibility. More recent TAPI versions use the lineInitializeEx() function. Both functions share five parameters that store information on the TAPI connection, including the address of a callback function that handles the Windows messages. The new function includes two additional parameters. One indicates the highest TAPI version it is designed to support and the other points to a LINEINITIALIZEEX-PARAMS structure. That structure contains additional parameters that are used to communicate between the TAPI and your application.

Three Notification Mechanisms

When you call the lineInitializeEx() function to establish the communication link we've been discussing, you must select one of three notification mechanisms. Any of these mechanisms will allow your application to receive information about telephony events. The three mechanisms that TAPI provides are Hidden Window, Event Handle, and Completion Port. We'll discuss each mechanism beginning with the means of invoking it, its basic qualities, and the issues or constraints associated with it.

To select a particular mechanism, you must specify its associated constant in the *dwOptions* field of its final parameter (a LINEINITIALIZEEXPARAMS structure) as follows:

- **Hidden Window mechanism** (the only one available to TAPI 1.x applications): Use the LINEINITIALIZEEXOPTION_USEHIDDENWINDOW constant.
- **Event Handle mechanism**: Use the LINEINITIALIZEEXOPTION_USEEVENT constant.
- **Completion Port mechanism**: Use the LINEINITIALIZEEXOPTION_USECOMPLETION PORT constant.

Each of these mechanisms behaves in a somewhat different way. As its name implies, the Hidden Window mechanism instructs TAPI to create a hidden window to which all messages will be sent. The Event Handle mechanism instructs TAPI to create an event object on behalf of your application, returning a handle to the object in the *hEvent* field in the LINEINITIALIZEEXPARAMS structure. Finally, the Completion Port mechanism instructs TAPI to send a message to your application whenever a telephony event occurs using the PostQueued-CompletionStatus() function. Note that it is your responsibility to set up a completion port. TAPI will send a message to the completion port that your application specifies in the *hCompletionPort* field of the LINEINITIALIZEEX-PARAMS structure. The message will be tagged with the completion key that the application specified in the *dwCompletionKey* field in LINEINITIALIZEEX-

PARAMS. In the TapiIntf.pas unit, we have demonstrated the first two of these mechanisms.

There are also issues or possible constraints that come up with each of these mechanisms. We will mention some of the warnings specified in the TAPI Help file. If you use the Hidden Windows mechanism, you must provide a means of handling messages in a queue and you must poll that queue regularly to avoid delaying processing of telephony events. Delphi handles much of this automatically in its handling of Windows messages. Still, you need to write the callback routine so that it responds to each TAPI event. In the TapiIntf.pas unit, we define several new messages to inform the calling application of telephony states. Be aware that when you call the lineShutdown() function, TAPI will automatically handle the details of shutting things down, destroying the hidden window in the process.

With the Event Handle mechanism, your application should not attempt to manipulate a TAPI event directly, such as by calling SetEvent(), ResetEvent(), CloseHandle(), or similar Windows functions. If you ignore this warning, your application could likely manifest strange and unpredictable behavior. Instead of using any of the previously mentioned Windows functions, your application should simply wait for this event using other Windows functions, such as WaitForSingleObject() or MsgWaitForMultipleObjects().

As we've seen, the Completion Port mechanism requires you to perform additional chores. Importantly, you must first create the completion port using the Windows CreateIoCompletionPort() function. While we do not use this approach with TAPI in our sample code, we did discuss the CreateIoCompletionPort() function and its use with Winsock in Chapter 5. Once you have set up the mechanism, your application will retrieve events using the GetQueuedCompletionStatus() function. When GetQueuedCompletionStatus() returns, it will send the specified *dwCompletionKey* to your application. TAPI will write this value to the DWORD pointed to by the *lpCompletionKey* parameter, with a pointer to a LINEMESSAGE structure returned to the location pointed to by *lpOverlapped*. After your application has processed the event, you must release the memory used to contain the LINEMESSAGE structure. Because your application created the completion port itself (unlike the objects that TAPI creates for you automatically), you must also close it, but be careful not to close the completion port until after you have called the lineShutdown() function. For additional information on these three methods, see the TAPI Help file.

TAPI Line Support—Basic and Extended Capabilities

As we mentioned already, systems can have one or more line devices. And TAPI provides you with a straightforward means through which to refer to individual ones. To accomplish this, you should simply enumerate the line device IDs. These will always have a range from zero to one less than the value of *dwNumDevs*. For convenience, in our TapiIntf unit, we store the value of *dwNumDevs* in a property of the main class so that it is available whenever we need it.

When working with TAPI, you should not assume that a particular line device is capable of performing a specific TAPI function, unless, of course, it is one of the basic ones. To make this determination, you should first query the device's capabilities by calling the lineGetDevCaps() and lineGetAddressCaps() functions. Again, valid address IDs for the latter function range from zero to one less than the number of addresses returned by lineGetDevCaps(). Let's explore TAPI capabilities further.

TIP: When working with TAPI, never assume that a particular line device will be capable of performing every TAPI function; if you want to include any functionality beyond the basic line functions, check the device's capabilities using lineGetDevCaps().

Determining Capabilities and Configuring TAPI

If your application needs to use functionality beyond that of Basic Telephony, you must first determine the line device's capabilities as we mentioned earlier. Bear in mind that these capabilities can vary considerably depending upon such factors as network configuration (client versus client-server), specific hardware installed, service-provider software (especially drivers), and the telephone network to which the computer is connected. As stated already, you can safely assume that all of the capabilities of Basic Telephony will be available, but you can't assume anything beyond that.

To perform correctly, your application must find the proper TAPI version to use. To accomplish this, you must call the lineNegotiateAPIVersion() function to determine the API version and the lineNegotiateExtVersion() function to determine the extension version to use. In our example code, we store these values in properties of our TAPI class for later use.

The lineGetDevCaps() function will provide you with the telephony capabilities available on a given line device. This information will be returned in a data structure of the type LINEDEVCAPS. You should use this information to make programming decisions. You could also display it to the user. Figure 8-2 shows a dialog box from one of our sample applications that demonstrates how to provide such a summary. If your application is feature-rich and designed to work under a variety of telephony environments, you should be careful to disable any extended functionality that you find is not supported.

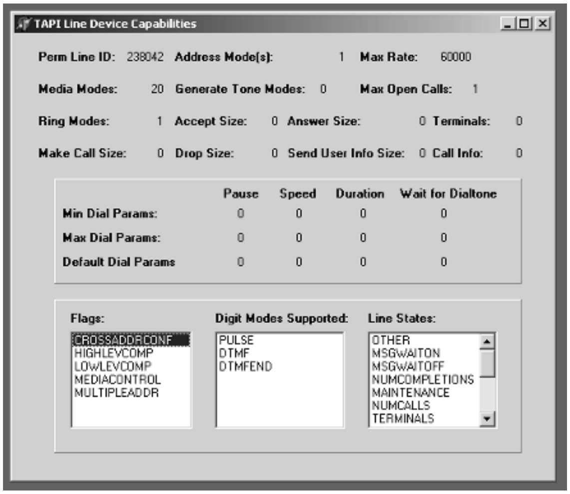


Figure 8-2: TAPI line device capabilities

Be aware that since new versions of TAPI include increased capabilities, this (LINEDEVCAPS) and similar structures will tend to get larger with each new version. Let's take a look at this important structure in more detail and explain how to deal with some of the more difficult issues.

If you need to work with device-specific extensions, you should use the Dev-Specific (*dwDevSpecificSize* and *dwDevSpecificOffset*) variably sized area of this data structure.

Note that older applications (using older TAPI versions, especially 1.x) don't include this field as part of the LINEADDRESSCAPS structure. Variable structures can be very tricky in TAPI, since they often vary in size from one version to the next. Here's yet another reason for determining and taking account of the TAPI version you're working with. After you call the lineGetAddressCaps() function, you should check the *dwAPIVersion* parameter to get this information from TAPI. This is the proper way to handle version-sensitive situations.

When calling either lineGetDevCaps() or lineGetAddressCaps(), it is quite possible to pass a size that's too small in the *dwTotalSize* parameter. When this happens, you'll get an error of LINEERR_STRUCTURETOOSMALL. You can handle the situation easily in code by testing for this specific error and then reallocating memory to the particular structure, either LINEDEVCAPS or LINEADDRESSCAPS. You can get the amount of memory needed by examining the *dwNeededSize* parameter of the respective structure. As we've discussed, the reason why this issue comes up is that the size of these structures varies with different versions of TAPI.

There are also issues related to service providers. A new service provider (which may or may not support a new TAPI version) has the important responsibility to examine the TAPI version passed to it. If the TAPI version used by the application is less than the highest version supported by the provider, the service provider must not fill in those fields that are not supported in older TAPI versions, since these fixed fields would fall in the variable portion of the older structure. Additionally, new applications must be cognizant of the TAPI version negotiated and should not attempt to examine the contents of fields in the fixed portion beyond the original end of the fixed portion of the structure for that negotiated TAPI version.

Configuring TAPI

Are there resources to provide users with the ability to view and edit configuration information? Yes! TAPI provides two functions for this purpose, lineConfigDialog() and lineConfigDialogEdit(). Both functions cause the service provider to display a modal dialog box (see Figure 8-3) that allows the user to configure parameters related to the specified line. But there is a significant difference between the two functions: The lineConfigDialog() function changes the configuration information immediately (dangerous in some situations), while the lineConfigDialogEdit() function saves the information in a structure that can be used to update the configuration later when you call to the lineSetDevConfig() function.

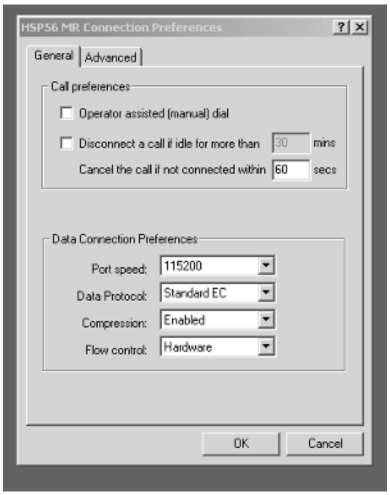


Figure 8-3: A modal dialog box

If necessary, you can use either function's *lpszDeviceClass* parameter in order to show a specific sub-screen of the full configuration information available. Some of the common strings that can be used in this parameter are "comm." and "tapi/line." You can find a complete list in the TAPI Help file. The former (comm) would be appropriate if the line supports the venerable Comm API. In that case, the provider would display information related specifically to comm. The latter string (tapi/line) would be more appropriate for the TAPI line functions we are discussing here.

TAPI's VarString

An interesting feature of LineConfigDialogEdit() is the structure it uses to store configuration data. This structure, called a VarString, is defined in the TAPI Help file and the original Project JEDI translation of TAPI.pas file (note that I added an additional field to point to the variable data in this structure):

```
PVarString = ^TVarString;
  varstring_tag = packed record
    dwTotalSize,
    dwNeededSize,
    dwUsedSize,
    dwStringFormat,
    dwStringSize,
    dwStringOffset: DWORD;
    // Modified by Alan C. Moore: new field, next line added
    data : array [0..0] of byte;
  end;
```

The first three fields of this structure—*dwTotalSize*, *dwNeededSize*, and *dwUsedSize*—are common to many structures in Microsoft APIs. They are so common, in fact, that they are sometimes omitted in the documentation. The first, *dwTotalSize*, indicates the total size (in bytes) allocated to the data structure. Generally, it is your responsibility to allocate sufficient memory, at least for the fixed portion of the data structure. However, like similar structures in this and other APIs, there is a variable portion of this structure whose size may not be known in advance. The reason is that different vendors will include different configuration information of different sizes.

How should you deal with this variable data part of the structure? A common approach is to guess the size of the variable portion and allocate memory equal to the fixed size and the estimated maximum variable size. You must also set *dwTotalSize* to this exact size. Further, you should initialize the bytes in the structure to 0. (Setting the *dwStringFormat* field is probably not needed but was added during the debugging phase in an attempt to correct a problem that we will discuss presently.) Here is code from TAPIIntf.pas that accomplishes this, where FDeviceConfig is a pointer to a VarString structure:

```
if FDeviceConfig=Nil then
  begin
    FDeviceConfig := AllocMem(SizeOf(VarString)+1000);
    FillChar(FDeviceConfig^, SizeOf(VarString)+1000, 0);
    FDeviceConfig.dwTotalSize := SizeOf(VarString)+1000;
    FDeviceConfig.dwStringFormat := STRINGFORMAT_BINARY;
  end;
```

The *dwNeededSize* field holds the size (in bytes) needed to hold all the returned information. The *dwUsedSize* field holds the size (in bytes) of the portion of the structure that contains useful information. These latter two fields are set after calling a function that fills the structure with configuration information.

Before calling lineConfigDialogEdit(), you need to call lineGetDevConfig() to retrieve initial configuration data. This configuration data will always be specific to the media stream associated with the specified line device. For a data modem (indicated by using the datamodem string when calling the lineGetDevConfig() and lineConfigDialogEdit() functions), the user could specify properties like data rate, character format, modulation schemes, and error control protocol settings. Whenever you open a line with the LINEMAPPER constant, you should call the lineGetID() function afterward to retrieve the actual ID number of the specific device associated with a line. You can then use that ID number to call other functions. In this case, you would definitely need it when you call lineGetDevConfig() to get the configuration information.

Once you have retrieved and stored configuration information in a VarString, you can use that information to restore the configuration if the user of your application wishes to later. You should call the lineSetDevConfig() function to return to the earlier configuration settings. We'll demonstrate all of these techniques in our discussion of these functions later in this chapter.

Again, the exact format of the data contained within the variable portion of the VarString structure is device-specific. In addition, and most important, this data is for TAPI's internal use only! Your application should never attempt to access the data directly or manipulate it; that task will be handled by TAPI using the various functions we will discuss. The data must be stored intact and/or copied intact as we have shown in our sample code. Since the data is specific to a single device and its associated media stream, you should not attempt to pass it to any other device, even one of the same device class. Now that we've discussed configuring line devices, we'll examine the process of line initialization, establishing a communications link with TAPI.

Line Initialization—Making a Connection with TAPI

Laying the foundation to perform even the most basic telephony operation is an involved process. First, you need to initialize TAPI itself. Next, you need to negotiate one or two TAPI versions for your application to use, taking into account the different TAPI versions and versions of service providers by different vendors. After that, you can examine each of the TAPI devices present on a computer, determining its capabilities, or you can take a shortcut and use the LINEMAPPER constant to find a device that meets your needs. Finally, having performed the preliminary steps, you may open the line device that has the capabilities you need. These steps are summarized in Figure 8-4. We'll examine them in some detail now.

As we just stated, your application needs a connection with TAPI to use any of TAPI's basic or supplementary line functions. You need such a connection even to call the configuration functions we discussed above. The TTapiInterface

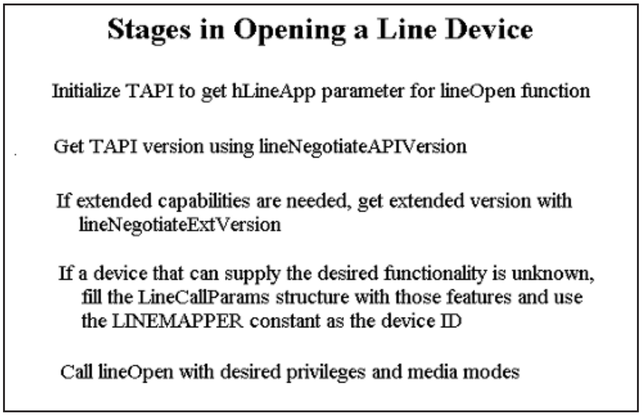


Figure 8-4

class we develop in this book handles this chore automatically. The sample application, TAPIInitTest.dpr (see Figure 8-5), tests the initialization routines in the TTapiInterface class.

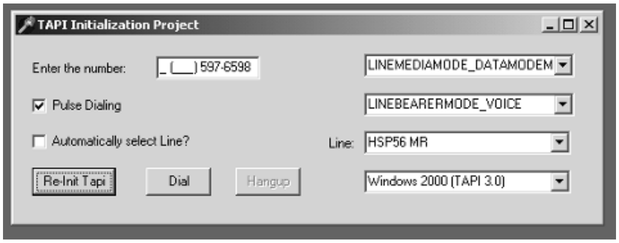


Figure 8-5: Testing the initialization routines

The connection with TAPI is essential for your application to receive telephony messages from Windows. Your application can establish this connection using either the lineInitializeEx() or the phoneInitializeEx() function. We will discuss the former function in detail in the second half of this chapter. Certain parameters of these functions allow your application to specify the message notification mechanism your application desires to use. We will provide specific information about the lineInitializeEx() function later in this chapter.

Neither initialization function—lineInitializeEx() nor phoneInitializeEx()—is device specific. The same can be said of the anachronistic functions they replace—lineInitialize() and phoneInitialize(). When your application calls any of these functions, TAPI does not interact with any particular device (line or phone) or an abstraction of any device. Rather, TAPI begins by simply setting up the telephony environment. These tasks include loading the TAPI DLL, loading TAPISRV.EXE, and loading the device drivers specified in the Windows registry. Devices include telephony service providers and any needed components. TAPI must also establish a communication link (as we have described above) between itself and the calling application during its initialization process. TAPI will consult the Windows registry to retrieve information about registered telephony applications. If TAPI determines that the registry contains an invalid entry, it will return an INIFILECORRUPT error. When this error occurs (regardless of the initialization function that triggered it), you should notify the user so that he or she may attempt to resolve the problem. The user may need to rebuild the registry or a portion thereof. That's the bad news. The good news is that the user can often invoke the Telephony Control Panel dialog box to accomplish this task rather than having to edit the registry manually.

Another possible error, LINEERR_NODRIVER, will occur if the telephony driver was not installed properly. Usually this means that your application cannot locate a critical element, such as a previously installed service provider or a required component (often a device driver) of the service provider. When your application encounters this error, you should again advise the user to correct the problem, this time using the Driver Setup capabilities of the Telephony Control Panel.

As we have stated, your application must call a line initialization function prior to calling any line functions. What happens if you call an initialization function more than once (for example, to specify a different message notification scheme)? You can expect to get an error, as we'll discuss presently. So, before calling lineInitializeEx() for a different purpose, you must first call lineShutdown(). Note that both lineInitializeEx() and lineShutdown() and the corresponding phone functions operate synchronously. They always return a success or failure indication rather than an asynchronous Request ID.

TIP: If you need to call lineInitializeEx() a second time (to establish a different kind of connection), be sure to call lineShutdown() first to close the existing connection.

Upon successful completion, the lineInitializeEx() function will return two essential pieces of information to your application: an application handle and the number of available line devices. In our TAPI class, we store both of these values for later use by other functions. An application handle represents the application's connection to TAPI. For TAPI, this value identifies the calling application. TAPI functions that use line or call handles (explained later in this chapter) do not require an application handle. This is because an application can determine its application handle from the specified line, phone, or call handle.

As we mentioned briefly, the lineInitializeEx() function also returns the number of line devices available to an application through TAPI. Device identifiers (device IDs) are used to identify line devices. As in other Windows APIs, these device IDs are zero-based positive integers ranging from zero to one less than the number of line devices. For example, if lineInitializeEx() reports that there are two line devices in a system, the valid line device IDs would be 0 and 1; if it reported five, the valid line device IDs would be 0, 1, 2, 3, and 4.

As with most other APIs, it is equally important to properly shut TAPI down (in Winsock we called WSACleanUp() for this purpose). Once your application is finished calling TAPI's line functions, you must call the lineShutdown() function, passing its application handle (the one set when you called lineInitialize() or lineInitializeEx()) to that function. This enables TAPI to terminate an application's usage of its functionality and free any resources assigned to that application. If you neglect to do this, it is possible that some resources may not be freed. We do this automatically when we call the destructor for our TAPI class.

Another critical issue is version control. As time goes on, there have been, and will continue to be, new versions of TAPI (as of publication, version 3, associated with Windows 2000, is the most recent), of applications that use TAPI, and of service providers that relate to TAPI. These new versions will almost certainly define new features, new functions to access those features, and new fields in data structures to hold new information.

Among other things, TAPI version numbers are helpful in providing guidance in the interpretation of various data structures. Over time, many of these data structures have grown to support new functionality. Examine TAPI.pas, included on the companion CD-ROM accompanying this book. You'll notice that the Project JEDI folks who translated the TAPI C header file have indicated the new fields added in TAPI versions 2.0, 2.2, and 3.0. In our TAPI wrapper class, we provide two initializations, one for TAPI 2.2 (that supports Windows 9.x) and one for TAPI 3.0 (for Windows 2000 and beyond). Here are the routines that accomplish this, the first initializing TAPI for Windows 9.x and the second for Windows 2000:

```
procedure TTapiInterface.InitToWin9X;
begin
  FCountryCode := 0;
  FVersion     := $00020002;
  FExtVersion  := $00000000;
  fNumLineDevs := 0;
  FAPIVersion  := $00020002;
  FLoVersion   := $00010004;
  FHiVersion   := $00020002;
end;

procedure TTapiInterface.InitToWin2000;
begin
  FCountryCode := 0;
  FVersion     := $00030000;
  FExtVersion  := $00030000;
  fNumLineDevs := 0;
  FAPIVersion  := $00030000;
  FLoVersion   := $00010004;
  FHiVersion   := $00030000;
end;
```

Let's Negotiate

Given the possibility of different application versions, TAPI versions, and vendor service-provider versions, how does TAPI allow for optimal interoperability? Once again, TAPI provides a simple solution. It uses a two-step version negotiation mechanism in which an application agrees on two different version numbers. The first one is the version number for Basic and Supplementary Telephony. This negotiation result is referred to as the *TAPI version.* The second is for provider-specific extensions, if any, and is referred to as the *extension version*. These versions must be "agreed upon" by all of the players—your application, TAPI itself, and the service provider for each line device. Not surprisingly, the format of the data structures and data types used by TAPI's basic and supplementary features is defined by the TAPI version, while the extension version determines the format of the data structures defined by the vendor-specific extensions.

Let's take a detailed look at this two-step version negotiation process. First, you must negotiate the TAPI version number, obtaining the extension ID that is associated with any vendor-specific extensions supported on the device. Second, you may need to negotiate the extension version. Be aware that there are certain situations in which you should skip the process of version negotiation. If your application does not use any TAPI extensions, you can certainly skip this second negotiation. In this case, extensions will not be activated by the service provider. If your application does require extensions, and the service provider's extensions (the extension ID) are not recognized by your application, you should skip the negotiation for extension version as well. However, in our TAPI class, we negotiate both. Note that each vendor will define its own set of legal (recognized) versions for each set of extension specifications it supports.

We've discussed the negotiation process, but we have not discussed the functions used to negotiate the TAPI version and the extension version. The first function is lineNegotiateAPIVersion(). In addition to returning an appropriate TAPI version, it also retrieves the extension ID. If no extensions are supported, this number will be set to zero. When you call this function, you must provide a range of TAPI versions with which your application is compatible. With this information, TAPI will then negotiate with the line's service provider to determine which TAPI version range it supports. Then TAPI selects a version number (usually but not always the highest version number) in the overlapping version range supplied by your application, the TAPI DLL, and the service provider.

Now for the second negotiation, which is the one that's not always used. If you need to use available extended functionality, you must call the lineNegotiateExtVersion() function to negotiate the extension version. This process is similar to the primary negotiation phase we just discussed. In this case, your application will include, as parameters to the function call, the already agreed-upon TAPI version and the extension version range it supports. TAPI will pass this information to the service provider for the line. In turn, the service provider will check the TAPI version and the extension version range against its own and will select the appropriate extension version number, if one exists.

These two functions—lineNegotiateAPIVersion() and lineNegotiateExtVersion()—lay an important foundation for other functions, including one we'll be considering soon, lineGetDevCaps(). When you call this latter function to retrieve device capabilities for a particular line, those results will reflect the results of version negotiation. These line device capabilities will be consistent with both the TAPI version and the line's device-specific extension capabilities. Note that your application must specify both of these version numbers when it opens a line. This enables your application, the TAPI DLL, and the service provider to agree upon a specific TAPI version or versions as we discussed above. Again, if you don't need to use device-specific extensions, the extension version should be set to zero.

Sometimes multiple applications will open the same line device. When this happens, the first application to open the device has a special status. That application will select the TAPI version(s) for all future applications that may also use that particular line device; note that service providers do not support multiple versions simultaneously. If your application must open multiple line devices, you should follow the advice in the TAPI Help file and operate all of the line devices under the same TAPI version.

Determining Capabilities

As promised, we'll now explore the process of determining a line device's capabilities. To determine such capabilities, you must use the lineGetDevCaps() function. Remember, before calling this function, your application must go through the process we just described above—you must negotiate the TAPI version number to use and, if desired, the extension version to use. (These are included among this function's parameters). As we've seen, the TAPI and extension version numbers are those under which TAPI and the service provider will operate. This way, your application will know in advance the functionality available to it. In the TapiInterface class we develop, we store many types of capabilities as Boolean properties.

What if the version ranges do not overlap? In that case, the application, TAPI, or service-provider versions will be incompatible and TAPI will return an error. In our sample code, we show how to display this information for the user. If this function does complete successfully, it will return information about the line capabilities in its last parameter, a pointer to a variably sized structure of type

LINEDEVCAPS. This structure will be filled with the line device's capabilities data. You may use this information in making programming decisions or display it for the user.

A single line can include a number of addresses. That number of addresses will be indicated in one of the fields of the LINEDEVCAPS structure. Similar to line IDs, address IDs range from zero to one less than the returned number. Address capabilities can vary just as line capabilities vary. To discover these address capabilities, you should call the lineGetAddressCaps() function for each available dwDeviceID/dwAddressID combination.

Opening a Line Device

Now that we have laid the proper foundation, we are ready for the final step, which is actually opening a line device. Once you have obtained a line device's capabilities, your application must actually open that line device before it can access its telephony functions. Keep this in mind: As defined by TAPI, a line device is an abstraction of a line. Therefore, opening a line and opening a line device can be thought of as interchangeable. When an application has opened a line device successfully, it will receive a handle for it. The application can then perform any of the common tasks on that line, including accepting inbound calls, placing outbound calls, or monitoring and logging call activities on the line. Usually an application that has successfully opened a line device can use that device to make an outbound call. The exception is a situation in which that line supports only inbound calls.

To open a line device for any purpose, you should call the lineOpen() function. Of course, when your application is finished using the line device, you should close it by calling the lineClose() function. You can call the lineOpen() function in one of two ways: with a device ID or without a device ID.

Using the first method, call the lineOpen() function with a specific line device, including its line device ID in the *dwDeviceID* parameter. This will open that specific line device. If an application is interested in handling inbound calls, it will generally use this approach so that the application will be aware of the specific line that wants to handle inbound calls. When a line device has been opened successfully, your application will receive a handle representing the open line.

Using the second method, your application must specify the properties it wants from a line device and use the value LINEMAPPER instead of a specific line-device ID as the parameter for the lineOpen() function. The function will open any available line device that supports the properties you specified. Of course, opening a line in this manner may fail. However, if it is successful, you can determine the line device ID by calling the lineGetID() function and specifying the handle (*lphLine*) to the open line device returned by the call to lineOpen().

There are some cases in which a line cannot be opened. Fortunately, these are sometimes temporary in nature. You can generally determine the reason by examining the error code returned by the lineOpen() function.

> **NOTE:** The example code on the companion CD always checks these error codes and reports any problem.

Let's discuss some of the possible errors. A result of LINEERR_ALLOCATED indicates that the line could not be opened because of a persistent condition, such as a serial port having been opened in exclusive mode by another process. A result of LINEERR_RESOURCEUNAVAIL indicates a dynamic resource over-commitment. Such an over-commitment may be transitory, such as during the process of monitoring media modes or tones. In such a case, changes in these activities by other applications may make it possible for your application to reopen the line within a short period of time.

LINEERR_REINIT is another important error. It always indicates that your application has made an illegal attempt to reinitialize TAPI. As we mentioned earlier, you are not permitted to do that! Sometimes such an attempt could be made inadvertently, perhaps the result of adding or removing a TSP (Telephony Service Provider). When this happens, TAPI will reject calls to the lineOpen() function, returning the LINEERR_REINIT error until the last application (using TAPI) shuts down its usage of TAPI (by calling lineShutdown()). At that point, you may begin the process again with a new configuration and call lineInitializeEx(). All of the error codes are listed in our reference to this function at the end of the chapter.

Give Me Your ID

Closely related to the lineOpen() function is the lineGetID() function. Earlier we discussed the LINEMAPPER constant, which locates an appropriate device given a list of requested services. Given the current line handle, the lineGetID() function will retrieve a line device ID—the real line device ID of the opened line. You can also use this function to retrieve the device ID of a phone device or media device. The latter might include such device classes as Waveform, MIDI, phone, and line. Any of these might be associated in some way with a call, an address, or a line. Once you have retrieved the ID, you can use it with the appropriate API (such as Wave, MIDI, Phone, or Line) to select the corresponding media device associated with the specified call.

Specifying Media Modes

In opening a line, one important issue we need to discuss concerns the media mode(s) it will support. This is particularly important if your application supports inbound calls or wants to be the target of call handoffs on a line. The media modes that a particular line can support are specified in the lineOpen() function's *dwMediaModes* parameter. When you call this function, it will register your application as having an interest in monitoring calls or receiving ownership of calls that are of the specified media mode(s). As usual, you must accomplish this by including certain flags in this parameter, as follows: If an application is interested in monitoring calls, it should specify LINECALLPRIVILEGE_MONITOR; if it is interested only in outbound calls, it should specify LINECALLPRIVILEGE_NONE; if it wants to control unclassified calls (calls of unknown media mode), it should specify LINECALLPRIVILEGE_OWNER and LINEMEDIAMODE_UNKNOWN; if it knows the specific media mode with which it wants to deal, it should specify that media mode. Of course, you may specify more than one of these bits by using the OR operator.

Each service provider has a default media mode. When you specify other media modes in calling lineOpen(), those will be added to the one(s) already there, starting with the provider's default value. Your application may specify multiple flags simultaneously to handle multiple media modes. After the line has been opened, the lineMonitorMedia() function will modify the mask that controls LINE_MONITORMEDIA messages. But sometimes problems can occur. For example, if you open a line device with owner privilege and an extension media mode has not been registered, you will receive a LINEERR_INVALMEDIAMODE error. In addition, conflicts may arise if multiple applications open the same line device for the same media mode. These conflicts can be resolved by a priority scheme in which the user assigns relative priorities to applications. With this approach, only the application with the highest priority for a given media mode will ever receive ownership (unsolicited) of a call of that media mode.

There are two ways in which an application may receive ownership of a call: when an inbound call first arrives and when a call is handed off. How may my application receive such ownership, you ask? Any application—even a lower priority application—can acquire ownership by calling lineGetNewCalls() or lineGetConfRelatedCalls(). What if your application opens a line for monitoring when calls already exist on that line? In such a case, LINE_CALLSTATE messages for those existing calls will not automatically be passed to the new monitoring application. However, your application can query the number of current calls on the line and obtain handles to these calls by invoking the lineGetNewCalls() function.

If you want your application to handle automated voice calls, you should also select the interactive voice constant and receive the lowest priority for interactive voice. Here's why: Service providers will report all voice media modes as interactive voice. If your application does not perform media mode determination for the UNKNOWN media type and has not opened the line device for interactive voice, voice calls will not be able to reach the automated voice application. They will simply be dropped. For more information on this, see the TAPI Help file.

There are still more interesting possibilities. A single application, or different instantiations of an application, may open the same line multiple times with the same or different parameters. Keep in mind what we discussed earlier: When you open a line device, you must specify the negotiated TAPI version. If you want to use the line's extended capabilities, you should specify the line's device-specific extension version. You'll recall that these version numbers are obtained by calling the lineNegotiateAPIVersion() and lineNegotiateExtVersion() functions, respectively. This version numbering allows the mixing and matching of different application versions with different TAPI versions and service provider versions.

Earlier we discussed using the LINEMAPPER constant with lineOpen() to allow an application to select a line indirectly. When you do this, you must specify the specific services you want from that line. There are other issues involved. The TAPI Help file stresses that when you open a line device using LINEMAPPER, you must pay attention to all of the fields from the beginning of the LINECALLPARAMS data structure through the *dwAddressMode* field. If *dwAddressMode* has a value of LINEADDRESSMODE_ADDRESSID, TAPI will regard any address on the line as acceptable; otherwise, if *dwAddressMode* is LINEADDRESSMODE_DIALABLEADDR, TAPI will search for a specific originating address (phone number). In this latter case, if *dwAddressMode* is a provider-specific extension, *dwOrigAddressSize* and *dwOrigAddressOffset* will be relevant along with the portion of the variable part of the structure to which they point. If *dwAddressMode* is a provider-specific extension, additional information may be contained in the *dwDeviceSpecific* variably sized field.

Working with Media Modes

Additionally, different "lines" handle different media. Not surprisingly, TAPI needs to know about the media with which it will be working. It accomplishes this through *media modes*, the support for which is determined by the service provider. While TAPI can work with many media, it does have its limitations. For example, TAPI is not designed to provide support for fax transmissions. One solution is to use the functions available through MAPI, the Microsoft

Messaging API, to send and receive faxes. You could also use the older COM port functions. However, such a discussion is beyond the scope of this book.

How does TAPI determine the initial media mode(s)? When a service provider receives notification of a call's existence, it first determines the call's media mode to the best of its ability. This process varies with telephony systems. On a POTS line, TAPI will receive a ringing voltage, but with EPBX or ISDN, it will wait for a protocol message informing it that a call is incoming. In some cases, TAPI will be able to identify the single correct media mode. In others, it may have to settle for narrowing it down to a few possibilities. Not surprisingly, these first media mode settings are simply referred to as initial media modes. The TAPI Help file suggests the following as considerations used for setting initial media mode bits:

- **Service provider configuration**: The service provider's configuration is intended to work with a single media mode or specific media modes only.
- **Hardware limitations**: Limitations of the communications hardware are usually reflected in the service provider's configuration; however, a particular card being used could further restrict available media modes.
- **Call to lineOpen() function**: Media modes possible are limited by application requests during calls to the lineOpen() function. TAPI will combine all of the media modes requested by various applications and send the sum of them to the service provider when calling the service provider function TSPI_lineSetDefaultMediaDetection().
- **Caller ID/ Direct Inward Dialing**: With Direct Inward Dialing (DID) at the called address, the switch will supply the service provider with the digits that were dialed (the called address). It is possible to configure a service provider so that particular called addresses are associated with particular media modes.
- **Distinctive ringing**: The ring pattern of an incoming call can be compared with a predetermined pattern indicating a certain media mode.
- **ISDN**: On an ISDN network, the service provider may analyze an incoming call's protocol frames to determine the media mode. If the call is indicated as a 3.1 kHz voice call, it is still possible that the actual media mode on the call could be working with other forms of data.
- **Auto answer and probe**: Some providers give you the option to let the service provider answer the call automatically and conduct some of the probing itself. TAPI will give the call to the correct application with the correct media mode identified.

Unfortunately, these approaches may not be enough to determine the media mode definitively. When a service provider passes the new call to TAPI, it will send a LINE_CALLSTATE message, including in the message all that it knows about the call's media mode(s). We'll now discuss the details of the possible cases.

When the service provider knows the call's media unambiguously, one flag (for that particular media mode) will be set in LINECALLINFO's *dwMediaMode* field. In this case, the media mode cannot be the single bit LINEMEDIAMODE_UNKNOWN; that is a different scenario. TAPI gives ownership of the call to the highest priority application that has opened a line for this media mode. It also provides call handles with monitor privileges to all other monitoring applications on the line.

In addition to placing voice calls, your users may wish to send data over a phone line. To do this, the line must be available (not busy) and the connection must be established. After that, data can be sent. An application accomplishes this by giving control back to the user, who, using a dialog box, specifies the file to send and then initiates the data transmission. Though TAPI functions continue to manage the opened line and the call in progress, actual transmission is started and controlled by non-TAPI functions. In this case, for example, the Comm API could be used to control the media stream. Nevertheless, setting up a data call is similar to setting up a speech call. Once the call is established, the duty of data transmission is transferred outside of TAPI to the people who wish to speak, although the line and call continue to be monitored by the application using TAPI functions.

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

In this scenario, the service provider considers only the media modes it is capable of handling and for which applications have opened the line with owner privileges. It becomes aware of these media modes through the call to the function, TSPI_SetDefaultMediaDetection(). TAPI will inform the provider about the union of all the lines that have been opened with a specified media mode. However, there is a limitation: The service provider can use this union to enable only the specific media mode detections in which applications are interested. If no applications have opened the line for ownership, the provider will not consider any media mode(s). Incoming calls will still be delivered to TAPI, but no initial ownership will be possible. Nevertheless, applications with monitoring status will still be informed of the call, and if none of them change their privilege to owner and answer the call, the call will remain unanswered.

Closing a Line Device

As we mentioned above, when you initialize TAPI, you must shut it down when you're finished. Similarly, when you open a line device, you must shut it down when you're finished using it. This could hardly be easier. To close a line, you simply call the lineClose() function. After you've closed the line by calling this function, your application's handle for that line device will no longer be valid.

A LINE_LINEDEVSTATE message will be sent to other interested applications to inform them about the state change on the line. If an application calls lineClose() while it still has active calls on the opened line, the application's ownership of these calls will be revoked. If the application is the sole owner of these calls, the calls will be dropped as well. It is good programming practice for an application to dispose of the calls it owns on an opened line by explicitly relinquishing ownership and/or by dropping these calls prior to closing the line.

If the close is successful, a LINE_LINEDEVSTATE message will be sent to all applications that are monitoring the line status of open/close changes. Outstanding asynchronous replies will be suppressed. Service providers may find it useful or necessary to forcibly reclaim line devices from an application that has opened a line. This may be useful to prevent a misbehaving application from monopolizing the line device for too long. If this happens, a LINE_CLOSE message will be sent to the application, specifying the line handle of the line device that was closed.

After it is called, the lineOpen() function we discussed above will allocate resources to the invoking application. Consequently, other applications may be prevented from opening a line if resources are unavailable. Because of that possibility, an application that uses a line device (such as for making outbound calls) should only occasionally close the line to free resources and allow other applications to open the line.

TIP: Be resource aware. Close line devices not being used in order to free their resources.

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

While the lineClose() function closes a single line, the lineShutdown() function does something even more drastic—it disconnects an application from its connection to TAPI. Be aware that if you call this function when the application still has lines open or calls active, the call handles will be deleted; this is equivalent to calling the lineClose() function automatically on each open line, a rather brutal way to proceed. It is better practice for applications to explicitly close all open lines before calling the lineShutdown() function. If such a shutdown is performed while asynchronous requests are outstanding, those requests will be canceled. Additionally, an application that has registered as an Assisted Telephony request recipient should de-register itself by calling lineRegisterRequestRecipient(), using the value FALSE for the *bEnable* parameter.

The TAPI Help file points out that if you call this function while your application has active calls on the line, your application will lose ownership of those calls. If your application had been the sole owner of these calls, they will be dropped. You should always dispose of calls on an opened line by explicitly giving up ownership and/or by dropping them. If successful, TAPI will send a LINE_LINEDEVSTATE message to all monitoring applications indicating that the open/close line status has changed. Any outstanding asynchronous replies will be suppressed. Finally, as we have pointed out already, if your application uses a line device only occasionally, it should close that line at the first opportunity in order to free up its resources. Failure to do so could prevent other applications from opening the line.

In the above introduction, we briefly discussed some of the TAPI line messages. For a detailed description of each message and an example of handling messages in a callback function, see Chapter 9.

Reference for Basic TAPI Functions

In this section we will provide a reference for the basic TAPI functions, those that support initialization, configuration, capabilities checking, opening, and closing. We'll begin with the function we have just mentioned, lineClose(), and discuss the remaining functions in alphabetical order. We'll also discuss the structures and constants that are used with these functions. Each function reference includes Delphi code from our TAPI class. These functions are used in one of the sample applications available on the companion CD.

function lineClose TAPI.pas

Syntax

function lineClose(hLine: HLINE): Longint; stdcall;

Description

This function closes the specified open line device.

Parameters

hLine: A handle (HLINE) to the open line device to be closed. After the line has been successfully closed, this handle is no longer valid.

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONFAILED, and LINEERR_OPERATIONUNAVAIL.

See Also

LINE_CLOSE, LINE_LINELINEDEVSTATE, lineOpen

Example

Listing 8-1 shows how to close a line device that is open.

Listing 8-1: Closing a line device

```
function TTapiInterface.CloseLine: boolean;
begin
  result := True;
  if NOT fLineIsOpen then exit;
  if NOT LineClose(fLineApp)=0 then
    result := False;
end;
```

function lineConfigDialog TAPI.pas

Syntax

function lineConfigDialog(dwDeviceID: DWORD; hwndOwner: HWND; lpszDeviceClass: LPCSTR): Longint; stdcall;

Description

This function causes the service provider of the specified line device to display a dialog box that allows the user to configure parameters related to that line device.

Parameters

dwDeviceID: A DWORD holding the line device to be configured

hwndOwner: A handle (HWND) to a window to which the dialog is to be attached. It can be set to NIL to indicate that any window created during the function should have no owner window.

lpszDeviceClass: A pointer to a NULL-terminated string (LPCSTR) that identifies a device class name. This device class allows the application to select a specific secondary screen of configuration information applicable to that device class. This parameter is optional and can be set to NIL or empty, in which case the highest level configuration is selected.

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INUSE, LINEERR_OPERATIONFAILED, LINEERR_INVALDEVICECLASS, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPARAM, LINEERR_UNINITIALIZED, LINEERR_INVALPOINTER, LINEERR_OPERATIONUNAVAIL, and LINEERR_NODEVICE.

See Also

lineConfigDialogEdit, lineGetID

Example

Listing 8-2 shows how to call the lineConfigDialog() function to show the configuration dialog box.

Listing 8-2: Calling lineConfigDialog() to show the configuration dialog box

```
procedure TfrmConfigDialogDemo.btnShowConfigDialogClick(Sender: TObject);
begin
  If lineConfigDialog(DWORD(0), 0, Nil) <> 0 then
    ShowMessage('Could not display Line Configuration Dialog Box')
  else
    if NOT TapiInterface.GetLineConfiguration then
      ShowMessage('Could not retrieve Line Configuration Information');
end;
```

function lineConfigDialogEdit   TAPI.pas

Syntax

function lineConfigDialogEdit(dwDeviceID: DWORD; hwndOwner: HWND; lpszDeviceClass: LPCSTR; lpDeviceConfigIn: Pointer; dwSize: DWORD; lpDeviceConfigOut: PVarString): Longint; stdcall;

Description

This function causes the provider of the specified line device to display a dialog box (attached to *hwndOwner* of the application) that allows the user to configure parameters related to the line device.

Parameters

dwDeviceID: A DWORD holding the line device to be configured

hwndOwner: A handle (of type HWND) to a window to which the dialog box is to be attached. It can be set to NIL to indicate that any window created during the function should have no owner window.

lpszDeviceClass: A pointer to a NULL-terminated string (LPCSTR) that identifies a device class name. This device class allows the application to select a specific subscreen of configuration information applicable to that device class. This parameter is optional and can be left NULL or empty, in which case the highest level configuration is selected.

lpDeviceConfigIn: A pointer to the opaque configuration data structure that was returned by the lineGetDevConfig() function or from a previous call to this function (lineConfigDialogEdit()). The date is returned in the variable portion of the VarString structure.

dwSize: A DWORD indicating the number of bytes in the structure pointed to by *lpDeviceConfigIn*. This value will have been returned in the *dwStringSize* field in the Varstring structure returned by lineGetDevConfig() or a previous call to this function (lineConfigDialogEdit()).

lpDeviceConfigOut: A pointer to the memory location of type VarString (PVarString) in which the device configuration structure is returned. If the request is successfully completed, this location will be filled with the device configuration. The *dwStringFormat* field in the VarString structure will be set to STRINGFORMAT_BINARY. Before you call the lineGetDevConfig() function or initiate a future call to this function (lineConfigDialogEdit()), you should set the *dwTotalSize* field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_OPERATIONFAILED, LINEERR_INVALDEVICECLASS, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPARAM, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NODRIVER, LINEERR_OPERATIONUNAVAIL, LINEERR_NOMEM, and LINEERR_NODEVICE.

See Also

lineConfigDialog, lineGetDevConfig, lineGetID, lineSetDevConfig, VarString

Example

Listing 8-3 shows how to call the lineConfigDialogEdit() function to show the configuration dialog box.

Listing 8-3: Calling lineConfigDialogEdit() to show the configuration dialog box

```
procedure TTapiInterface.OpenlineConfigDialogEdit;
begin
  TAPIResult := 0; // Need to initialize to use in function
 if NOT fLineIsOpen then
    OpenLine(TAPIResult, False);
  if NOT GetLineID then exit;
    if NOT GetLineConfiguration then Exit;
  if FDeviceConfigOut=Nil then
    begin
      FDeviceConfigOut := AllocMem(SizeOf(VarString)+10000);
      FDeviceConfigOut.dwTotalSize := SizeOf(VarString)+10000;
      FDeviceConfigOut.dwStringFormat := STRINGFORMAT_BINARY;
    end;
  FConfigSize :=  FDeviceConfig.dwStringSize;
  TAPIResult := lineConfigDialogEdit(
    DWord(0), HWND(AppHandle),  PChar
    ('comm/datamodem'),
    @FDeviceConfig.data,
    FDeviceConfig.dwStringSize,
    FDeviceConfigOut);
    If TAPIResult<> 0 then ReportError(TAPIResult)
    else FDeviceConfig^ := pVarString(FDeviceConfigOut)^;
end;
```

function lineGetAddressCaps   TAPI.pas

Syntax

function lineGetAddressCaps(hLineApp: HLINEAPP; dwDeviceID, dwAddressID, dwAPIVersion, dwExtVersion: DWORD; lpAddressCaps: PLineAddressCaps): Longint; stdcall;

Description

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

Parameters

hLineApp: The handle (HLINEAPP) to the application's registration with TAPI

dwDeviceID: A DWORD holding the address on the given line device whose capabilities are to be queried

dwAddressID: A DWORD holding the line device containing the address to be queried *dwAPIVersion*: A DWORD holding 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: A DWORD holding the version number of the service provider-specific extensions to be used. This number can be left zero if no device-specific extensions are to be used. Otherwise, the high-order word contains the major version number, and the low-order word contains the minor version number.

lpAddressCaps: A pointer (PLineAddressCaps) to a variably sized structure of type LINEADDRESSCAPS. If the request is successfully completed, this structure is filled with address capabilities information. Before you call lineGetAddressCaps(), you should set the *dwTotalSize* field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_OPERATIONFAILED, LINEERR_INCOMPATIBLEEXTVERSION, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALADDRESSID, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALAPPHANDLE, LINEERR_UNINITIALIZED, LINEERR_INVALPOINTER, LINEERR_OPERATIONUNAVAIL, LINEERR_NODRIVER, and LINEERR_NODEVICE.

See Also

LINEADDRESSCAPS, lineGetDevcaps, lineNegotiateAPIVersion

Example

Listing 8-4 shows how to use this function to get an address's capabilities.

Listing 8-4: Getting an address's capabilities

```
function TTapiInterface.GetAddressCapsSize(var SizeReturned : DWord): boolean;
var
  TempAddrCaps : PLineAddressCaps;
begin
  TempAddrCaps := Nil;
  TempAddrCaps := AllocMem(SizeOf(LineAddressCaps));
  try
    TempAddrCaps^.dwTotalSize := SizeOf(LineAddressCaps);
    TAPIResult := LineGetAddressCaps(fLineApp, 0, 0, FAPIVersion,
      0,  TempAddrCaps);
    result := TAPIResult=0;
    if NOT result then ReportError(TAPIResult)
      else SizeReturned := TempAddrCaps^.dwNeededSize;
  finally // wrap up
```

```
    FreeMem(TempAddrCaps, SizeOf(LineAddressCaps));
    TempAddrCaps := Nil;
  end;    // try/finally
end;
```

structure LINEADDRESSCAPS TAPI.pas

The huge LINEADDRESSCAPS structure (TLineAddressCaps in TAPI.pas) describes the capabilities of a specified line address. The lineGetAddressCaps() function and the TSPI_lineGetAddressCaps() function return this structure. It is defined as follows in TAPI.pas:

```
type
  PLineAddressCaps = ^TLineAddressCaps;
  lineaddresscaps_tag = packed record
    dwTotalSize,
    dwNeededSize,
    dwUsedSize,
    dwLineDeviceID,
    dwAddressSize,
    dwAddressOffset,
    dwDevSpecificSize,
    dwDevSpecificOffset,
    dwAddressSharing,
    dwAddressStates,
    dwCallInfoStates,
    dwCallerIDFlags,
    dwCalledIDFlags,
    dwConnectedIDFlags,
    dwRedirectionIDFlags,
    dwRedirectingIDFlags,
    dwCallStates,
    dwDialToneModes,
    dwBusyModes,
    dwSpecialInfo,
    dwDisconnectModes,
    dwMaxNumActiveCalls,
    dwMaxNumOnHoldCalls,
    dwMaxNumOnHoldPendingCalls,
    dwMaxNumConference,
    dwMaxNumTransConf,
    dwAddrCapFlags,
    dwCallFeatures,
    dwRemoveFromConfCaps,
    dwRemoveFromConfState,
    dwTransferModes,
    dwParkModes,
    dwForwardModes,
    dwMaxForwardEntries,
    dwMaxSpecificEntries,
    dwMinFwdNumRings,
    dwMaxFwdNumRings,
    dwMaxCallCompletions,
    dwCallCompletionConds,
    dwCallCompletionModes,
    dwNumCompletionMessages,
    dwCompletionMsgTextEntrySize,
    dwCompletionMsgTextSize,
    dwCompletionMsgTextOffset,
```

```
    dwAddressFeatures: DWORD;                       // TAPI v1.4
{$IFDEF TAPI20}
    dwPredictiveAutoTransferStates,                 // TAPI v2.0
    dwNumCallTreatments,                            // TAPI v2.0
    dwCallTreatmentListSize,                        // TAPI v2.0
    dwCallTreatmentListOffset,                      // TAPI v2.0
    dwDeviceClassesSize,                            // TAPI v2.0
    dwDeviceClassesOffset,                          // TAPI v2.0
    dwMaxCallDataSize,                              // TAPI v2.0
    dwCallFeatures2,                                // TAPI v2.0
    dwMaxNoAnswerTimeout,                           // TAPI v2.0
    dwConnectedModes,                               // TAPI v2.0
    dwOfferingModes,                                // TAPI v2.0
    dwAvailableMediaModes: DWORD;                   // TAPI v2.0
{$ENDIF}
  end;
  TLineAddressCaps = lineaddresscaps_tag;
  LINEADDRESSCAPS = lineaddresscaps_tag;
```


Each of the parameters of LINEADDRESSCAPS is described in Table 8-1.

Table 8-1: Parameters of the LINEADDRESSCAPS structure

Parameter	Meaning
dwTotalSize	This field specifies the total size in bytes allocated to this data structure.
dwNeededSize	This field specifies the size in bytes for this data structure that is needed to hold all the returned information.
dwUsedSize	This field specifies the size in bytes of the portion of this data structure that contains useful information.
dwLineDeviceID	This field specifies the device ID of the line device with which this address is associated.
dwAddressSize	This field specifies the size in bytes of the variably sized address field and the offset in bytes from the beginning of this data structure.
dwAddressOffset	This field specifies the size in bytes of the variably sized address field and the offset in bytes from the beginning of this data structure.
dwDevSpecificSize	This field specifies the size in bytes of the variably sized device-specific field and the offset in bytes from the beginning of this data structure.
dwDevSpecificOffset	This field specifies the size in bytes of the variably sized device-specific field and the offset in bytes from the beginning of this data structure.
dwAddressSharing	This field specifies the sharing mode of the address. Values include the following constants: LINEADDRESSSHARING_PRIVATE indicates that an address with private sharing mode is only assigned to a single line or station. LINEADDRESSSHARING_BRIDGEDEXCL indicates that an address with a bridged-exclusive sharing mode is assigned to one or more other lines or stations (the exclusive portion refers to the fact that only one of the bridged parties can be connected with a remote party at any given time). LINEADDRESSSHARING_BRIDGEDNEW indicates that an address with a bridged-new sharing mode is assigned to one or more other lines or stations (the new portion refers to the fact that activities by the different bridged parties result in the creation of new calls on the address).

Parameter	Meaning
dwAddressSharing	LINEADDRESSSHARING_BRIDGEDSHARED indicates that an address with a bridged-shared sharing mode is also assigned to one or more other lines or stations (the shared portion refers to the fact that if one of the bridged parties is connected with a remote party, the remaining bridged parties can share in the conversation, as in a conference, by activating that call appearance). LINEADDRESSSHARING_MONITORED indicates that an address with a monitored address mode simply monitors the status of that address (the status is either idle or in use; the message LINE_ADDRESSSTATE notifies the application about these changes).
dwAddressStates	This field contains the address states changes for which the application may get notified in the LINE_ADDRESSSTATE message. It uses one of the LINEADDRESSSTATE_ constants described in Table 8-2.
dwCallInfoStates	This field specifies the call information elements that are meaningful for all calls on this address. An application may get notified about changes in some of these states in LINE_CALLINFO messages. It uses the LINECALLINFOSTATE_ constants described in Table 8-3.
dwCallerIDFlags	This field specifies an item of party ID information that may be provided for calls on this address. It uses the LINECALLPARTYID_ constants shown in Table 8-4.
dwCalledIDFlags	This field specifies an item of party ID information that may be provided for calls on this address. It uses the LINECALLPARTYID_ constants shown in Table 8-4.
dwConnectedIDFlags	This field specifies an item of party ID information that may be provided for calls on this address. It uses the LINECALLPARTYID_ constants shown in Table 8-4.
dwRedirectionIDFlags	This field specifies an item of party ID information that may be provided for calls on this address. It uses the LINECALLPARTYID_ constants shown in Table 8-4.
dwRedirectingIDFlags	This field specifies an item of party ID information that may be provided for calls on this address. It uses the LINECALLPARTYID_ constants shown in Table 8-4.
dwCallStates	This field specifies the various call states that can possibly be reported for calls on this address. This parameter uses the LINECALLSTATE_ constants shown in Table 8-5.
dwDialToneModes	This field specifies the various dial tone modes that can possibly be reported for calls made on this address. This field is meaningful only if the dial tone call state can be reported. It uses the following LINEDIALTONEMODE_ constants: LINEDIALTONEMODE_NORMAL indicates that this is a "normal" dial tone, which typically is a continuous tone. LINEDIALTONEMODE_SPECIAL indicates that this is a special dial tone indicating a certain condition is currently in effect. LINEDIALTONEMODE_INTERNAL indicates that this is an internal dial tone, as within a PBX. LINEDIALTONEMODE_EXTERNAL indicates that this is an external (public network) dial tone. LINEDIALTONEMODE_UNKNOWN indicates that the dial tone mode is currently unknown but may become known later. LINEDIALTONEMODE_UNAVAIL indicates that the dial tone mode is unavailable and will not become known.

Parameter	Meaning
wBusyModes	This field specifies the various busy modes that can possibly be reported for calls made on this address. This field is meaningful only if the busy call state can be reported. It uses the following LINEBUSYMODE_ constants: LINEBUSYMODE_STATION indicates that the busy signal means that the called party's station is busy (this is usually signaled with a "normal" busy tone). LINEBUSYMODE_TRUNK indicates that the busy signal means that a trunk or circuit is busy (this is usually signaled with a "long" busy tone). LINEBUSYMODE_UNKNOWN indicates that the busy signal's specific mode is currently unknown but may become known later. LINEBUSYMODE_UNAVAIL indicates that the busy signal's specific mode is unavailable and will not become known.
dwSpecialInfo	This field specifies the various special information types that can possibly be reported for calls made on this address. This field is meaningful only if the specialInfo call state can be reported. It uses the following LINESPECIALINFO_ constants: LINESPECIALINFO_NOCIRCUIT indicates that this special information tone precedes a no-circuit or emergency announcement (trunk blockage category). LINESPECIALINFO_CUSTIRREG indicates that this special information tone precedes one of the following: a vacant number, an Alarm Indication Signal (AIS), a Centrex number change with a nonworking station, an access code that was not dialed or dialed in error, or a manual intercept operator message (customer irregularity category). LINESPECIALINFO_REORDER indicates that this special information tone precedes a reorder announcement (equipment irregularity category). LINESPECIALINFO_UNKNOWN indicates that specifics about the special information tone are currently unknown but may become known later. LINESPECIALINFO_UNAVAIL indicates that specifics about the special information tone are unavailable and will not become known.
dwDisconnectModes	This field specifies the various disconnect modes that can possibly be reported for calls made on this address. This field is meaningful only if the disconnected call state can be reported. It uses the following LINEDISCONNECTMODE_ constants: LINEDISCONNECTMODE_NORMAL indicates that this is a "normal" disconnect request by the remote party; the call was terminated normally. LINEDISCONNECTMODE_UNKNOWN indicates that the reason for the disconnect request is unknown. LINEDISCONNECTMODE_REJECT indicates that the remote user has rejected the call. LINEDISCONNECTMODE_PICKUP indicates that the call was picked up from elsewhere. LINEDISCONNECTMODE_FORWARDED indicates that the call was forwarded by the switch. LINEDISCONNECTMODE_BUSY indicates that the remote user's station is busy. LINEDISCONNECTMODE_NOANSWER indicates that the remote user's station does not answer. LINEDISCONNECTMODE_NODIALTONE indicates that a dial tone was not detected within a service-provider defined timeout at a point during dialing when one was expected (such as at a "W" in the dialable string), a situation that can also occur without a service provider-defined timeout period or without a value specified in the dwWaitForDialTone member of the LINEDIALPARAMS structure.

Parameter	Meaning
dwDisconnectModes (cont.)	LINEDISCONNECTMODE_BADADDRESS indicates that the destination address in invalid. LINEDISCONNECTMODE_UNREACHABLE indicates that the remote user could not be reached. LINEDISCONNECTMODE_CONGESTION indicates that the network is congested. LINEDISCONNECTMODE_INCOMPATIBLE indicates that the remote user's station equipment is incompatible with the type of call requested. LINEDISCONNECTMODE_UNAVAIL indicates that the reason for the disconnect is unavailable and will not become known later.
dwMaxNumActiveCalls	This field specifies the maximum number of active call appearances that the address can handle. This number does not include calls on hold or calls on hold pending transfer or conference.
dwMaxNumOnHoldCalls	This field specifies the maximum number of call appearances at the address that can be on hold.
dwMaxNumOnHoldPendingCalls	This field specifies the maximum number of call appearances at the address that can be on hold pending transfer or conference.
dwMaxNumConference	This field specifies the maximum number of parties that can be conferenced in a single conference call on this address.
dwMaxNumTransConf	This field specifies the number of parties (including "self") that can be added in a conference call that is initiated as a generic consultation call using lineSetupTransfer().
dwAddrCapFlags	This field specifies a series of packed bit flags that describe a variety of address capabilities. It uses the LINEADDRCAPFLAGS_ constants shown in Table 8-6.
dwCallFeatures	This field specifies the switching capabilities or features available for all calls on this address using the LINECALLFEATURE_ constants explained in Table 8-7. This member represents the call-related features which may possibly be available on an address (static availability as opposed to dynamic availability). Invoking a supported feature requires the call to be in the proper state and the underlying line device to be opened in a compatible mode. A zero in a bit position indicates that the corresponding feature is never available. A one indicates that the corresponding feature may be available if the application has the right privileges to the call and the call is in the appropriate state for the operation to be meaningful. This field allows an application to discover which call features can be (and which can never be) supported by the address.
dwRemoveFromConfCaps	This field specifies the address's capabilities for removing calls from a conference call. It uses the following LINEREMOVEFROMCONF_ constants: LINEREMOVEFROMCONF_NONE indicates that parties cannot be removed from the conference call. LINEREMOVEFROMCONF_LAST indicates that only the most recently added party can be removed from the conference call. LINEREMOVEFROMCONF_ANY indicates that any participating party can be removed from the conference call.
dwRemoveFromConfState	Using the LINECALLSTATE_ constants, this field specifies the state of the call after it has been removed from a conference call. (See Table 8-5.)
dwTransferModes	This field specifies the address's capabilities for resolving transfer requests. It uses the following LINETRANSFERMODE_ constants: LINETRANSFERMODE_TRANSFER indicates to resolve the initiated transfer by transferring the initial call to the consultation call.

Parameter	Meaning
dwTransferModes (cont.)	LINETRANSFERMODE_CONFERENCE indicates to resolve the initiated transfer by conferencing all three parties into a three-way conference call (in this case a conference call is created and returned to the application).
dwParkModes	This field specifies the different call park modes available at this address using the LINEPARKMODE_ constants: LINEPARKMODE_DIRECTED specifies directed call park in which the address where the call is to be parked must be supplied to the switch. LINEPARKMODE_NONDIRECTED specifies non-directed call park in which the address where the call is parked is selected by the switch and provided by the switch to the application.
dwForwardModes	This field specifies the different modes of forwarding available for this address. It uses the LINEFORWARDMODE_ constants described in Table 8-8.
dwMaxForwardEntries	This field specifies the maximum number of entries that can be passed to lineForward in the lpForwardList parameter.
dwMaxSpecificEntries	This field specifies the maximum number of entries in the lpForwardList parameter passed to lineForward() that can contain forwarding instructions based on a specific caller ID (selective call forwarding). This field is zero if selective call forwarding is not supported.
dwMinFwdNumRings	This field specifies the minimum number of rings that can be set to determine when a call is officially considered "no answer."
dwMaxFwdNumRings	This field specifies the maximum number of rings that can be set to determine when a call is officially considered "no answer." If this number of rings cannot be set, then dwMinFwdNumRings and dwMaxFwdNumRings will be equal.
dwMaxCallCompletions	This field specifies the maximum number of concurrent call completion requests that can be outstanding on this line device. Zero implies that call completion is not available.
dwCallCompletionCond	This field specifies the different call conditions under which call completion can be requested using the following LINECALLCOMPLCOND_ constants: LINECALLCOMPLCOND_BUSY indicates to complete the call under the busy condition. LINECALLCOMPLCOND_NOANSWER indicates to complete the call under the ringback no answer condition.
dwCallCompletionModes	This field specifies the way in which the call can be completed using the following LINECALLCOMPLCOND_ constants: LINECALLCOMPLMODE_CAMPON indicates to queue the call until the call can be completed. LINECALLCOMPLMODE_CALLBACK requests the called station to return the call when it returns to idle. LINECALLCOMPLMODE_INTRUDE adds the application to the existing call at the called station if busy (barge in). LINECALLCOMPLMODE_MESSAGE leaves a short predefined message for the called station (Leave Word Calling); a specific message can be identified.
dwNumCompletionMessages	This field specifies the number of call completion messages that can be selected from using the LINECALLCOMPLMODE_MESSAGE option. Individual messages are identified by values in the range zero through one less than dwNumCompletionMessages.
dwCompletionMsgTextEntrySize	This field specifies the size in bytes of each of the call completion text descriptions pointed to by dwCompletionMsgTextSize/Offset.

Parameter	Meaning
dwCompletionMsgTextSize	This field specifies the size in bytes of the data structure of the variably sized field containing descriptive text about each of the call completion messages. Each message is dwCompletionMsgTextEntrySize bytes long. The string format of these textual descriptions is indicated by dwStringFormat in the line's device capabilities.
dwCompletionMsgTextOffset	This field specifies the offset in bytes from the beginning of the data structure of the variably sized field containing descriptive text about each of the call completion messages. Each message is dwCompletionMsgTextEntrySize bytes long. The string format of these textual descriptions is indicated by dwStringFormat in the line's device capabilities.
dwAddressFeatures	This field specifies the features available for this address using the LINEADDRFEATURE_ constants. Invoking a supported feature requires the address to be in the proper state and the underlying line device to be opened in a compatible mode. A zero in a bit position indicates that the corresponding feature is never available. A one indicates that the corresponding feature may be available if the address is in the appropriate state for the operation to be meaningful. This field allows an application to discover which address features can be (and which can never be) supported by the address.
dwPredictiveAutoTransferStates	This field specifies the call state or states upon which a call made by a predictive dialer can be set to automatically transfer the call to another address—one or more of the LINECALLSTATE_ constants. A value of zero indicates that automatic transfer-based on call state is unavailable. (See Table 8-5.)
dwNumCallTreatments	This field specifies the number of entries of LINECALLTREATMENTENTRY structures. These entries are delimited by the dwCallTreatmentSize and dwCallTreatmentOffset fields of the LINECALLTREATMENT structure.
dwCallTreatmentListSize	This field specifies the total size in bytes of LINEADDRESSCAPS of an array of LINECALLTREATMENTENTRY structures, indicating the call treatments supported on the address (which can be selected using lineSetCallTreatment()). The value will be dwNumCallTreatments times SIZEOF (LINECALLTREATMENTENTRY).
dwCallTreatmentListOffset	This field specifies the offset from the beginning of LINEADDRESSCAPS of an array of LINECALLTREATMENTENTRY structures, indicating the call treatments supported on the address (which can be selected using lineSetCallTreatment()). The value will be dwNumCallTreatments times SIZEOF (LINECALLTREATMENTENTRY).
dwDeviceClassesSize	This field specifies the length in bytes of LINEADDRESSCAPS of a string consisting of the device class identifiers supported on this address for use with lineGetID(), separated by NULLS; the last class indentifier is followed by two NULLS.
dwDeviceClassesOffset	This field specifies the offset from the beginning of LINEADDRESSCAPS of a string consisting of the device class identifiers supported on this address for use with lineGetID(), separated by NULLS; the last class identifier is followed by two NULLS.
dwMaxCallDataSize	This field specifies the maximum number of bytes that an application can set in LINECALLINFO using lineSetCallData().
dwCallFeatures2	This field specifies additional switching capabilities or features available for all calls on this address using the LINECALLFEATURE2_ constants. It is an extension of the dwCallFeatures member.

Parameter	Meaning
dwMaxNoAnswerTimeout	This field specifies the maximum value in seconds that can be set in the dwNoAnswerTimeout member in LINECALLPARAMS when making a call. A value of zero indicates that automatic abandonment of unanswered calls is not supported by the service provider or that the timeout value is not adjustable by applications.
dwConnectedModes	This field specifies the LINECONNECTEDMODE_ values that may appear in the dwCallStateMode member of LINECALLSTATUS and in LINE_CALLSTATE messages for calls on this address.
dwOfferingModes	This field specifies the LINEOFFERINGMODE_ values that may appear in the dwCallStateMode member of LINECALLSTATUS and in LINE_CALLSTATE messages for calls on this address.
dwAvailableMediaModes	This field specifies the media modes that can be invoked on new calls created on this address, when the dwAddressFeatures member indicates that new calls are possible. If this field is zero, it indicates that the service provider either does not know or cannot indicate which media modes are available, in which case any or all of the media modes indicated in the dwMediaModes field in LINEDEVCAPS may be available.

Table 8-2: LINEADDRESSSTATE_ constants used in the LINEADDRESSCAPS dwAddressStates parameter

Constant	Meaning
LINEADDRESSSTATE_OTHER	This constant indicates that address status items, other than those listed below, have changed. The application should check the current address status to determine which items have changed.
LINEADDRESSSTATE_ DEVSPECIFIC	This constant indicates that the device-specific item of the address status has changed.
LINEADDRESSSTATE_ INUSEZERO	This constant indicates that the address has changed to idle (it is not in use by any stations).
LINEADDRESSSTATE_ INUSEONE	This constant indicates that the address has changed from being idle or from being in use by many bridged stations to being in use by just one station.
LINEADDRESSSTATE_ INUSEMANY	This constant indicates that the monitored or bridged address has changed to being in use by one station to being used by more than one station.
LINEADDRESSSTATE_ NUMCALLS	This constant indicates that the number of calls on the address has changed. This is the result of events such as a new inbound call, an outbound call on the address, or a call changing its hold status.
LINEADDRESSSTATE_ FORWARD	This constant indicates that the forwarding status of the address has changed, including the number of rings for determining a "no answer" condition. The application should check the address status to retrieve details about the address's current forwarding status.
LINEADDRESSSTATE_ TERMINALS	This constant indicates that the terminal settings for the address have changed.
LINEADDRESSSTATE_ CAPSCHANGE	This constant indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the LINE_ADDRESSCAPS structure for the address have changed. The application should use lineGet-AddressCaps() to read the updated structure.

Constant	Meaning
LINEADDRESSSTATE_ CAPSCHANGE (cont.)	If a service provider sends a LINE_ADDRESSSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 1.4 or above; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINE_DEVSTATE_REINIT, requiring them to shut down and reinitialize their connection to TAPI in order to obtain the updated information.

Table 8-3: LINECALLINFOSTATE_ constants used in the LINEADDRESSCAPS dwCallInfoStates parameter

Parameter	Meaning
LINECALLINFOSTATE_OTHER	This constant indicates that call information items, other than those listed below, have changed. The application should check the current call information to determine which items have changed.
LINECALLINFOSTATE_ DEVSPECIFIC	This constant indicates the device-specific field of the call information.
LINECALLINFOSTATE_ BEARERMODE	This constant indicates the bearer mode field of the call information record.
LINECALLINFOSTATE_RATE	This constant indicates the rate field of the call information record.
LINECALLINFOSTATE_ MEDIAMODE	This constant indicates the media mode field of the call information record.
LINECALLINFOSTATE_ APPSPECIFIC	This constant indicates the application-specific field of the call information record.
LINECALLINFOSTATE_CALLID	This constant indicates the caller ID field of the call information record.
LINECALLINFOSTATE_ RELATEDCALLID	This constant indicates the related caller ID field of the call information record.
LINECALLINFOSTATE_ORIGIN	This constant indicates the origin field of the call information record.
LINECALLINFOSTATE_ REASON	This constant indicates the reason field of the call information record.
LINECALLINFOSTATE_ COMPLETIONID	This constant indicates the completion ID field of the call information record.
LINECALLINFOSTATE_ NUMOWNERINCR	This constant indicates that the number of the owner field in the call information record was increased.
LINECALLINFOSTATE_ NUMOWNERDECR	This constant indicates that the number of the owner field in the call information record was decreased.
LINECALLINFOSTATE_ NUMMONITORS	This constant indicates that the number of the monitors field in the call information record has changed.
LINECALLINFOSTATE_TRUNK	This constant indicates that the trunk field of the call information record has changed.
LINECALLINFOSTATE_ CALLERID	This constant indicates that one of the caller ID-related fields of the call information record has changed.
LINECALLINFOSTATE_ CALLEDID	This constant indicates that one of the called ID-related fields of the call information record has changed.
LINECALLINFOSTATE_ CONNECTEDID	This constant indicates that one of the connected ID-related fields of the call information record has changed.
LINECALLINFOSTATE_ REDIRECTIONID	This constant indicates that one of the redirection ID-related fields of the call information record has changed.

Parameter	Meaning
LINECALLINFOSTATE_ REDIRECTINGID	This constant indicates that one of the redirecting ID-related fields of the call information record has changed.
LINECALLINFOSTATE_ DISPLAY	This constant indicates the display field of the call information record.
LINECALLINFOSTATE_ USERUSERINFO	This constant indicates the user-to-user information of the call information record.
LINECALLINFOSTATE_ HIGHLEVELCOMP	This constant indicates the high-level compatibility field of the call information record.
LINECALLINFOSTATE_ LOWLEVELCOMP	This constant indicates the low-level compatibility field of the call information record.
LINECALLINFOSTATE_ CHARGINGINFO	This constant indicates the charging information of the call information record.
LINECALLINFOSTATE_ TERMINAL	This constant indicates the terminal mode information of the call information record.
LINECALLINFOSTATE_ DIALPARAMS	This constant indicates the dial parameters of the call information record.
LINECALLINFOSTATE_ MONITORMODES	This constant indicates that one or more of the digit, tone, or media monitoring fields in the call information record has changed.

Table 8-4: LINECALLPARTYID_ constants used with various LINEADDRESSCAPS ID flags

Constant	Meaning
LINECALLPARTYID_BLOCKED	This constant indicates that the caller ID information for the call has been blocked by the caller but would otherwise have been available.
LINECALLPARTYID_ OUTOFAREA	This constant indicates that the caller ID information for the call is not available because it is not propagated all the way by the network.
LINECALLPARTYID_NAME	This constant indicates that the caller ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the caller ID name variably sized field.
LINECALLPARTYID_ADDRESS	This constant indicates that the caller ID information for the call is the caller's number and is provided in the caller ID variably sized field.
LINECALLPARTYID_PARTIAL	This constant indicates that the caller ID information for the call is valid but is limited to partial number information.
LINECALLPARTYID_ UNKNOWN	This constant indicates that the caller ID information is currently unknown; it may become known later.
LINECALLPARTYID_UNAVAIL	This constant indicates that the caller ID information is unavailable and will not become known later.

Table 8-5: LINECALLSTATE_ constants used with the LINEADDRESSCAPS dwCallStates parameter

Constant	Meaning
LINECALLSTATE_IDLE	This constant indicates that the call is idle; no call exists.
LINECALLSTATE_OFFERING	This constant indicates that the call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the offering state does not automatically alert the user; alerting is done by the switch instructing the line to ring. It does not affect any call states.
LINECALLSTATE_ACCEPTED	This constant indicates that the call was offered and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.
LINECALLSTATE_DIALTONE	This constant indicates that the call is receiving a dial tone from the switch, which means that the switch is ready to receive a dialed number.
LINECALLSTATE_DIALING	This constant indicates that the destination address information (a phone number) is being sent to switch over the call. Note that the operation, lineGenerateDigits(), does not place the line into the dialing state.
LINECALLSTATE_RINGBACK	This constant indicates that the call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.
LINECALLSTATE_BUSY	This constant indicates that the call is receiving a busy tone. A busy tone indicates that the call cannot be completed—either a circuit (trunk) or the remote party's station are in use.
LINECALLSTATE_ SPECIALINFO	This constant indicates that special information is being sent by the network. Special information is typically sent when the destination cannot be reached.
LINECALLSTATE_ CONNECTED	This constant indicates that the call has been established and the connection is made. Information is able to flow over the call between the originating address and the destination address.
LINECALLSTATE_ PROCEEDING	This constant indicates that the dialing process has completed and the call is proceeding through the switch or telephone network.
LINECALLSTATE_ONHOLD	This constant indicates that the call is on hold by the switch.
LINECALLSTATE_ CONFERENCED	This constant indicates that the call is currently a member of a multiparty conference call.
LINECALLSTATE_ ONHOLDPENDCONF	This constant indicates that the call is currently on hold while it is being added to a conference.
LINECALLSTATE_ ONHOLDPENDTRANSF	This constant indicates that the call is currently on hold awaiting transfer to another number.
LINECALLSTATE_ DISCONNECTED	This constant indicates that the remote party has disconnected from the call.
LINECALLSTATE_UNKNOWN	This constant indicates that the state of the call is not known. This situation may be due to limitations of the call progress detection implementation.

Table 8-6: LINEADDRCAPFLAGS_ constants used with the LINEADDRESSCAPS dwAddrCapFlags parameter

Constant	Meaning
LINEADDRCAPFLAGS_ FWDNUMRINGS	This constant indicates whether the number of rings for a "no answer" can be specified when forwarding calls on no answer.
LINEADDRCAPFLAGS_ PICKUPGROUPID	This constant indicates whether or not a group ID is required for call pickup.
LINEADDRCAPFLAGS_SECURE	This constant indicates whether or not calls on this address can be made secure at call setup time.
LINEADDRCAPFLAGS_ BLOCKIDDEFAULT	This constant indicates whether the network by default sends or blocks caller ID information when making a call on this address. If TRUE (set), ID information is blocked by default; if FALSE, ID information is transmitted by default.
LINEADDRCAPFLAGS_ BLOCKIDOVERRIDE	This constant indicates whether the default setting for sending or blocking of caller ID information can be overridden per call. If TRUE, override is possible; if FALSE, override is not possible.
LINEADDRCAPFLAGS_DIALED	This constant indicates whether a destination address can be dialed on this address for making a call. TRUE if a destination address must be dialed; FALSE if the destination address is fixed (as with a "hot phone").
LINEADDRCAPFLAGS_ ORIGOFFHOOK	This constant indicates whether the originating party's phone can automatically be taken offhook when making calls.
LINEADDRCAPFLAGS_ DESTOFFHOOK	This constant indicates whether the called party's phone can automatically be forced offhook when making calls.
LINEADDRCAPFLAGS_ FWDCONSULT	This constant indicates whether call forwarding involves the establishment of a consultation call.
LINEADDRCAPFLAGS_ SETUPCONFNULL	This constant indicates whether setting up a conference call starts out with an initial call (FALSE) or with no initial call (TRUE).
LINEADDRCAPFLAGS_ AUTORECONNECT	This constant indicates whether dropping a consultation call automatically reconnects to the call on consultation hold. TRUE if reconnect happens automatically; otherwise, FALSE.
LINEADDRCAPFLAGS_ COMPLETIONID	This constant indicates whether the completion IDs returned by lineCompleteCall() are useful and unique. TRUE if useful; otherwise, FALSE.
LINEADDRCAPFLAGS_ TRANSFERHELD	This constant indicates whether a (hard) held call can be transferred. Often, only calls on consultation hold may be able to be transferred.
LINEADDRCAPFLAGS_ CONFERENCEHELD	This constant indicates whether a (hard) held call can be added to a conference call. Often, only calls on consultation hold may be able to be added to as a conference call.
LINEADDRCAPFLAGS_ PARTIALDIAL	This constant indicates whether partial dialing is available.
LINEADDRCAPFLAGS_ FWDSTATUSVALID	This constant indicates whether the forwarding status in the LINEADDRESSSTATUS structure for this address is valid.
LINEADDRCAPFLAGS_ FWDINTEXTADDR	This constant indicates whether internal and external calls can be forwarded to different forwarding addresses. This flag is meaningful only if forwarding of internal and external calls can be controlled separately. It is TRUE if internal and external calls can be forwarded to different destination addresses; otherwise, FALSE.

Constant	Meaning
LINEADDRCAPFLAGS_ FWDBUSYNAADDR	This constant indicates whether call forwarding for busy and for no answer can use different forwarding addresses. This flag is meaningful only if forwarding for busy and for no answer can be controlled separately. It is TRUE if forwarding for busy and for no answer can use different destination addresses; otherwise, FALSE.
LINEADDRCAPFLAGS_ ACCEPTTOALERT	This constant will be TRUE if an offering call must be accepted using the lineAccept() function, alerting the users at both ends of the call; otherwise, it will be FALSE. Typically, this is only used with ISDN.
LINEADDRCAPFLAGS_ CONFDROP	This constant will be TRUE if invoking lineDrop() on a conference call parent also has the side effect of dropping (disconnecting) the other parties involved in the conference call; FALSE if dropping a conference call still allows the other parties to talk among themselves.

Table 8-7: LINECALLFEATURE_ constants used with the LINEADDRESSCAPS dwCallFeatures parameter

Constant	Associated Function/TAPI Version
LINECALLFEATURE_ACCEPT	lineAccept()/All
LINECALLFEATURE_ ADDTOCONF	lineAddToConference()/All
LINECALLFEATURE_ANSWER	lineAnswer()/All
LINECALLFEATURE_ BLINDTRANSFER	lineBlindTransfer()/All
LINECALLFEATURE_ COMPLETECALL	lineCompleteCall()/All
LINECALLFEATURE_ COMPLETETRANSF	lineCompleteTransfer()/All
LINECALLFEATURE_DIAL	lineDial()/All
LINECALLFEATURE_DROP	lineDrop()/All
LINECALLFEATURE_ GATHERDIGITS	lineGatherDigits()/All
LINECALLFEATURE_ GENERATEDIGITS	lineGenerateDigits()/All
LINECALLFEATURE_ GENERATETONE	lineGenerateTone()/All
LINECALLFEATURE_HOLD	lineHold()/All
LINECALLFEATURE_ MONITORDIGITS	lineMonitorDigits()/All
LINECALLFEATURE_ MONITORMEDIA	lineMonitorMedia()/All
LINECALLFEATURE_ MONITORTONES	lineMonitorTones( )/All
LINECALLFEATURE_PARK	linePark()/All
LINECALLFEATURE_ PREPAREADDCONF	linePrepareAddToConference()/All
LINECALLFEATURE_ REDIRECT	lineRedirect()/All

Constant	Associated Function/TAPI Version
LINECALLFEATURE_ REMOVEFROMCONF	lineRemoveFromConference()/All
LINECALLFEATURE_ SECURECALL	lineSecureCall()/All
LINECALLFEATURE_ SENDUSERUSER	lineSendUserUserInfo()/All
LINECALLFEATURE_ SETCALLPARAMS	lineSetCallParams()/All
LINECALLFEATURE_ SETMEDIACONTROL	lineSetMediaControl()/All
LINECALLFEATURE_ SETTERMINAL	lineSetTerminal()/All
LINECALLFEATURE_ SETUPCONF	lineSetupConference()/All
LINECALLFEATURE_ SETUPTRANSFER	lineSetupTransfer()/All
LINECALLFEATURE_ SWAPHOLD	lineSwapHold()/All
LINECALLFEATURE_UNHOLD	lineUnhold()/All
LINECALLFEATURE_ RELEASEUSERUSERINFO	lineReleaseUserUserInfo()/TAPI 1.4
LINECALLFEATURE_ SETTREATMENT	lineSetTreatment()/TAPI 2.0
LINECALLFEATURE_SETQOS	lineSetCallQualityOfService()/TAPI 2.0

Table 8-8: LINEFORWARDMODE_ constants used with the LINEADDRESSCAPS dwForwardModes parameter

Constant	Meaning
LINEFORWARDMODE_ UNCOND	This constant indicates to forward all calls unconditionally, irrespective of their origin. You should use this value when unconditional forwarding for internal and external calls cannot be controlled separately. Unconditional forwarding overrides forwarding on busy and/or no answer conditions.
LINEFORWARDMODE_ UNCONDINTERNAL	This constant indicates to forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.
LINEFORWARDMODE_ UNCONDEXTERNAL	This constant indicates to forward all external calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.
LINEFORWARDMODE_ UNCONDSPECIFIC	This constant indicates to forward all calls that originated at a specified address unconditionally (selective call forwarding).
LINEFORWARDMODE_BUSY	This constant indicates to forward all calls on busy, irrespective of their origin. Use this value when forwarding for internal and external calls on busy and when on no answer cannot be controlled separately.
LINEFORWARDMODE_ BUSYINTERNAL	This constant indicates to forward all internal calls on busy. Use this value when forwarding for internal and external calls on busy and when on no answer can be controlled separately.

Constant	Meaning
LINEFORWARDMODE_ BUSYEXTERNAL	This constant indicates to forward all external calls on busy. Use this value when forwarding for internal and external calls on busy and when on no answer can be controlled separately.
LINEFORWARDMODE_ BUSYSPECIFIC	This constant indicates to forward all calls that originated at a specified address on busy (selective call forwarding).
LINEFORWARDMODE_ NOANSW	This constant indicates to forward all calls on no answer, irrespective of their origin. Use this value when call forwarding for internal and external calls on no answer cannot be controlled separately.
LINEFORWARDMODE_ NOANSWINTERNAL	This constant indicates to forward all internal calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately.
LINEFORWARDMODE_ NOANSWEXTERNAL	This constant indicates to forward all external calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately.
LINEFORWARDMODE_ NOANSWSPECIFIC	This constant indicates to forward all calls that originated at a specified address on no answer (selective call forwarding).
LINEFORWARDMODE_ BUSYNA	This constant indicates to forward all calls on busy/no answer, irrespective of their origin. Use this value when forwarding for internal and external calls on busy and on no answer cannot be controlled separately.
LINEFORWARDMODE_ BUSYNAINTERNAL	This constant indicates to forward all internal calls on busy/no answer. Use this value when call forwarding on busy and on no answer cannot be controlled separately for internal calls.
LINEFORWARDMODE_ BUSYNAEXTERNAL	This constant indicates to forward all external calls on busy/no answer. Use this value when call forwarding on busy and on no answer cannot be controlled separately for internal calls.
LINEFORWARDMODE_ BUSYNASPECIFIC	This constant indicates to forward all calls that originated at a specified address on busy/no answer (selective call forwarding).

See Also

LINEADDRESSCAPS, LINECALLTREATMENTENTRY, lineGetDevCaps, lineNegotiateAPIVersion

structure LINECALLTREATMENTENTRY TAPI.pas

The LINECALLTREATMENTENTRY structure provides information on the type of call treatment, such as music, recorded announcement, or silence, on the current call. The LINEADDRESSCAPS structure can contain an array of LINECALLTREATMENTENTRY structures. It is defined as follows in TAPI.pas:

```
PLineCallTreatmentEntry = ^TLineCallTreatmentEntry;
linecalltreatmententry_tag = packed record
  dwCallTreatmentID,                               // TAPI v2.0
  dwCallTreatmentNameSize,                         // TAPI v2.0
  dwCallTreatmentNameOffset: DWORD;                // TAPI v2.0
end;
TLineCallTreatmentEntry = linecalltreatmententry_tag;
LINECALLTREATMENTENTRY = linecalltreatmententry_tag;
```

The fields of the structure are described in Table 8-9.

Table 8-9: Fields of the LINECALLTREATMENTENTRY structure

Field	Member
dwCallTreatmentID	This field is one of the LINECALLTREATMENT_ constants (if the treatment is of a predefined type) or a service provider-specific value. Those constants are: LINECALLTREATMENT_BUSY indicates that when the call is not actively connected to a device (offering or onhold), the party hears a busy signal. LINECALLTREATMENT_MUSIC indicates that when the call is not actively connected to a device (offering or onhold), the party hears music. LINECALLTREATMENT_RINGBACK indicates that when the call is not actively connected to a device (offering or onhold), the party hears ringback tone. LINECALLTREATMENT_SILENCE indicates that when the call is not actively connected to a device (offering or onhold), the party hears silence.
dwCallTreatmentNameSize	This field indicates the size, in bytes, (including the terminating NULL) of a NULL-terminated string identifying the treatment. This would ordinarily describe the content of the music or recorded announcement. If the treatment is of a predefined type, a meaningful name should still be specified (for example, “Silence\0,” “Busy Signal\0,” “Ringback\0,” or “Music\0.”)
dwCallTreatmentNameOffset	This field indicates the offset from the beginning of LINEADDRESSCAPS of a NULL-terminated string identifying the treatment. This would ordinarily describe the content of the music or recorded announcement. If the treatment is of a predefined type, a meaningful name should still be specified (for example, “Silence\0,” “Busy Signal\0,” “Ringback\0,” or “Music\0.”)

See Also

LINEADDRESSCAPS, lineGetAddressCaps, lineSetCallTreatment

function lineGetAddressID   TAPI.pas

Syntax

function lineGetAddressID(hLine: HLINE; var dwAddressID: DWORD; dwAddressMode: DWORD; lpsAddress: LPCSTR; dwSize: DWORD): Longint; stdcall;

Description

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

Parameters

hLine: A handle (HLINE) to the open line device

var dwAddressID: A pointer to a DWORD-sized memory location in which the address ID will be returned

dwAddressMode: A DWORD holding 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 LINEADDRESSMODE_ constant LINEADDRESSMODE_DIALABLEADDR, which indicates that the address is specified by its dialable address. The *lpsAddress* parameter is the dialable address or canonical address format.

lpsAddress: A pointer (LPCSTR) to a data structure holding the address assigned to the specified line device. The format of the address is determined by *dwAddressMode*. Because the only valid value is LINEADDRESSMODE_DIALABLEADDR, *lpsAddress* uses the common dialable number format and is NULL-terminated.

dwSize: A DWORD indicating the size of the address contained in *lpsAddress*.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALLINEHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALADDRESSMODE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALADDRESS, LINEERR_UNINITIALIZED, and LINEERR_NOMEM.

See Also

lineMakeCall

Example

Listing 8-5 shows how to get an address's ID.

Listing 8-5: Getting an address's ID

```
function TTapiInterface.GetAddressID: boolean;
begin
  TapiResult := lineGetAddressID(fLine, fAddressID,
    LINEADDRESSMODE_DIALABLEADDR, PChar(FPhoneNumber),
    SizeOf(FPhoneNumber));
  result := TapiResult=0;
  if not result then ReportError(TapiResult);
end;
```

function lineGetAddressStatus TAPI.pas

Syntax

function lineGetAddressStatus(hLine: HLINE; dwAddressID: DWORD; lpAddressStatus: PLineAddressStatus): Longint; stdcall;

Description

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

Parameters

hLine: A handle (HLINE) to the open line device

dwAddressID: A DWORD indicating an address on the given open line device—the address to be queried *lpAddressStatus*: A pointer (PLineAddressStatus) to a variably sized data structure of type LINEADDRESSSTATUS. Before you call lineGetAddressStatus(), you should set the *dwTotalSize* field of the LINEADDRESSSTATUS structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALADDRESSID, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINEHANDLE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_OPERATIONUNAVAIL, and LINEERR_OPERATIONFAILED.

See Also

LINEADDRESSSTATUS

Example

Listing 8-6 shows how to get the status of an address on an open line.

Listing 8-6: Getting the status of an address on an open line

```
function TTapiInterface.GetAddressStatus: boolean;
var
  ATAPIResult: LongInt;
begin
  if CallState <> csConnected then
  begin
    ShowMessage('Call must be connected to get address status');
    exit;
  end;
  if fLineAddressStatus=Nil then
    fLineAddressStatus := AllocMem(SizeOf(LineAddressStatus)+1000);
  fLineAddressStatus.dwTotalSize := SizeOf(LineAddressStatus)+1000;
  TapiResult := lineGetAddressStatus(fLine, fAddressID,
    fLineAddressStatus);
  result := TapiResult=0;
  if NOT result then ReportError(TapiResult);
end;
```

structure LINEADDRESSSTATUS TAPI.pas

The LINEADDRESSSTATUS structure describes the current status of an address. It is defined as follows in TAPI.pas:

```
 PLineAddressStatus = ^TLineAddressStatus;
  lineaddressstatus_tag = packed record
    dwTotalSize,
    dwNeededSize,
    dwUsedSize,
    dwNumInUse,
    dwNumActiveCalls,
    dwNumOnHoldCalls,
    dwNumOnHoldPendCalls,
```

```
    dwAddressFeatures,
    dwNumRingsNoAnswer,
    dwForwardNumEntries,
    dwForwardSize,
    dwForwardOffset,
    dwTerminalModesSize,
    dwTerminalModesOffset,
    dwDevSpecificSize,
    dwDevSpecificOffset: DWORD;
  end;
  TLineAddressStatus = lineaddressstatus_tag;
  LINEADDRESSSTATUS = lineaddressstatus_tag;
```

The parameters of the LINEADDRESSSTATUS structure are explained in Table 8-10.

Table 8-10: Parameters of the LINEADDRESSSTATUS structure

Parameter	Meaning
dwTotalSize	This field specifies the total size in bytes allocated to this data structure.
dwNeededSize	This field specifies the size in bytes for this data structure that is needed to hold all the returned information.
dwUsedSize	This field specifies the size in bytes of the portion of this data structure that contains useful information.
dwNumInUse	This field specifies the number of stations that are currently using the address.
dwNumActiveCalls	This field specifies the number of calls on the address that are in call states other than idle, onHold, onHoldPendingTransfer, and onHoldPendingConference.
dwNumOnHoldCalls	This field specifies the number of calls on the address in the onHold state.
dwNumOnHoldPendCalls	This field specifies the number of calls on the address in the onHoldPendingTransfer or onHoldPendingConference state.
dwAddressFeatures	This field specifies the address-related API functions that can be invoked on the address in its current state. It uses the following LINEADDRFEATURE_ constants (the full list of constants is given in Table 8-11): LINEADDRFEATURE_FORWARD indicates the address can be forwarded. LINEADDRFEATURE_MAKECALL indicates an outbound call can be placed on the address. LINEADDRFEATURE_PICKUP indicates a call can be picked up at the address. LINEADDRFEATURE_SETMEDIACONTROL indicates media control can be set on this address. LINEADDRFEATURE_SETTERMINAL indicates the terminal modes for this address can be set. LINEADDRFEATURE_SETUPCONF indicates a conference call with a NULL initial call can be set up at this address. LINEADDRFEATURE_UNCOMPLETECALL indicates call completion requests can be canceled at this address. LINEADDRFEATURE_UNPARK indicates calls can be unparked using this address.
dwNumRingsNoAnswer	This field specifies the number of rings set for this address before an unanswered call is considered as no answer.

Parameter	Meaning
dwForwardNumEntries	The number of entries in the array referred to by dwForwardSize and dwForwardOffset.
dwForwardSize	The size in bytes of the data structure of the variably sized field that describes the address's forwarding information. This information is an array of dwForwardNum-Entries elements of type LINEFORWARD. The offsets of the addresses in the array are relative to the beginning of the LINEADDRESSSTATUS structure. The offsets dwCallerAddressOffset and dwDestAddressOffset in the variably sized field of type LINEFORWARD pointed to by dwForwardSize and dwForwardOffset are relative to the beginning of the LINEADDRESSSTATUS data structure (the "root" container).
dwForwardOffset	This field specifies the offset in bytes from the beginning of the data structure of the variably sized field that describes the address's forwarding information. This information is an array of dwForwardNumEntries elements of type LINEFORWARD. The offsets of the addresses in the array are relative to the beginning of the LINEADDRESSSTATUS structure. The offsets dwCallerAddressOffset and dwDestAddressOffset in the variably sized field of type LINEFORWARD pointed to by dwForwardSize and dwForwardOffset are relative to the beginning of the LINEADDRESSSTATUS data structure (the "root" container).
dwTerminalModesSize	This field specifies the size in bytes of the data structure of the variably sized device field containing an array with DWORD-sized entries that use the LINETERMMODE_ constants. This array is indexed by terminal IDs, in the range from zero to one less than dwNumTerminals. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the lineSetTerminal() function for this address. Values are: LINETERMMODE_LAMPS indicates that these are lamp events sent from the line to the terminal. LINETERMMODE_BUTTONS indicates that these are button-press events sent from the terminal to the line. LINETERMMODE_DISPLAY indicates that this is display information sent from the line to the terminal. LINETERMMODE_RINGER indicates that this is ringer-control information sent from the switch to the terminal. LINETERMMODE_HOOKSWITCH indicates that these are hookswitch events sent between the terminal and the line. LINETERMMODE_MEDIATOLINE indicates that this is the unidirectional media stream from the terminal to the line associated with a call on the line (you should use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently). LINETERMMODE_MEDIAFROMLINE indicates that this is the unidirectional media stream from the line to the terminal associated with a call on the line (you should use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently). LINETERMMODE_MEDIABIDIRECT indicates that this is the bidirectional media stream associated with a call on the line and the terminal (you should use this value when the routing of both unidirectional channels of a call's media stream cannot be controlled independently).

Parameter	Meaning
dwTerminalModesOffset	This field specifies the offset in bytes from the beginning of this data structure of the variably sized device field containing an array with DWORD-sized entries that use the LINETERMMODE_ constants. This array is indexed by terminal IDs, in the range from zero to one less than dwNumTerminals. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the lineSetTerminal() function for this address. Values are: LINETERMMODE_LAMPS indicates that these are lamp events sent from the line to the terminal. LINETERMMODE_BUTTONS indicates that these are button-press events sent from the terminal to the line. LINETERMMODE_DISPLAY indicates that this is display information sent from the line to the terminal. LINETERMMODE_RINGER indicates that this is ringer-control information sent from the switch to the terminal. LINETERMMODE_HOOKSWITCH indicates that these are hookswitch events sent between the terminal and the line. LINETERMMODE_MEDIATOLINE indicates that this is the unidirectional media stream from the terminal to the line associated with a call on the line (you should use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently). LINETERMMODE_MEDIAFROMLINE indicates that this is the unidirectional media stream from the line to the terminal associated with a call on the line (you should use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently). LINETERMMODE_MEDIABIDIRECT indicates that this is the bidirectional media stream associated with a call on the line and the terminal (you should use this value when the routing of both unidirectional channels of a call's media stream cannot be controlled independently).
deDevSpecificSize	This field specifies the size in bytes of the data structure's variably sized device-specific field.
dwDevSpecificOffset	This field specifies the offset in bytes from the beginning of this data structure's variably sized device-specific field.

Device-specific extensions should use the DevSpecific (*dwDevSpecificSize* and *dwDevSpecificOffset*) variably sized area of this data structure.

This data structure is returned by lineGetAddressStatus(). When items in this data structure change as a consequence of activities on the address, a LINE_ADDRESSSTATE message will be sent to the application. A parameter to this message is the address state, the constant LINEADDRESSSTATE_, which is an indication that the status item in this record changed.

See Also

LINE_ADDRESSSTATE, LINEFORWARD, lineGetAddressStatus, lineSetTerminal

Example

Listing 8-7 shows how to query the *lpAddressStatus* field of the lineGetAddressStatus() function to retrieve information about a particular call.

Listing 8-7: Querying the lpAddressStatus field of lineGetAddressStatus() to retrieve call information

```
procedure TForm1.btnTestGetAddressStatusClick(Sender: TObject);
var
  ALineAddressStatus : pLineAddressStatus;
begin
  ALineAddressStatus := AllocMem(SizeOf(TLineAddressStatus)+1000);
  ALineAddressStatus.dwTotalSize := SizeOf(TLineAddressStatus)+1000;
  if NOT TapiInterface.GetAddressStatus(ALineAddressStatus) then
    ShowMessage('Could not get address status')
  else
    begin
      cbLineForward.Checked := DWordIsSet(ALineAddressStatus^.dwAddressFeatures,
          LINEADDRFEATURE_FORWARD);
      cbLineMakeCall.Checked := DWordIsSet(ALineAddressStatus^.dwAddressFeatures,
           LINEADDRFEATURE_MAKECALL);
      cbLinePickup.Checked := DWordIsSet(ALineAddressStatus^.dwAddressFeatures,
           LINEADDRFEATURE_PICKUP);
      cbLineSetMediaControl.Checked :=
        DWordIsSet(ALineAddressStatus^.dwAddressFeatures,
           LINEADDRFEATURE_SETMEDIACONTROL);
      cbLineSetTerminal.Checked :=
        DWordIsSet(ALineAddressStatus^.dwAddressFeatures,
          LINEADDRFEATURE_SETTERMINAL);
      cbLineSetupConf.Checked := DWordIsSet(ALineAddressStatus^.dwAddressFeatures,
         LINEADDRFEATURE_SETUPCONF);
      cbLineUncompleteCall.Checked :=
        DWordIsSet(ALineAddressStatus^.dwAddressFeatures,
          LINEADDRFEATURE_UNCOMPLETECALL);
      cbLineUnpark.Checked := DWordIsSet(ALineAddressStatus^.dwAddressFeatures,
        LINEADDRFEATURE_UNPARK);
    end;
  FreeMem(ALineAddressStatus);
  ALineAddressStatus := Nil;
end;
```

LINEADDRFEATURE Constants

The LINEADDRFEATURE_ constants list the operations that can be invoked on an address. These constants are explained in Table 8-11.

Table 8-11: LINEADDRFEATURE_ constants

Constant	Meaning
LINEADDRFEATURE_ FORWARD	This constant indicates that the address can be forwarded.
LINEADDRFEATURE_ MAKECALL	This constant indicates that an outgoing call can be placed on the address.
LINEADDRFEATURE_ PICKUP	This constant indicates that a call can be picked up at the address.

Constant	Meaning
LINEADDRFEATURE_ PICKUPDIRECT	This constant indicates that the linePickup() function can be used to pick up a call on a specific address.
LINEADDRFEATURE_ PICKUPGROUP	This constant indicates that the linePickup() function can be used to pick up a call in the group.
LINEADDRFEATURE_ PICKUPHELD	This constant indicates that the linePickup() function (with a NULL destination address) can be used to pick up a call that is held on the address. This is normally used only in a bridged-exclusive arrangement.
LINEADDRFEATURE_ PICKUPWAITING	This constant indicates that the linePickup() function (with a NULL destination address) can be used to pick up a call waiting call. Note that this does not necessarily indicate that a waiting call is actually present because it is often impossible for a telephony device to automatically detect such a call; it does, however, indicate that the hook-flash function will be invoked to attempt to switch to such a call.
LINEADDRFEATURE_ SETMEDIACONTROL	This constant indicates that media control can be set on this address.
LINEADDRFEATURE_ SETTERMINAL	This constant indicates that the terminal modes for this address can be set.
LINEADDRFEATURE_ SETUPCONF	This constant indicates that a conference call with a NULL initial call can be set up at this address.
LINEADDRFEATURE_ UNCOMPLETECALL	This constant indicates that call completion requests can be canceled at this address.
LINEADDRFEATURE_ UNPARK	This constant indicates that calls can be unparked using this address. Note: If none of the new modified "PICKUP" bits is set in the dwAddressFeatures member in LINE-ADDRESSSTATUS, but the LINEADDRFEATURE_PICKUP bit is set, then any of the pickup modes may work; the service provider has simply not specified which ones.
LINEADDRFEATURE_ FORWARDDND	This constant indicates that the lineForward() function (with an empty destination address) can be used to turn on the Do Not Disturb feature on the address. LINEADDRFEATURE_FORWARD will also be set.
LINEADDRFEATURE_ FORWARDFWD	This constant indicates that the lineForward() function can be used to forward calls on the address to other numbers. LINEADDRFEATURE_FORWARD will also be set. Note: If neither of the new modified "FORWARD" bits is set in the dwAddressFeatures member in LINEADDRESSSTATUS, but the LINEADDRFEATURE_FORWARD bit is set, then any of the forward modes may work; the service provider has simply not specified which ones. No extensibility. All 32 bits are reserved. This constant is used both in LINE-ADDRESSCAPS (returned by lineGetAddressCaps()) and in LINEADDRESSSTATUS (returned by lineGetAddressStatus()). LINEADDRESSCAPS reports the availability of the address features by the service provider (mainly the switch) for a given address. An application would make this determination when it initializes. The LINEADDRESS-STATUS structure reports, for a given address, which address features can actually be invoked while the address is in the current state. An application would make this determination dynamically after address-state changes, typically caused by call-related activities on the address.

See Also

LINEADDRESSCAPS, LINEADDRESSSTATUS, lineForward, lineGetAddress-Caps, lineGetAddressStatus, linePickup

function lineGetDevCaps TAPI.pas

Syntax

function lineGetDevCaps(hLineApp: HLINEAPP; dwDeviceID, dwAPIVersion, dwExtVersion: DWORD; lpLineDevCaps: PLineDevCaps): Longint; stdcall;

Description

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

Parameters

hLineApp: The handle (HLINEAPP) to the application's registration with TAPI

dwDeviceID: A DWORD indicating the line device to be queried

dwAPIVersion: A DWORD indicating 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: A DWORD indicating the version number of the service provider-specific extensions to be used. This number is obtained by lineNegotiateExtVersion(). It can be left at zero if no device-specific extensions are to be used. Otherwise, the high-order word contains the major version number, and the low-order word contains the minor version number.

lpLineDevCaps: A pointer (PLineDevCaps) 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 the LINEDEVCAPS structure to indicate the amount of memory available to TAPI for returning information.

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_OPERATIONFAILED, LINEERR_INCOMPATIBLEEXTVERSION, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALAPPHANDLE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NODRIVER, LINEERR_OPERATIONUNAVAIL, and LINEERR_NODEVICE.

See Also

LINEDEVCAPS, lineGetAddressCaps, lineNegotiateAPIVersion, lineNegotiateExtVersion

Example

Listing 8-8 shows how to retrieve an address's capabilities.

Listing 8-8: Retrieving an address's capabilities

```
// Caller must inialize TempLineDevCaps to proper memory size
function TTapiInterface.GetDeviceCaps(DeviceNum : DWord;
         var TempLineDevCaps: PLINEDEVCAPS) : boolean;
var
   LocalString : String;
begin
  TempLineDevCaps^.dwTotalSize := FDeviceCapsAllocSize;
  TAPIResult := LineGetDevCaps(fLineApp, DeviceNum, FApiVersion, 0,
    TempLineDevCaps);
  result := TAPIResult=0;
  if NOT result then ReportError(TAPIResult);
  if NOT GetAddressCapsSize(FAddressCapsAllocSize)
    then ShowMessage('Could not get address caps size');
end;
```

structure LINEDEVCAPS TAPI.pas

The LINEDEVCAPS structure contains information about the capabilities of a line device. Device-specific extensions use the DevSpecific variably sized portion of this data structure (defined by *dwDevSpecificSize* and *dwDevSpecificOffset*) to store their information. Note that older applications using earlier versions of TAPI will likely have been compiled without the variable sized field in the LINEDEVCAPS structure. So, you cannot simply use SizeOf(LINEDEVCAPS) to set the memory for this and other structures that contain variable portions. In our sample code, we use SizeOf(TLineDevCaps)+1000; we also use this approach with similar structures with variably sized portions.

Your application must pass a *dwAPIVersion* parameter when calling the lineGetDevCaps() function. TAPI uses this value to receive guidance when retrieving capabilities. If your application uses a *dwTotalSize* value that is less than the size of the fixed portion of the structure (as defined in the *dwAPIVersion* specified), the function will return an error of LINEERR_STRUCTURETOOSMALL. You may want to check for this particular error and reallocate more memory if it is returned.

If you allocate sufficient memory in your application before calling the lineGetDevCaps() function (which in turn calls the related TSPI function), TAPI will set the *dwNeededSize* and *dwUsedSize* fields to the fixed size of the structure as it existed in the specified TAPI version. Service providers are responsible for examining the version of TAPI and taking appropriate action (see the TAPI Help file for additional details).

The LINEDEVCAPS structure is defined as follows in TAPI.pas:

```
 PLineDevCaps = ^TLineDevCaps;
 linedevcaps_tag = packed record
   dwTotalSize,
   dwNeededSize,
   dwUsedSize,
   dwProviderInfoSize,
   dwProviderInfoOffset,
   dwSwitchInfoSize,
   dwSwitchInfoOffset,
   dwPermanentLineID,
   dwLineNameSize,
   dwLineNameOffset,
   dwStringFormat,
   dwAddressModes,
   dwNumAddresses,
   dwBearerModes,
   dwMaxRate,
   dwMediaModes,
   dwGenerateToneModes,
   dwGenerateToneMaxNumFreq,
   dwGenerateDigitModes,
   dwMonitorToneMaxNumFreq,
   dwMonitorToneMaxNumEntries,
   dwMonitorDigitModes,
   dwGatherDigitsMinTimeout,
   dwGatherDigitsMaxTimeout,
   dwMedCtlDigitMaxListSize,
   dwMedCtlMediaMaxListSize,
   dwMedCtlToneMaxListSize,
   dwMedCtlCallStateMaxListSize,
   dwDevCapFlags,
   dwMaxNumActiveCalls,
   dwAnswerMode,
   dwRingModes,
   dwLineStates,
   dwUUIAcceptSize,
   dwUUIAnswerSize,
   dwUUIMakeCallSize,
   dwUUIDropSize,
   dwUUISendUserUserInfoSize,
   dwUUICallInfoSize: DWORD;
   MinDialParams,
   MaxDialParams,
   DefaultDialParams: TLineDialParams;
   dwNumTerminals,
   dwTerminalCapsSize,
   dwTerminalCapsOffset,
   dwTerminalTextEntrySize,
   dwTerminalTextSize,
   dwTerminalTextOffset,
   dwDevSpecificSize,
   dwDevSpecificOffset,
   dwLineFeatures: DWORD;                                 // TAPI v1.4
{$IFDEF TAPI20}
   dwSettableDevStatus,                                   // TAPI v2.0
   dwDeviceClassesSize,                                   // TAPI v2.0
   dwDeviceClassesOffset: DWORD;                          // TAPI v2.0
{$ENDIF}
```

```
{$IFDEF TAPI22}
    PermanentLineGuid: TGUID;                         // TAPI v2.2
{$ENDIF}
{$IFDEF TAPI30}
    dwAddressTypes: DWORD;                            // TAPI v3.0
    ProtocolGuid: TGUID;                              // TAPI v3.0
    dwAvailableTracking: DWORD;                       // TAPI v3.0
{$ENDIF}
  end;
  TLineDevCaps = linedevcaps_tag;
  LINEDEVCAPS = linedevcaps_tag;
```

The LINEDEVCAPS structure describes the capabilities of a line device. Its many fields are described in Table 8-12.

Table 8-12: Fields of the LINEDEVCAPS structure

Field	Meaning
dwTotalSize	This field specifies the total size in bytes allocated to this data structure.
dwNeededSize	This field specifies the size in bytes for this data structure that is needed to hold all the returned information.
dwUsedSize	This field specifies the size in bytes of the portion of this data structure that contains useful information.
dwProviderInfoSize	This field specifies the size in bytes of the variably sized field containing service provider information. The dwProviderInfoSize/Offset field pair is intended to provide information about the provider hardware and/or software, such as the vendor name and version numbers of hardware and software. This information can be useful when a user needs to call customer service with problems regarding the provider.
dwProviderInfoOffset	This field specifies the offset in bytes from the beginning of this data structure of the variably sized field containing service provider information. The dwProviderInfoSize/Offset field pair is intended to provide information about the provider hardware and/or software, such as the vendor name and version numbers of hardware and software. This information can be useful when a user needs to call customer service with problems regarding the provider.
dwSwitchInfoSize	This field specifies the size in bytes of the variably sized device field containing switch information. The dwSwitchInfoSize/Offset field pair is intended to provide information about the switch to which the line device is connected, such as the switch manufacturer, the model name, the software version, and so on. This information can be useful when a user needs to call customer service with problems regarding the switch.
dwSwitchInfoOffset	This field specifies the offset in bytes from the beginning of this data structure of the variably sized device field containing switch information. The dwSwitchInfoSize/Offset field pair is intended to provide information about the switch to which the line device is connected, such as the switch manufacturer, the model name, the software version, and so on. This information can be useful when a user needs to call customer service with problems regarding the switch.
dwPermanentLineID	This field specifies the permanent DWORD identifier by which the line device is known in the system's configuration. It is a permanent name for the line device. This permanent name (as opposed to dwDevice ID) does not change as lines are added or removed from the system. It can therefore be used to link line-specific information in INI files (or other files) in a way that is not affected by adding or removing other lines.

Field	Meaning
dwLineNameSize	This field specifies the size in bytes of the variably sized device field containing a user configurable name for this line device. This name can be configured by the user when configuring the line device's service provider and is provided for the user's convenience.
dwLineNameOffset	This field specifies the offset in bytes from the beginning of this data structure of the variably sized device field containing a user configurable name for this line device. This name can be configured by the user when configuring the line device's service provider and is provided for the user's convenience.
dwStringFormat	This field specifies the string format used with this line device. It uses the following STRINGFORMAT_ constants: STRINGFORMAT_ASCII indicates the ASCII string format using one byte per character. STRINGFORMAT_DBCS indicates the DBCS string format using two bytes per character. STRINGFORMAT_ UNICODE, indicating the Unicode string format using two bytes per character.
dwAddressModes	This field specifies the mode by which the originating address is specified. This field uses the LINEADDRESSMODE_ constants.
dwNumAddresses	This field specifies the number of addresses associated with this line device. Individual addresses are referred to by address IDs. Address IDs range from zero to one less than the value indicated by dwNumAddresses.
dwBearerModes	This field is a flag array that specifies the different bearer modes that the address is able to support. It uses the following LINEBEARERMODE_ constants: LINEBEARERMODE_VOICE indicates a regular 3.1kHz analog voice grade bearer service (bit integrity is not assured; voice can support fax and modem media modes). LINEBEARERMODE_SPEECH indicates 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. Also, speech is not intended to support fax and modem media modes). LINEBEARERMODE_MULTIUSE indicates multi-use mode defined by ISDN. LINEBEARERMODE_DATA indicates the unrestricted data transfer on the call (the data rate is specified separately). LINEBEARERMODE_ALTSPEECHDATA indicates the alternate transfer of speech or unrestricted data on the same call (ISDN). LINEBEARERMODE_NONCALLSIGNALING indicates a non-call-associated signaling connection from the application to the service provider or switch (treated as a "media stream" by the Telephony API). LINEBEARERMODE_PASSTHROUGH indicates that the service provider will give direct access to the attached hardware for control by the application (this mode is used primarily by applications desiring temporary direct control over asynchronous modems accessed via the Win32 comm functions for the purpose of configuring or using special features not otherwise supported by the service provider).
dwMaxRate	This field contains the maximum data rate in bits per second for information exchange over the call.
dwMediaModes	This field is a flag array that specifies the different media modes the address is able to support. It uses the LINEMEDIAMODE_ constants described in Table 8-24.

Field	Meaning
dwGenerateToneModes	This field specifies the different kinds of tones that can be generated on this line. It uses the following LINETONEMODE_ constants: LINETONEMODE_CUSTOM indicates that the tone is a custom tone defined by the specified frequencies. LINETONEMODE_RINGBACK indicates that the tone to be generated is ringback tone. LINETONEMODE_BUSY indicates that the tone is a standard (station) busy tone. LINETONEMODE_BEEP indicates that the tone is a beep, as used to announce the beginning of a recording. LINETONEMODE_BILLING indicates that the tone is a billing information tone, such as a credit card prompt tone.
dwGenerateToneMaxNumFreq	This field specifies the maximum number of frequencies that can be specified in describing a general tone using the LINEGENERATETONE data structure when generating a tone using lineGenerateTone(). A value of zero indicates that tone generation is not available.
dwGenerateDigitModes	This field specifies the digit modes that can be generated on this line. It uses the following LINEDIGITMODE_ constants: LINEDIGITMODE_PULSE indicates to generate digits as pulse/rotary pulse sequences. LINEDIGITMODE_DTMF indicates to generate digits as DTMF tones.
dwMonitorToneMaxNumFreq	This field specifies the maximum number of frequencies that can be specified in describing a general tone using the LINEMONITORTONE data structure when monitoring a general tone using lineMonitorTones(). A value of zero indicates that tone monitor is not available.
dwMonitorToneMaxNumEntries	This field specifies the maximum number of entries that can be specified in a tone list to lineMonitorTones().
dwMonitorDigitModes	This field specifies the digit modes that can be detected on this line. It uses the following LINEDIGITMODE_ constants: LINEDIGITMODE_PULSE indicates to detect digits as audible clicks that are the result of rotary pulse sequences. LINEDIGITMODE_DTMF indicates to detect digits as DTMF tones. LINEDIGITMODE_DTMFEND indicates to detect the down edges of digits detected as DTMF tones.
dwGatherDigitsMinTimeout	This field specifies the minimum values in milliseconds that can be specified for both the first digit and inter-digit timeout values used by lineGatherDigits(). If both this field and the next are zero, timeouts are not supported.
dwGatherDigitsMaxTimeout	This field specifies the maximum values in milliseconds that can be specified for both the first digit and inter-digit timeout values used by lineGatherDigits(). If both this field and the previous are zero, timeouts are not supported.
dwMedCtlDigitMaxListSize	This field specifies the maximum number of entries that can be specified in the digit list parameter of lineSetMediaControl().
dwMedCtlMediaMaxListSize	This field specifies the maximum number of entries that can be specified in the media list parameter of lineSetMediaControl().
dwMedCtlToneMaxListSize	This field specifies the maximum number of entries that can be specified in the tone list parameter of lineSetMediaControl().
dwMedCtlCallStateMaxListSize	This field specifies the maximum number of entries that can be specified in the call state list parameter of lineSetMediaControl().
dwDevCapFlags	This field specifies various Boolean device capabilities. It uses the LINEDEVCAPFLAGS_ constants described in Table 8-13.

Field	Meaning
dwMaxNumActiveCalls	This field specifies the maximum number of (minimum bandwidth) calls that can be active (connected) on the line at any one time. The actual number of active calls may be lower if higher bandwidth calls have been established on the line.
dwAnswerMode	This field specifies the effect on the active call when answering another offering call on a line device. This field uses the following LINEANSWERMODE_ constants: LINEANSWERMODE_NONE indicates that answering another call on the same line has no effect on the existing active call(s) on the line. LINEANSWERMODE_DROP indicates that the currently active call will be automatically dropped. LINEANSWERMODE_HOLD indicates that the currently active call will automatically be placed on hold.
dwRingModes	This field specifies the number of different ring modes that can be reported in the LINE_LINEDEVSTATE message with the ringing indication. Different ring modes range from one to dwRingModes. Zero indicates no ring.
dwLineStates	This field specifies the different line status components for which the application may be notified in a LINE_LINEDEVSTATE message on this line. It uses the LINEDEVSTATE_ constants described in Table 8-14.
dwUUIAcceptSize	This field specifies the maximum size of user-to-user information that can be sent during a call accept.
dwUUIAnswerSize	This field specifies the maximum size of user-to-user information that can be sent during a call answer.
dwUUIMakeCallSize	This field specifies the maximum size of user-to-user information that can be sent during a make call.
dwUUIDropSize	This field specifies the maximum size of user-to-user information that can be sent during a call drop.
dwUUISendUserUserInfoSize	This field specifies the maximum size of user-to-user information that can be sent separately any time during a call with lineSendUserUserInfo.
dwUUICallInfoSize	This field specifies the maximum size of user-to-user information that can be received in the LINECALLINFO structure.
MinDialParams	This field specifies the minimum values for the dial parameters (in milliseconds) that can be set for calls on this line. Dialing parameters can be set to values in this range. The granularity of the actual settings is service provider-specific.
MaxDialParams	This field specifies the maximum values for the dial parameters in milliseconds that can be set for calls on this line. Dialing parameters can be set to values in this range. The granularity of the actual settings is service provider-specific.
DefaultDialParams	This field specifies the default dial parameters used for calls on this line. These parameter values can be overridden on a per-call basis.
dwNumTerminals	This field specifies the number of terminals that can be set for this line device, its addresses, or its calls. Individual terminals are referred to by terminal IDs and range from zero to one less than the value indicated by dwNumTerminals.
dwTerminalCapsSize	This field specifies the size in bytes of the variably sized device field containing an array with entries of type LINETERMCAPS. This array is indexed by terminal IDs, in the range from zero to dwNumTerminals minus one. Each entry in the array specifies the terminal device capabilities of the corresponding terminal.

Field	Meaning
dwTerminalCapsOffset	This field specifies the offset in bytes from the beginning of this data structure of the variably sized device field containing an array with entries of type LINETERMCAPS. This array is indexed by terminal IDs, in the range from zero to dwNumTerminals minus one. Each entry in the array specifies the terminal device capabilities of the corresponding terminal.
dwTerminalTextEntrySize	This field specifies the size in bytes of each of the terminal text descriptions pointed at by dwTerminalTextSize/Offset.
dwTerminalTextSize	This field specifies the size in bytes of the variably sized field containing descriptive text about each of the line's available terminals. Each message is dwTerminalTextEntrySize bytes long. The string format of these textual descriptions is indicated by dwStringFormat in the line's device capabilities.
dwTerminalTextOffset	This field specifies the offset in bytes from the beginning of this data structure of the variably sized field containing descriptive text about each of the line's available terminals. Each message is dwTerminalTextEntrySize bytes long. The string format of these textual descriptions is indicated by dwStringFormat in the line's device capabilities.
dwDevSpecificSize	This field specifies the size in bytes of the variably sized device-specific field.
dwDevSpecificOffset	This field specifies the offset in bytes from the beginning of this data structure of the variably sized device-specific field.
dwLineFeatures	This field specifies the features available for this line using the LINEFEATURE_ constants shown in Table 8-15. Invoking a supported feature requires the line to be in the proper state and the underlying line device to be opened in a compatible mode. A zero in a bit position indicates that the corresponding feature is never available. A one indicates that the corresponding feature may be available if the line is in the appropriate state for the operation to be meaningful. This field allows an application to discover which line features can be (and which can never be) supported by the device.

Table 8-13: LINEDEVCAPFLAGS_ constants used in the dwDevCapFlags field of the LINEDEVCAPS structure

Constant	Meaning
LINEDEVCAPFLAGS_ CROSSADDRCONF	This constant specifies whether calls on different addresses on this line can be added to a conference call.
LINEDEVCAPFLAGS_ HIGHLEVCOMP	This constant specifies whether high-level compatibility information elements are supported on this line.
LINEDEVCAPFLAGS_ LOWLEVCOMP	This constant specifies whether low-level compatibility information elements are supported on this line.
LINEDEVCAPFLAGS_ MEDIACONTROL	This constant specifies whether media control operations are available for calls at this line.
LINEDEVCAPFLAGS_ MULTIPLEADDR	This constant specifies whether lineMakeCall() or lineDial() can deal with multiple addresses at once (such as for inverse multiplexing).
LINEDEVCAPFLAGS_ CLOSEDROP	This constant specifies what happens when an open line is closed while the application has calls active on the line. If TRUE (set), then 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() function call will not automatically drop any calls that are still active on the line if the service provider knows that some other device can keep the call alive.

Constant	Meaning
LINEDEVCAPFLAGS_ CLOSEDROP (cont.)	For example, if an analog line has the computer and phoneset both connected 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.
LINEDEVCAPFLAGS_ DIALBILLING	The remaining three flag constants indicate whether the "$," "@," or "W" dialable string modifier is supported for a given line device (see discussion of dialable addresses in Chapter 10). It is TRUE if the modifier is supported; otherwise, FALSE. Note that the "?" (prompt user to continue dialing) is never supported by a line device. These flags allow an application to determine "up front" which modifiers would result in the generation of a LINEERR. The application has the choice of pre-scanning dialable strings for unsupported characters or passing the "raw" string from lineTranslateAddress() directly to the provider as part of lineMakeCall() (lineDial(), etc.) and let the function generate an error to tell it which unsupported modifier occurs first in the string.
LINEDEVCAPFLAGS_ DIALQUIET	See LINEDEVCAPFLAGS_DIALBILLING.
LINEDEVCAPFLAGS_ DIALDIALTONE	See LINEDEVCAPFLAGS_DIALBILLING.

Table 8-14: LINEDEVSTATE_ constants used with the dwLineStates field of the LINEDEVCAPS structure

Constant	Meaning
LINEDEVSTATE_OTHER	This constant specifies that device-status items other than those listed below have changed. The application should check the current device status to determine which items have changed.
LINEDEVSTATE_RINGING	This constant indicates that the switch tells the line to alert the user.
LINEDEVSTATE_ CONNECTED	This constant indicates that the line was previously disconnected and is now connected to TAPI.
LINEDEVSTATE_ DISCONNECTED	This constant indicates that the line was previously connected and is now disconnected from TAPI.
LINEDEVSTATE_MSGWAITON	This constant indicates that the "message waiting" indicator is turned on.
LINEDEVSTATE_ MSGWAITOFF	This constant indicates that the "message waiting" indicator is turned off.
LINEDEVSTATE_ NUMCOMPLETIONS	This constant indicates that the number of outstanding call completions on the line device has changed.
LINEDEVSTATE_INSERVICE	This constant indicates that the line is connected to TAPI. This happens when TAPI is first activated or when the line wire is physically plugged in and in service at the switch while TAPI is active.
LINEDEVSTATE_ OUTOFSERVICE	This constant indicates that the line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.
LINEDEVSTATE_ MAINTENANCE	This constant indicates that maintenance is being performed on the line at the switch. TAPI cannot be used to operate on the line device.
LINEDEVSTATE_OPEN	This constant indicates that the line has been opened.
LINEDEVSTATE_CLOSE	This constant indicates that the line has been closed.
LINEDEVSTATE_NUMCALLS	This constant indicates that the number of calls on the line device has changed.
LINEDEVSTATE_TERMINALS	This constant indicates that the terminal settings have changed.
LINEDEVSTATE_ROAMMODE	This constant indicates that the roam mode of the line device has changed.

Constant	Meaning
LINEDEVSTATE_BATTERY	This constant indicates that the battery level has changed significantly (cellular).
LINEDEVSTATE_SIGNAL	This constant indicates that the signal level has changed significantly (cellular).
LINEDEVSTATE_DEVSPECIFIC	This constant indicates that the line's device-specific information has changed.
LINEDEVSTATE_REINIT	This constant indicates that items have changed in the configuration of line devices. To become aware of these changes (such as for the appearance of new line devices), the application should reinitialize its use of TAPI. The hDevice parameter is left NULL for this state change as it applies to any of the lines in the system.
LINEDEVSTATE_LOCK	This constant indicates that the locked status of the line device has changed.
LINEDEVSTATE_ CAPSCHANGE	This constant indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the LINEDEVCAPS structure for the address have changed. The application should use lineGetDevCaps() to read the updated structure. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated this or a subsequent API version; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINEDEVSTATE_REINIT, requiring them to shut down and reinitialize their connection to TAPI in order to obtain the updated information.
LINEDEVSTATE_ CONFIGCHANGE	This constant indicates that configuration changes have been made to one or more of the media devices associated with the line device. The application, if it desires, may use lineGetDevConfig() to read the updated information. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated this or a subsequent API version; applications negotiating a previous API version will not receive any notification.
LINEDEVSTATE_ TRANSLATECHANGE	This constant indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the LINETRANSLATECAPS structure have changed. The application should use lineGetTranslateCaps() to read the updated structure. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated this or a subsequent API version; applications negotiating a previous API version will receive LINE_LINEDEVSTATE messages specifying LINEDEVSTATE_REINIT, requiring them to shut down and reinitialize their connection to TAPI in order to obtain the updated information.
LINEDEVSTATE_ COMPLCANCEL	This constant indicates that the call completion identified by the completion ID contained in parameter dwParam2 of the LINE_LINEDEVSTATE message has been externally cancelled and is no longer considered valid (if that value were to be passed in a subsequent call to lineUncompleteCall(), the function would fail with LINEERR_INVALCOMPLETIONID). If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated this or a subsequent API version; applications negotiating a previous API version will not receive any notification.
LINEDEVSTATE_REMOVED	This constant indicates that the device is being removed from the system by the service provider (most likely through user action, via a control panel or similar utility). A LINE_LINEDEVSTATE message with this value will normally be immediately followed by a LINE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in LINEERR_ NODEVICE being returned to the application. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated this or a subsequent API version; applications negotiating a previous API version will not receive any notification.

LINEFEATURE_ Constants

The LINEFEATURE_ constants are defined in Table 8-15. They list the operations that can be invoked on a line using TAPI. The LINEFEATURE_ constants are used in LINEDEVSTATUS (returned by the lineGetLineDevStatus() function). LINEDEVSTATUS reports, for a given line, which line features can actually be invoked while the line is in the current state. An application would make this determination dynamically after line state changes, typically caused by address or call-related activities on the line.

Table 8-15: LINEFEATURE_ constants

Constant	Meaning
LINEFEATURE_DEVSPECIFIC	This constant indicates that device-specific operations can be used on the line.
LINEFEATURE_ DEVSPECIFICFEAT	This constant indicates that device-specific features can be used on the line.
LINEFEATURE_FORWARD	This constant indicates that forwarding of all addresses can be used on the line.
LINEFEATURE_ FORWARDDND	This constant indicates that the lineForward() function (with an empty destination address) can be used to turn on the Do Not Disturb feature on all addresses on the line. LINEFEATURE_FORWARD will also be set. This flag is exposed only to applications that negotiate a TAPI version of 2.0 or higher.
LINEFEATURE_ FORWARDFWD	This constant indicates that the lineForward() function can be used to forward calls on all addresses on the line to other numbers. LINEFEATURE_FORWARD will also be set. This flag is exposed only to applications that negotiate a TAPI version of 2.0 or higher.
LINEFEATURE_MAKECALL	This constant indicates that an outgoing call can be placed on this line using an unspecified address.
LINEFEATURE_ SETDEVSTATUS	This constant indicates that the lineSetLineDevStatus() function can be invoked on the line device. This flag is exposed only to applications that negotiate a TAPI version of 2.0 or higher.
LINEFEATURE_ SETMEDIACONTROL	This constant indicates that media control can be set on this line.
LINEFEATURE_SETTERMINAL	This constant indicates that terminal modes for this line can be set.

structure LINETERMCAPS TAPI.pas

The LINETERMCAPS structure describes the capabilities of a line's terminal device. This structure does not support extensions. It is defined as follows in TAPI.pas:

```
PLineTermCaps = ^TLineTermCaps;
linetermcaps_tag = packed record
  dwTermDev,
  dwTermModes,
  dwTermSharing: DWORD;
end;
TLineTermCaps = linetermcaps_tag;
LINETERMCAPS = linetermcaps_tag;
```

The fields of the LINETERMCAPS structure are described in Table 8-16.

Table 8-16: Fields of the LINETERMCAPS structure

Field	Meaning
dwTermDev	This field specifies the device type of the terminal. It uses the following LINETERMDEV_ constants: LINETERMDEV_PHONE indicates that the terminal is a phone set. LINETERMDEV_HEADSET indicates that the terminal is a headset. LINETERMDEV_SPEAKER indicates that the terminal is an external speaker and microphone.
dwTermModes	This field specifies the terminal mode(s) the device is able to deal with. It uses the following LINETERMMODE_ constants: LINETERMMODE_BUTTONS indicates that button-press events will be sent from the terminal to the line. LINETERMMODE_LAMPS indicates lamp events sent from the line to the terminal. LINETERMMODE_DISPLAY indicates display information was sent from the line to the terminal. LINETERMMODE_RINGER indicates that ringer-control information was sent from the switch to the terminal. LINETERMMODE_HOOKSWITCH indicates that hookswitch events were sent from the terminal to the line. LINETERMMODE_MEDIATOLINE indicates the unidirectional media stream from the terminal to the line associated with a call on the line (use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently). LINETERMMODE_MEDIAFROMLINE indicates the unidirectional media stream from the line to the terminal associated with a call on the line (use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently). LINETERMMODE_MEDIABIDIRECT indicates that this is the bidirectional media stream associated with a call on the line and the terminal (use this value when the routing of both unidirectional channels of a call's media stream cannot be controlled independently).
dwTermSharing	This field specifies how the terminal device is shared between line devices. It uses the following LINETERMSHARING_ constants: LINETERMSHARING_PRIVATE indicates that the terminal device is private to a single line device. LINETERMSHARING_SHAREDEXCL indicates that the terminal device can be used by multiple lines (the last line device to do a lineSetTerminal() to the terminal for a given terminal mode will have exclusive connection to the terminal for that mode). LINETERMSHARING_SHAREDCONF indicates that the terminal device can be used by multiple lines (the lineSetTerminal() requests of the various terminals end up being "merged" at the terminal).

structure LINETRANSLATECAPS TAPI.pas

The LINETRANSLATECAPS structure describes the address translation capabilities. This structure does not support extensions. It is defined as follows in TAPI.pas:

```
PLineTranslateCaps = ^TLineTranslateCaps;
linetranslatecaps_tag = packed record
  dwTotalSize,
  dwNeededSize,
  dwUsedSize,
  dwNumLocations,
  dwLocationListSize,
  dwLocationListOffset,
```

```
    dwCurrentLocationID,
    dwNumCards,
    dwCardListSize,
    dwCardListOffset,
    dwCurrentPreferredCardID: DWORD;
  end;
  TLineTranslateCaps = linetranslatecaps_tag;
  LINETRANSLATECAPS = linetranslatecaps_tag;
```

The fields of the LINETRANSLATECAPS structure are described in Table 8-17.

Table 8-17: Fields of the LINETRANSLATECAPS structure

Field	Meaning
dwTotalSize	This field specifies the total size in bytes allocated to this data structure.
dwNeededSize	This field specifies the size in bytes for this data structure that is needed to hold all the returned information.
dwUsedSize	This field specifies the size in bytes of the portion of this data structure that contains useful information.
dwNumLocations	This field specifies the number of entries in the LocationList. It includes all locations defined, including 0 (default).
dwLocationListSize	This field specifies the total number of bytes in the entire list of locations known to address translation. The list consists of a sequence of LINELOCATIONENTRY structures.
dwLocationListOffset	This field points to the first byte of the first LINELOCATIONENTRY structure in a list of locations known to address translation. The list consists of a sequence of LINELOCATIONENTRY structures.
dwCurrentLocationID	This field specifies the dwPermanentLocationID from the LINELOCATIONENTRY for the current location.
dwNumCards	This field specifies the number of entries in the CardList.
dwCardListSize	This field indicates the total number of bytes in the entire list of calling cards known to address translation. It includes only non-hidden card entries and always includes card 0 (direct dial). The list consists of a sequence of LINECARDENTRY structures.
dwCardListOffset	This field points to the first byte of the first LINECARDENTRY structure in the list of calling cards known to address translation. It includes only non-hidden card entries and always includes card 0 (direct dial). The list consists of a sequence of LINECARDENTRY structures.
dwCurrentPreferredCardID	This field specifies the dwPreferredCardID from the LINELOCATIONENTRY for the current location.

See Also

LINECARDENTRY, LINELOCATIONENTRY

structure LINECARDENTRY TAPI.pas

The LINECARDENTRY structure describes a calling card. The LINETRANSLATECAPS structure can contain an array of LINECARDENTRY structures. Older applications compiled with earlier TAPI versions will have no knowledge of these new fields. If they use SIZEOF(LINECARDENTRY), they may end up with a structure size that is too small. Because this is an array in the variable portion of a LINETRANSLATECAPS structure, it is imperative that older applications receive LINECARDENTRY structures in the format they previously expected, or they will not able to index properly through the array. The application passes in a *dwAPIVersion* parameter with the lineGetTranslateCaps() function, which can be used for guidance by TAPI in handling this situation. The lineGetTranslateCaps() function should use the LINECARDENTRY fields and size that correspond to the indicated TAPI version when building the LINETRANSLATECAPS structure to be returned to the application. The LINECARDENTRY structure is defined as follows in TAPI.pas:

```
PLineCardEntry = ^TLineCardEntry;
linecardentry_tag = packed record
  dwPermanentCardID,
  dwCardNameSize,
  dwCardNameOffset,
  dwCardNumberDigits,                  // TAPI v1.4
  dwSameAreaRuleSize,                  // TAPI v1.4
  dwSameAreaRuleOffset,                // TAPI v1.4
  dwLongDistanceRuleSize,              // TAPI v1.4
  dwLongDistanceRuleOffset,            // TAPI v1.4
  dwInternationalRuleSize,             // TAPI v1.4
  dwInternationalRuleOffset,           // TAPI v1.4
  dwOptions: DWORD;                    // TAPI v1.4
end;
TLineCardEntry = linecardentry_tag;
LINECARDENTRY = linecardentry_tag;
```

The fields of the LINECARDENTRY structure are described in the Table 8-18.

Table 8-18: Fields of the LINECARDENTRY structure

Field	Meaning
dwPermanentCardID	This field indicates the permanent identifier that identifies the card.
dwCardNameSize	This field indicates the size of a NULL-terminated string (size includes the NULL) that describes the card in a user-friendly manner.
dwCardNameOffset	This field indicates the offset to the beginning of a NULL-terminated string (size includes the NULL) that describes the card in a user-friendly manner.
dwCardNumberDigits	This field indicates the number of digits in the existing card number. The card number itself is not returned for security reasons (it is stored in scrambled form by TAPI). The application can use this to insert filler bytes into a text control in "password" mode to show that a number exists.
dwSameAreaRuleSize	This field indicates the total number of bytes in the dialing rule defined for calls to numbers in the same area code. The rule is a NULL-terminated string.
dwSameAreaRuleOffset	This field indicates the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure holding the dialing rule defined for calls to numbers in the same area code. The rule is a NULL-terminated string.
dwLongDistanceRuleSize	This field indicates the total number of bytes in the dialing rule defined for calls to numbers in other areas in the same country/region. The rule is a NULL-terminated string
dwLongDistanceRuleOffset	This field indicates the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure holding the dialing rule defined for calls to numbers in other areas in the same country/region. The rule is a NULL-terminated string

Field	Meaning
dwInternationalRuleSize	This field indicates the total number of bytes in the dialing rule defined for calls to numbers in other countries/regions. The rule is a NULL-terminated string.
dwInternationalRuleOffset	This field indicates the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure holding the dialing rule defined for calls to numbers in other countries/regions. The rule is a NULL-terminated string.
dwOptions	This field indicates the other settings associated with this calling card using the LINECARDOPTION_ constants.

structure LINELOCATIONENTRY TAPI.pas

The LINELOCATIONENTRY structure describes a location used to provide an address translation context. The LINETRANSLATECAPS structure can contain an array of LINELOCATIONENTRY structures. Older applications compiled with earlier TAPI versions will not know about these new fields and will use a LINELOCATIONENTRY size that is smaller than the new size. Because this is an array in the variable portion of a LINETRANSLATECAPS structure, it is imperative that older applications receive LINELOCATIONENTRY structures in the format they previously expected, or they are not able to index through the array properly. The application passes in a *dwAPIVersion* parameter with the lineGetTranslateCaps() function, which can be used for guidance by TAPI in handling this situation. The lineGetTranslateCaps() function should use the LINELOCATIONENTRY members and size that match the indicated API version when building the LINETRANSLATECAPS structure to be returned to the application. This structure does not support extensions. It is defined as follows in TAPI.pas:

```
PLineLocationEntry = ^TLineLocationEntry;
linelocationentry_tag = packed record
  dwPermanentLocationID,
  dwLocationNameSize,
  dwLocationNameOffset,
  dwCountryCode,
  dwCityCodeSize,
  dwCityCodeOffset,
  dwPreferredCardID,

  dwLocalAccessCodeSize,                  // TAPI v1.4
  dwLocalAccessCodeOffset,                // TAPI v1.4
  dwLongDistanceAccessCodeSize,           // TAPI v1.4
  dwLongDistanceAccessCodeOffset,         // TAPI v1.4
  dwTollPrefixListSize,                   // TAPI v1.4
  dwTollPrefixListOffset,                 // TAPI v1.4
  dwCountryID,                            // TAPI v1.4
  dwOptions,                              // TAPI v1.4
  dwCancelCallWaitingSize,                // TAPI v1.4
  dwCancelCallWaitingOffset: DWORD;       // TAPI v1.4
end;
TLineLocationEntry = linelocationentry_tag;
LINELOCATIONENTRY = linelocationentry_tag;
```

The fields of the LINELOCATIONENTRY structure are described in Table 8-19.

Table 8-19: Fields of the LINELOCATIONENTRY structure

Field	Meaning
dwPermanentLocationID	This field indicates the permanent identifier that identifies the location.
dwLocationNameSize	This field indicates the size of a NULL-terminated string (size includes the NULL) that describes the location in a user-friendly manner.
dwLocationNameOffset	This field indicates the offset to the beginning of a NULL-terminated string (size includes the NULL) that describes the location in a user-friendly manner.
dwCountryCode	This field indicates the country code of the location.
dwCityCodeSize	This field indicates the size of a NULL-terminated string (the size includes the NULL) specifying the city/area code associated with the location. This information, along with the country code, can be used by applications to "default" entry fields for the user when entering phone numbers to encourage the entry of proper canonical numbers.
dwCityCodeOffset	This field indicates the offset to the beginning of a NULL-terminated string specifying the city/area code associated with the location. This information, along with the country code, can be used by applications to "default" entry fields for the user when entering phone numbers to encourage the entry of proper canonical numbers.
dwPreferredCardID	This field indicates the preferred calling card when dialing from this location.
dwLocalAccessCodeSize	This field indicates the size, in bytes, of a NULL-terminated string containing the access code to be dialed before calls to addresses in the local calling area.
dwLocalAccessCodeOffset	This field indicates the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure of a NULL-terminated string containing the access code to be dialed before calls to addresses in the local calling area.
dwLongDistanceAccessCodeSize	This field indicates the size, in bytes, of a NULL-terminated string containing the access code to be dialed before calls to addresses outside the local calling area.
dwLongDistanceAccessCodeOffset	This field indicates the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure of a NULL-terminated string containing the access code to be dialed before calls to addresses outside the local calling area.
dwTollPrefixListSize	This field indicates the size, in bytes, of a NULL-terminated string containing the toll prefix list for the location. The string contains only prefixes consisting of the digits "0" through "9," separated from each other by a single "," (comma) character.
dwTollPrefixListOffset	This field indicates the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure of a NULL-terminated string containing the toll prefix list for the location. The string contains only prefixes consisting of the digits "0" through "9," separated from each other by a single "," (comma) character.

Field	Meaning
dwCountryID	This field indicates the country identifier of the country/region selected for the location. This can be used with the lineGetCountry() function to obtain additional information about the specific country/region, such as the country/region name (the dwCountryCode member cannot be used for this purpose because country codes are not unique).
dwOptions	This field indicates the options in effect for this location, with values taken from the LINELOCATIONOPTION_ constants.
dwCancelCallWaitingSize	This field indicates the size, in bytes, of a NULL-terminated string containing the dial digits and modifier characters that should be prefixed to the dialable string (after the pulse/tone character) when an application sets the LINETRANSLATEOPTION_CANCELCALLWAITING bit in the dwTranslateOptions parameter of lineTranslateAddress(). If no prefix is defined, this may be indicated by dwCancelCallWaitingSize being set to zero or by it being set to 1 and dwCancelCallWaitingOffset pointing to an empty string (single NULL byte).
dwCancelCallWaitingOffset	This field indicates the offset, in bytes, from the beginning of the LINETRANSLATECAPS structure of a NULL-terminated string containing the dial digits and modifier characters that should be prefixed to the dialable string (after the pulse/tone character) when an application sets the LINETRANSLATEOPTION_CANCELCALLWAITING bit in the dwTranslateOptions parameter of lineTranslateAddress(). If no prefix is defined, this may be indicated by dwCancelCallWaitingSize being set to zero or by it being set to 1 and dwCancelCallWaitingOffset pointing to an empty string (single NULL byte).

See Also

lineGetCountry, lineGetTranslateCaps, lineTranslateAddress, LINETRANSLATECAPS

LINELOCATIONOPTION_ Constants

The LINELOCATIONOPTION_ constants (defined in Table 8-20) define values used in the *dwOptions* member of the LINELOCATIONENTRY structure that is returned as part of the LINETRANSLATECAPS structure returned by the lineGetTranslateCaps() function.

Table 8-20: LINELOCATIONOPTION_ constants

Constant	Meaning
LINELOCATIONOPTION_ PULSEDIAL	This constant indicates if the default dialing mode at this location is pulse dialing. If this bit is set, lineTranslateAddress() will insert a "P" dial modifier at the beginning of the dialable string returned when this location is selected. If this bit is not set, lineTranslateAddress() will insert a "T" dial modifier at the beginning of the dialable string.

function lineGetDevConfig TAPI.pas

Syntax

function lineGetDevConfig(dwDeviceID: DWORD; lpDeviceConfig: PVarString; lpszDeviceClass: LPCSTR): Longint; stdcall;

Description

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.

Parameters

dwDeviceID: A DWORD holding the line device to be configured

lpDeviceConfig: A pointer (PVarString) to the memory location of type VarString where the device configuration structure is returned. If the request is successfully completed, this location is filled with the device configuration. The *dwStringFormat* field in the VarString structure will be set to STRINGFORMAT_BINARY. Before you call lineGetDevConfig(), you should set the *dwTotalSize* field of this structure to indicate the amount of memory available to TAPI for returning information.

lpszDeviceClass: A pointer (LPCSTR) to a NULL-terminated ASCII string that specifies the device class of the device whose configuration is requested. Valid device class lineGetID() strings are the same as those specified for the function.

Return Value

This function returns zero if the function is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INVALDEVICECLASS, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_STRUCTURETOOSMALL, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, LINEERR_UNINITIALIZED, and LINEERR_NODEVICE.

See Also

lineConfigDialog, lineGetID, lineSetDevConfig, VarString

Example

Listing 8-9 shows how to retrieve configuration information on a line.

Listing 8-9: Retrieving configuration information on a line

```
function TTapiInterface.GetLineConfiguration: boolean;
begin
  if FDeviceConfig=Nil then
```

```
    begin
      FDeviceConfig := AllocMem(SizeOf(VarString)+10000);
      FillChar(FDeviceConfig^, SizeOf(VarString)+10000, 0);
      FDeviceConfig.dwTotalSize := SizeOf(VarString)+10000;
      FDeviceConfig.dwStringFormat := STRINGFORMAT_BINARY;
    end;
  TAPIResult := lineGetDevConfig(DWord(fLine),
     FDeviceConfig,
     'comm/datamodem');
  result := TAPIResult=0;
  if not result then ReportError(TAPIResult)
  else
    flineConfigInfoEntered := True;
end;
```


function lineGetID   TAPI.pas

Syntax

function lineGetID(hLine: HLINE; dwAddressID: DWORD; hCall: HCALL; dwSelect: DWORD; lpDeviceID: PVarString; lpszDeviceClass: LPCSTR): Longint stdcall;

Description

This function returns a device ID for the specified device class associated with the selected line, address, or call. Given a line handle, it can be used to retrieve a line-device ID. This function is particularly useful in determining the actual line-device ID of a line that was opened using the LINEMAPPER constant as the device ID. It can also be used to obtain the device ID of a phone device or a media device for use with the appropriate API associated with that device (such as Phone, MIDI, Wave, or Audio).

Parameters

hLine: A handle (HLINE) to an open line device

dwAddressID: A DWORD holding an address on the given open line device

hCall: A handle (HCALL) to a call

dwSelect: A DWORD that specifies whether the requested device ID is associated with the line, address, or single call. The *dwSelect* parameter can only have a single flag set. This parameter uses the following LINECALL-SELECT_ constants:
LINECALLSELECT_LINE selects the specified line device (the *hLine* parameter must be a valid line handle; *hCall* and *dwAddressID* are ignored).
LINECALLSELECT_ADDRESS selects the specified address on the line (both *hLine* and *dwAddressID* must be valid; *hCall* is ignored).
LINECALLSELECT_CALL selects the specified call (*hCall* must be valid; *hLine* and *dwAddressID* are both ignored).

lpDeviceID: A pointer (PVarString) to a memory location of type VarString where the device ID is returned. If the request is succesfully completed, 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. Before you call lineGetID(), you should set the *dwTotalSize* field of this structure to indicate the amount of memory available to TAPI for returning information.

lpszDeviceClass: A pointer (LPCSTR) 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. This parameter provides a place for the provider to return different icons based on the type of service being referenced by the caller. The permitted strings are the same as those used in the SYSTEM.INI section to identify device classes.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALLINEHANDLE, LINEERR_NOMEM, LINEERR_INVALADDRESSID, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSELECT, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NODEVICE, and LINEERR_UNINITIALIZED.

See Also

VarString

Example

Listing 8-10 shows how to retrieve a line ID.

Listing 8-10: Retrieving a line ID

```
function TTapiInterface.GetLineID: boolean;
var
  TempStr: string;
begin
  TAPIResult := lineGetID(fLine, 0, 0,
    LINECALLSELECT_LINE, PVarString(FDeviceID), 'tapi/line');
  result := TAPIResult=0;
  if NOT Result then
    ReportError(TAPIResult);
end;
```

function lineGetLineDevStatus TAPI.pas

Syntax

function lineGetLineDevStatus(hLine: HLINE; lpLineDevStatus: PLineDevStatus): Longint; stdcall;

Description

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

Parameters

hLine: A handle (HLINE) to the open line device to be queried

lpLineDevStatus: A pointer (PLineDevStatus) to a variably sized data structure of type LINEDEVSTATUS. If the request is successfully completed, this structure is filled with the line's device status. Before you call lineGetLineDevStatus(), you should set the *dwTotalSize* field of the LINEDEVSTATUS structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONFAILED, and LINEERR_OPERATIONUNAVAIL.

See Also

LINEDEVSTATUS, lineGetAddressStatusExample

Example

Listing 8-11 shows how to get a line device's status.

Listing 8-11: Getting a line device's status

```
function TTapiInterface.GetLineDevStatus: boolean;
begin
  if fLineDevStatus=Nil then
    fLineDevStatus := AllocMem(SizeOf(TLineDevStatus)+1000);
  fLineDevStatus.dwTotalSize := SizeOf(TLineDevStatus)+1000;
  TapiResult := lineGetLineDevStatus(fLine, fLineDevStatus);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

structure LINEDEVSTATUS TAPI.pas

The LINEDEVSTATUS structure describes the current status of a line device. The lineGetLineDevStatus() and the TSPI_lineGetLineDevStatus() functions return the LINEDEVSTATUS structure. Device-specific extensions should use the DevSpecific (*dwDevSpecificSize* and *dwDevSpecificOffset*) variably sized area of this data structure. The members *dwAvailableMediaModes* through *dwAppInfoOffset* are available only to applications that open the line device with an API version of 2.0 or later. It is defined as follows in TAPI.pas:

```
  PLineDevStatus = ^TLineDevStatus;
  linedevstatus_tag = packed record
    dwTotalSize,
    dwNeededSize,
    dwUsedSize,
    dwNumOpens,
    dwOpenMediaModes,
    dwNumActiveCalls,
    dwNumOnHoldCalls,
    dwNumOnHoldPendCalls,
    dwLineFeatures,
    dwNumCallCompletions,
    dwRingMode,
    dwSignalLevel,
    dwBatteryLevel,
    dwRoamMode,
    dwDevStatusFlags,
    dwTerminalModesSize,
    dwTerminalModesOffset,
    dwDevSpecificSize,
    dwDevSpecificOffset: DWORD;
{$IFDEF TAPI20}
    dwAvailableMediaModes,                        // TAPI v2.0
    dwAppInfoSize,                                // TAPI v2.0
    dwAppInfoOffset: DWORD;                       // TAPI v2.0
{$ENDIF}
  end;
  TLineDevStatus = linedevstatus_tag;
  LINEDEVSTATUS = linedevstatus_tag;
```

The fields of the LINEDEVSTATUS structure are defined in Table 8-21.

Table 8-21: Fields of the LINEDEVSTATUS structure

Field	Meaning
dwTotalSize	This field indicates the total size, in bytes, allocated to this data structure.
dwNeededSize	This field indicates the size, in bytes, for this data structure that is needed to hold all the returned information.
dwUsedSize	This field indicates the size, in bytes, of the portion of this data structure that contains useful information.
dwNumOpens	This field indicates the number of active opens on the line device.
dwOpenMediaModes	This field is a bit array that indicates for which media types the line device is currently open.
dwNumActiveCalls	This field indicates the number of calls on the line in call states other than idle, onHold, onHoldPendingTransfer, and onHoldPendingConference.

Field	Meaning
dwNumOnHoldCalls	This field indicates the number of calls on the line in the onHold state.
dwNumOnHoldPendCalls	This field indicates the number of calls on the line in the onHoldPendingTransfer or onHoldPendingConference state.
dwLineFeatures	This field specifies the line-related TAPI functions that are currently available on this line. This member uses one or more of the LINEFEATURE_ constants. See Table 8-15.
dwNumCallCompletions	This field indicates the number of outstanding call completion requests on the line.
dwRingMode	This field indicates the current ring mode on the line device.
dwSignalLevel	This field indicates the current signal level of the connection on the line. This is a value in the range of $00000000 (weakest signal) to $0000FFFF (strongest signal).
dwBatteryLevel	This field indicates the current battery level of the line device hardware. This is a value in the range of $00000000 (battery empty) to $0000FFFF (battery full).
dwRoamMode	This field indicates the current roam mode of the line device. This member uses one of the LINEROAMMODE_ constants.
dwDevStatusFlags	This field specifies the status flags indicate information, such as whether the device is locked. It consists of one or more members of LINEDEVSTATUSFLAGS_ constants.
dwTerminalModesSize	This field specifies the size in bytes of the data structure of the variably sized device field containing an array with DWORD-sized entries that use the LINETERMMODE_ constants. This array is indexed by terminal IDs, in the range from zero to one less than dwNumTerminals. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the lineSetTerminal() function for this address. The values are: LINETERMMODE_LAMPS indicates that these are lamp events sent from the line to the terminal. LINETERMMODE_BUTTONS indicates that these are button-press events sent from the terminal to the line. LINETERMMODE_DISPLAY indicates that this is display information sent from the line to the terminal. LINETERMMODE_RINGER indicates that this is ringer-control information sent from the switch to the terminal. LINETERMMODE_HOOKSWITCH indicates that these are hookswitch events sent between the terminal and the line. LINETERMMODE_MEDIATOLINE indicates that this is the unidirectional media stream from the terminal to the line associated with a call on the line (use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently).
dwTerminalModesOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized device field containing an array with DWORD-sized entries that use the LINETERMMODE_ constants. This array is indexed by terminal IDs, in the range from zero to dwNumTerminals minus one. Each entry in the array specifies the current terminal modes for the corresponding terminal set using the lineSetTerminal() function for this line. Values are the same as those listed under dwTerminalModesSize.
dwDevSpecificSize	This field indicates the size, in bytes, of the variably sized device-specific field.
dwDevSpecificOffset	This field indicates the offset, in bytes, from the beginning of the variably sized device-specific field.

Field	Meaning
dwAvailableMediaModes	This field indicates the media types that can be invoked on new calls created on this line device, when the dwLineFeatures member indicates that new calls are possible. If this member is zero, it indicates that the service provider either does not know or cannot indicate which media types are available, in which case any or all of the media types indicated in the dwMediaModes member in LINEDEVCAPS may be available.
dwAppInfoSize	This field indicates the length, in bytes, of an array of LINEAPPINFO structures. The dwNumOpens member indicates the number of elements in the array. Each element in the array identifies an application that has the line open.
dwAppInfoOffset	This field indicates the offset from the beginning of LINEDEVSTATUS. The dwNumOpens member indicates the number of elements in the array. Each element in the array identifies an application that has the line open.

See Also

LINEAPPINFO, LINEDEVCAPS, lineGetLineDevStatus, lineSetTerminal, TSPI_lineGetLineDevStatus

structure LINEAPPINFO TAPI.pas

The LINEAPPINFO structure contains information about the application that is currently running. The LINEDEVSTATUS structure can contain an array of LINEAPPINFO structures. The structure is defined in TAPI.pas as follows:

```
PLineAppInfo = ^TLineAppInfo;
lineappinfo_tag = packed record
  dwMachineNameSize,                                  // TAPI v2.0
  dwMachineNameOffset,                                // TAPI v2.0
  dwUserNameSize,                                     // TAPI v2.0
  dwUserNameOffset,                                   // TAPI v2.0
  dwModuleFilenameSize,                               // TAPI v2.0
  dwModuleFilenameOffset,                             // TAPI v2.0
  dwFriendlyNameSize,                                 // TAPI v2.0
  dwFriendlyNameOffset,                               // TAPI v2.0
  dwMediaModes,                                       // TAPI v2.0
  dwAddressID: DWORD;                                 // TAPI v2.0
end;
TLineAppInfo = lineappinfo_tag;
LINEAPPINFO = lineappinfo_tag;
```

The fields of the LINEAPPINFO structure are described in Table 8-22.

Table 8-22: Fields of the LINEAPPINFO structure

Field	Member
dwMachineNameSize	Size, in bytes, of a string specifying the name of the computer on which the application is executing.
dwMachineNameOffset	Offset from the beginning of LINEDEVSTATUS of a string specifying the name of the computer on which the application is executing.
dwUserNameSize	Size, in bytes, of a string specifying the user name under whose account the application is running.
dwUserNameOffset	Offset from the beginning of LINEDEVSTATUS of a string specifying the user name under whose account the application is running.

Field	Member
dwModuleFilenameSize	Size, in bytes, of a string specifying the module filename of the application. This string can be used in a call to lineHandoff() to perform a directed handoff to the application.
dwModuleFilenameOffset	Offset from the beginning of LINEDEVSTATUS of a string specifying the module filename of the application. This string can be used in a call to lineHandoff() to perform a directed handoff to the application.
dwFriendlyNameSize	Size, in bytes, of the string provided by the application to lineInitialize() or lineInitializeEx(), which should be used in any display of applications to the user.
dwFriendlyNameOffset	Offset from the beginning of LINEDEVSTATUS of the string provided by the application to lineInitialize() or lineInitializeEx(), which should be used in any display of applications to the user.
dwMediaModes	The media types for which the application has requested ownership of new calls; zero if when it opened the line, dwPrivileges did not include LINECALLPRIVILEGE_OWNER.
dwAddressID	If the line handle was opened using LINEOPENOPTION_SINGLEADDRESS, it contains the address identifier specified; set to $FFFFFFFF if the single address option was not used. An address identifier is permanently associated with an address; the identifier remains constant across operating system upgrades.

See Also

LINEDEVSTATUS, lineGetLineDevStatus, lineHandoff, lineInitialize, lineInitializeEx, TSPI_lineGetLineDevStatus

function lineGetTranslateCaps TAPI.pas

Syntax

```
function lineGetTranslateCaps(hLineApp: HLINEAPP; dwAPIVersion: DWORD;
lpTranslateCaps: PLineTranslateCaps): Longint; stdcall;
```

Description

This function returns address translation capabilities.

Parameters

hLineApp: The application handle (HLINEAPP) returned by lineInitializeEx(). If an application has not yet called the lineInitializeEx() function, it can set the *hLineApp* parameter to NULL.

dwAPIVersion: A DWORD that indicates the highest version of TAPI supported by the application (not necessarily the value negotiated by lineNegotiateAPIVersion() on some particular line device).

lpTranslateCaps: A pointer (PLineTranslateCaps) to a location to which a LINETRANSLATECAPS structure will be loaded. Before you call lineGetTranslateCaps(), you should set the *dwTotalSize* field of the structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_NOMEM, LINEERR_INIFILECORRUPT, LINEERR_OPERATIONFAILED, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, and LINEERR_NODRIVER.

See Also

lineInitializeEx, lineNegotiateAPIVersion, LINETRANSLATECAPS

Example

Listing 8-12 shows how to retrieve the address translation capabilities for a line.

Listing 8-12: Retrieve a line's address translation capabilities

```
function TTapiInterface.GetTranslateCaps: boolean;
var
  I: Integer;
begin
  TapiResult := lineGetTranslateCaps(fLineApp, FHiVersion, fLineTranslateCaps);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult)
  else
    begin
      fNumLocations := fLineTranslateCaps^.dwNumLocations ;
      for I := 0 to fNumLocations-1 do    // Iterate
      begin
        fPLineLocationEntry := Pointer(fLineTranslateCaps);
        Inc(fPLineLocationEntry, fLineTranslateCaps.dwLocationListOffset
          + ((fLineTranslateCaps.dwLocationListSize * (I+1))-
            fLineTranslateCaps.dwLocationListSize));
        fPLineLocationEntry := AllocMem(SizeOf(LineLocationEntry));
        with fPLineLocationEntry^ do
          LocationArray[I] := DWord(dwPermanentLocationID);
        FreeMem(fPLineLocationEntry, SizeOf(fPLineLocationEntry^));
        fPLineLocationEntry := Nil;
      end;    // for loop
    end;
end;
```

function lineInitialize   TAPI.pas

Syntax

function lineInitialize(lphLineApp: PHLineApp; hInstance: HINST; lpfnCallback: TLineCallback; lpszAppName: LPCSTR; var dwNumDevs: DWORD): Longint; stdcall;

Description

This function is obsolete. It continues to be exported by TAPI.DLL and TAPI32.DLL for backward compatibility with applications using API versions 1.3 and 1.4. Applications that use TAPI version 2.0 or greater must use lineInitializeEx() instead. 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.

Parameters

lphLineApp: A pointer (PHLineApp) to a location that is filled with the application's usage handle for TAPI

hInstance: The instance handle (HINST) of the client application or DLL

lpfnCallback: The address of a callback function (TLineCallback) that is invoked to determine status and events on the line device, addresses, or calls. For more information, see lineCallbackFunc() (in the TAPI Help file) and TLineCallback.

lpszAppName: A pointer (LPCSTR) to a NULL-terminated ASCII string that contains only displayable ASCII characters. If this parameter is not NULL, it contains an application-supplied name for the application. This name is provided in the LINECALLINFO structure to indicate, in a user-friendly way, which application originated, or originally accepted or answered the call. This information can be useful for call logging purposes. If *lpszAppName* is NULL, the application's filename is used instead.

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

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALAPPNAME, LINEERR_OPERATIONFAILED, LINEERR_INIFILECORRUPT, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_REINIT, LINEERR_NODRIVER, LINEERR_NODEVICE, LINEERR_NOMEM, and LINEERR_NOMULTIPLEINSTANCE.

See Also

lineInitializeEx

Example

This function is obsolete and no listing is provided. See the example for the next function (Listing 8-13) for an idea on how the lineInitialize() function could be used.

function lineInitializeEx TAPI.pas

Syntax

```
function lineInitializeEx(lphLineApp: PHLineApp; hInstance: HINST; lpfnCallback:
TLineCallback; lpszAppName: LPCSTR; var dwNumDevs, dwAPIVersion:
DWORD;   var LineInitializeExParams: TLineInitializeExParams): Longint; stdcall;
```

Description

This function initializes the application's use of TAPI for subsequent use of the line abstraction. It registers the application's specified notification mechanism and returns the number of line devices available to the application. A line device is any device that provides an implementation for the line-prefixed functions in the telephony API.

Parameters

lphLineApp: A pointer (PHLineApp) to a location that is filled with the application's usage handle for TAPI

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

lpfnCallback: The address (TLineCallback) of a callback function that is invoked to determine status and events on the line device, addresses, or calls when the application is using the "hidden window" method of event notification (for more information see lineCallbackFunc() in the TAPI Help file and TLineCallback). This parameter is ignored and should be set to NULL when the application chooses to use the "event handle" or "completion port" event notification mechanisms.

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

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

dwAPIVersion: A pointer to a DWORD-sized location. The application must initialize this DWORD before calling this function to the highest API version it is designed to support (for example, the same value it would pass into the *dwAPIHighVersion* parameter of lineNegotiateAPIVersion()). Artificially high values must not be used; the value must be accurately set (for this release, to $00020000). TAPI will translate any newer messages or structures into values or formats supported by the application's version. Upon successful completion of this request, this location is filled with the highest API version supported by TAPI (for this release, $00020000), thereby allowing the application to detect and adapt to having been installed on a system with an older version of TAPI.

var LineInitializeExParams: A pointer (TLineInitializeExParams) to a structure of type LINEINITIALIZEEXPARAMS containing additional parameters used to establish the association between the application and TAPI (specifically, the application's selected event notification mechanism and associated parameters)

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALAPPNAME, LINEERR_OPERATIONFAILED, LINEERR_INIFILECORRUPT, LINEERR_INVALPOINTER, LINEERR_REINIT, LINEERR_NOMEM, and LINEERR_INVALPARAM.

See Also

Line Callback (Chapter 9), LINECALLINFO, lineGetAddressCaps, lineGetDevCaps, lineGetMessage, lineInitialize, LINEINITIALIZEEXPARAMS, LINEMESSAGE, lineNegotiateAPIVersion, lineShutdown

Example

Listing 8-13 shows how to prepare for and call the lineInitializeEx() function. Note that our initialization routine provides options for opening TAPI with version 2.2 or 3.0 and using either the Hidden Window or Event Handle method.

Listing 8-13: Preparing for and calling the lineInitializeEx() function

```
function TTapiInterface.TapiLineInitializeUsingWindow: boolean;
begin
  FPLineInitializeExParams.dwTotalSize := SizeOf(TLineInitializeExParams);
  FPLineInitializeExParams.dwOptions := LINEINITIALIZEEXOPTION_USEHIDDENWINDOW;
  TAPIResult := LineInitializeEx(pHLineApp(@fLineApp), 0, ALineCallback, Nil,
  Cardinal(fNumLineDevs), FHiVersion, FPLineInitializeExParams);
  result := TAPIResult=0;
  if NOT result then ReportError(TAPIResult);
end;

function TTapiInterface.TapiLineInitializeUsingEvent: boolean;
begin
  FPLineInitializeExParams.dwTotalSize := SizeOf(TLineInitializeExParams);
  FPLineInitializeExParams.dwOptions := LINEINITIALIZEEXOPTION_USEEVENT;
  TAPIResult := LineInitializeEx(pHLineAPP(@fLineApp), 0, @ALineCallback, Nil,
  Cardinal(fNumLineDevs), FAPIVersion, FPLineInitializeExParams);
```

```
  result := TAPIResult=0;
  if NOT result then ReportError(TAPIResult);
end;

function TTapiInterface.TapiLineInitialize(ATAPIVersion : TTapiVersion;
            ATAPIInitMethod : TTAPIInitMethod): boolean;
var
  i : integer;
begin
  Result := false;
  case ATAPIVersion of    //
    tvWin95 : InitToWin9X;
    tvWin2000 : InitToWin2000;
  end;    // case
  case ATAPIInitMethod of    //
    timHiddenWindow : if NOT TapiLineInitializeUsingWindow then
      begin
        ShowMessage('Could Not Initialize TAPI');
        exit;
      end;
    timEventHandle : if NOT TapiLineInitializeUsingEvent then
      begin
        ShowMessage('Could Not Initialize TAPI');
        exit;
      end;
    timCompletionPort : ShowMessage('This method is not supported');
  end;    // case
  TAPI_Initialized := True;
  OnSendTapiMessage('Devices available: ' + IntToStr(fNumLineDevs));
  if NOT NegotiateVersionOfTAPI then
    begin
      ShowMessage('Could Not Negotiate a TAPI Version');
      exit;
    end;
  for I := 0 to (fNumLineDevs-1) do    // Iterate
  begin
    TAPIResult := lineNegotiateExtVersion(fLineApp, I, DWord(FAPIVersion),
      DWord(FLoVersion), DWord(FHiVersion), FExtVersion);
    if TAPIResult <> 0 then
      begin
        ReportError(TAPIResult);
        FExtVersion := 0;
      end;
  end;    // for
  result := True;
  GetDeviceCapsSize(FDeviceCapsAllocSize);
  GetAddressCapsSize(FAddressCapsAllocSize);
end;
```

function lineNegotiateAPIVersion   TAPI.pas

Syntax

function lineNegotiateAPIVersion(hLineApp: HLINEAPP; dwDeviceID, dwAPILowVersion, dwAPIHighVersion: DWORD; var dwAPIVersion: DWORD; var lpExtensionID: TLineExtensionID): Longint; stdcall;

Description

This function allows an application to negotiate an API version to use.

Parameters

hLineApp: The handle (HLINEAPP) to the application's registration with TAPI

dwDeviceID: A DWORD indicating the line device to be queried

dwAPILowVersion: A DWORD indicating 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: A DWORD indicating 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.

var dwAPIVersion: A pointer to a DWORD-sized location that contains the API version number that was negotiated. If negotiation is successful, this number will be in a range between *dwAPILowVersion* and *dwAPIHighVersion*.

var lpExtensionID: A pointer (TLineExtensionID) 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 zeroes if the line provides no extensions. An application can ignore the returned parameter if it does not use extensions.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_OPERATIONFAILED, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_OPERATIONUNAVAIL, and LINEERR_NODEVICE.

See Also

LINEEXTENSIONID, lineInitializeEx, lineNegotiateExtVersion

Example

Listing 8-14 shows how to call the lineNegotiateAPIVersion() function.

Listing 8-14: Calling the lineNegotiateAPIVersion() function

```
function TTapiInterface.NegotiateVersionOfTAPI: boolean;
begin
  TAPIResult := LineNegotiateAPIVersion(fLineApp, 0, DWord(FLoVersion),
    DWord(FHiVersion), DWord(FAPIVersion), FLineExtensionID);
    result := (TAPIResult = 0);
    if result then OnSendTapiMessage('Negotiation of TAPI version successful')
```

```
    else
      ReportError(TAPIResult);
end;
```

function lineNegotiateExtVersion TAPI.pas

Syntax

function lineNegotiateExtVersion(hLineApp: HLINEAPP; dwDeviceID, dwAPIVersion, dwExtLowVersion, dwExtHighVersion: DWORD; var dwExtVersion: DWORD): Longint; stdcall;

Description

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.

Parameters

hLineApp: The handle (HLINEAPP) to the application's registration with TAPI

dwDeviceID: A DWORD indicating the line device to be queried

dwAPIVersion: A DWORD indicating the API version number that was negotiated for the specified line device using lineNegotiateAPIVersion()

dwExtLowVersion: A DWORD indicating 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: A DWORD indicating 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.

var dwExtVersion: A pointer to a DWORD-sized location that contains the extension version number that was negotiated. If negotiation is successful, this number will be in the range between *dwExtLowVersion* and *dwExtHighVersion*.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_NOMEM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_NODRIVER, LINEERR_INCOMPATIBLEEXTVERSION, LINEERR_OPERATIONFAILED, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NODEVICE, and LINEERR_OPERATIONUNAVAIL.

See Also

lineInitializeEx, lineNegotiateAPIVersion

Example

See Listing 8-13.

function lineOpen TAPI.pas

Syntax

```
function lineOpen(hLineApp: HLINEAPP; dwDeviceID: DWORD; lphLine: PHLine;
dwAPIVersion, dwExtVersion, dwCallbackInstance, dwPrivileges, dwMediaModes:
DWORD;  lpCallParams: PLineCallParams): Longint; stdcall;
```

Description

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. To stop handling requests on the line, the application simply calls the lineClose() function.

Parameters

hLineApp: A handle (HLINEAPP) to the application's registration with TAPI

dwDeviceID: A DWORD that identifies the line device to be opened. It can either be a valid device ID or the value LINEMAPPER, indicating that this value is used to open a line device in the system that supports the properties specified in *lpCallParams*. The application can use lineGetID() to determine the ID of the line device that was opened.

lphLine: A pointer (PHLine) 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: A DWORD indicating the API version number under which the application and Telephony API have agreed to operate. This number is obtained with lineNegotiateAPIVersion().

dwExtVersion: A DWORD indicating the extension version number under which the application and the service provider agree to operate. This number is obtained with lineNegotiateExtVersion(), and is zero if the application does not use any extensions.

dwCallbackInstance: A DWORD containing user-instance data passed back to the application with each message associated with this line or addresses or calls on this line. This parameter is not interpreted by the Telephony API.

dwPrivileges: A DWORD indicating the privilege the application wants for the calls it is notified for. This parameter can be a combination of the LINECALLPRIVILEGE_ constants shown in Table 8-23. For applications using API version 2.0 or greater, values for this parameter can also be combined with the LINEOPENOPTION_ constants. Other flag combinations return the LINEERR_INVALPRIVSELECT error.

dwMediaModes: A DWORD indicating the media mode or modes of interest to the application. This parameter is used to register the application as a potential target for inbound call and call handoff for the specified media mode. This parameter is meaningful only if the bit LINECALLPRIVILEGE_OWNER in *dwPrivileges* is set (and ignored if it is not). This parameter uses the LINEMEDIAMODE_ constants shown in Table 8-24.

lpCallParams: A pointer (PLineCallParams) to a structure of type LINECALLPARAMS. This pointer is only used if LINEMAPPER is used; otherwise, *lpCallParams* is ignored. It describes the call parameter that the line device should be able to provide.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_ALLOCATED, LINEERR_LINEMAPPERFAILED, LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_NOMEM, LINEERR_INCOMPATIBLEEXTVERSION, LINEERR_OPERATIONFAILED, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALMEDIAMODE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_INVALPRIVSELECT, LINEERR_REINIT, LINEERR_NODEVICE, and LINEERR_OPERATIONUNAVAIL.

See Also

LINE_CALLSTATE, LINE_MONITORMEDIA, LINE_PROXYREQUEST, LINECALLPARAMS, lineClose, lineInitializeEx, lineMakeCall, lineNegotiateAPIVersion, lineNegotiateExtVersion, and lineShutdown. In the TAPI Help file, see lineForward, lineGetConfRelatedCalls, lineGetNewCalls, lineMonitorMedia, linePickup, lineProxyMessage, lineProxyResponse, lineSetupConference, lineUnpark

Example

Listing 8-15 shows how to call the lineOpen() function.

Listing 8-15: Calling the lineOpen() function

```
function TTapiInterface.OpenLine(var OpenResult: DWord;
           AcceptCalls : boolean): boolean;
begin
  if NOT TAPI_Initialized then
    if NOT TapiLineInitialize(TAPIVersion, TAPIInitMethod) then
      begin
        ShowMessage('Could not Initialize TAPI');
```

```
        result := false;
        exit
      end;
  OpenResult := 0;
  if AutoSelectLine then // automatically select the device
    // use LINEMAPPER to get an appropriate line
    if AcceptCalls then
      // open a line (outgoing and incoming calls) and get the line handle
      OpenResult := LineOpen(fLineApp, LINEMAPPER, @fLine,
        FVersion, 0, 0, LINECALLPRIVILEGE_OWNER, fMediaMode,
        @fLineCallParams)
    else
      OpenResult := LineOpen(fLineApp, LINEMAPPER, @fLine,
        FAPIVersion, 0, 0, LINECALLPRIVILEGE_NONE, fMediaMode, nil)
  else
  if AcceptCalls then
      // open a line (outgoing and incoming calls) and get the line handle
     OpenResult := LineOpen(fLineApp, FDev, @fLine,
       FAPIVersion, 0, 0, LINECALLPRIVILEGE_OWNER,
       fMediaMode,
       @fLineCallParams)
  else
    OpenResult := LineOpen(fLineApp, FDev, @fLine,
      FAPIVersion, 0, 0, LINECALLPRIVILEGE_NONE, fMediaMode, nil);
     // open a line (outgoing calls only) and get the line handle
  result := OpenResult=0;
  if  Not Result then ReportError(OpenResult)
  else
    fLineIsOpen := True;
end;
```

Table 8-23: LINECALLPRIVILEGE_ constants used in the lineOpen() function's dwPrivileges parameter

Constant	Meaning
LINECALLPRIVILEGE_NONE	This constant indicates that the application wants to make only outbound calls.
LINECALLPRIVILEGE_MONITOR	This constant indicates that the application only wants to monitor inbound and outbound calls.
LINECALLPRIVILEGE_OWNER	This constant indicates that the application wants to own inbound calls of the types specified in dwMediaModes.
LINECALLPRIVILEGE_MONITOR + LINECALLPRIVILEGE_OWNER	This constant indicates that the application wants to own inbound calls of the types specified in dwMediaModes, but if it cannot be an owner of a call, it wants to be a monitor.
LINEOPENOPTION_ SINGLEADDRESS	This constant indicates that the application is interested only in new calls that appear on the address specified by the dwAddressID field in the LINECALLPARAMS structure pointed to by the lpCallParams parameter (which must be specified). If LINEOPENOPTION_SINGLEADDRESS is specified but either lpCallParams is invalid or the included dwAddressID does not exist on the line, the open fails with LINERR_INVALADDRESSID. In addition to setting the dwAddressID member of the LINECALLPARAMS structure to the desired address, the application must also set dwAddress-Mode in LINECALLPARAMS to LINEADDRESSMODE_ ADDRESSID.

Constant	Meaning
LINEOPENOPTION_ SINGLEADDRESS (cont.)	The LINEOPENOPTION_SINGLEADDRESS option affects only TAPI's assignment of initial call ownership of calls created by the service provider using a LINE_NEWCALL message. An application that opens the line with LINECALLPRIVILEGE_MONITOR will continue to receive monitoring handles to all calls created on the line. Furthermore, the application is not restricted in any way from making calls or performing other operations that affect other addresses on the line opened.
LINEOPENOPTION_PROXY	This constant indicates that the application is willing to handle certain types of requests from other applications that have the line open, as an adjunct to the service provider. Requests will be delivered to the application using LINE_PROXYREQUEST messages; the application responds to them by calling lineProxyResponse() and can also generate TAPI messages to other applications having the line open by calling lineProxyMessage(). When this option is specified, the application must also specify which specific proxy requests it is prepared to handle. It does so by passing, in the lpCallParams parameter, a pointer to a LINECALLPARAMS structure in which the dwDevSpecificSize and dwDevSpecificOffset members have been set to delimit an array of DWORDs. Each element of this array shall contain one of the LINEPROXYREQUEST_ constants. For example, a proxy handler application that supports all five of the Agent-related functions would pass in an array of five DWORDs (dwDevSpecificSize would be 20 decimal) containing the five defined LINEPROXYREQUEST_ values.
LINEOPENOPTION_PROXY (cont.)	The proxy request handler application can run on any machine that has authorization to control the line device. However, requests will always be routed through the server on which the service provider is executing that actually controls the line device. Thus, it is most efficient if the application handling proxy requests (such as ACD agent control) executes directly on the server along with the service provider. Subsequent attempts by the same application or other applications to open the line device and register to handle the same proxy requests as an application that is already registered fail with LINEERR_NOTREGISTERED.

Table 8-24: LINEMEDIAMODE_ constants

Constant	Meaning
LINEMEDIAMODE_UNKNOWN	This constant indicates that the target application is the one that handles calls of unknown media mode (unclassified calls).
LINEMEDIAMODE_INTERACTIVEVOICE	This constant indicates that the target application is the one that handles calls with the interactive voice media mode (live conversations).
LINEMEDIAMODE_AUTOMATEDVOICE	This constant indicates that voice energy is present on the call, and the voice is locally handled by an automated application.
LINEMEDIAMODE_DATAMODEM	This constant indicates that the target application is the one that handles calls with the data modem media mode.
LINEMEDIAMODE_G3FAX	This constant indicates that the target application is the one that handles calls with the group 3 fax media mode.
LINEMEDIAMODE_TDD	This constant indicates that the target application is the one that handles calls with the TDD (Telephony Devices for the Deaf) media mode.

Constant	Meaning
LINEMEDIAMODE_G4FAX	This constant indicates that the target application is the one that handles calls with the group 4 fax media mode.
LINEMEDIAMODE_DIGITALDATA	This constant indicates that the target application is the one that handles calls that are digital data calls.
LINEMEDIAMODE_TELETEX	This constant indicates that the target application is the one that handles calls with the teletex media mode.
LINEMEDIAMODE_VIDEOTEX	This constant indicates that the target application is the one that handles calls with the videotex media mode.
LINEMEDIAMODE_TELEX	This constant indicates that the target application is the one that handles calls with the telex media mode.
LINEMEDIAMODE_MIXED	This constant indicates that the target application is the one that handles calls with the ISDN mixed media mode.
LINEMEDIAMODE_ADSI	This constant indicates that the target application is the one that handles calls with the ADSI (Analog Display Services Interface) media mode.
LINEMEDIAMODE_VOICEVIEW	This constant indicates that the media mode of the call is VoiceView.

function lineSetDevConfig TAPI.pas

Syntax

function lineSetDevConfig(dwDeviceID: DWORD; lpDeviceConfig: Pointer; dwSize: DWORD; lpszDeviceClass: LPCSTR): Longint; stdcall;

Description

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.

Parameters

dwDeviceID: A DWORD indicating the line device to be configured

lpDeviceConfig: A pointer to the opaque configuration data structure that was returned by lineGetDevConfig() in the variable portion of the VarString structure

dwSize: A DWORD indicating the number of bytes in the structure pointed to by *lpDeviceConfig*. This value will have been returned in the *dwStringSize* field in the VarString structure returned by lineGetDevConfig().

lpszDeviceClass: A pointer (LPCSTR) to a NULL-terminated ASCII string that specifies the device class of the device whose configuration is to be set. Valid device class strings are the same as those specified for the lineGetID() function.

Return Value

This function returns zero if the function is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INVALDEVICECLASS, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALPOINTER, LINEERR_OPERATIONFAILED, LINEERR_INVALPARAM, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINESTATE, LINEERR_UNINITIALIZED, LINEERR_NOMEM, and LINEERR_NODEVICE.

See Also

lineConfigDialog, lineGetDevConfig, lineGetID, VarString

Example

Listing 8-16 shows how to use the lineSetDevConfig() function.

Listing 8-16: Using the lineSetDevConfig() function

```
function TTapiInterface.SetLineConfiguration: boolean;
begin
  TAPIResult := lineSetDevConfig(DWord(0), @FDeviceConfigOut.data,
     FConfigSize, 'comm/datamodem');
     //can substitute 'tapi/line' for last parameter
  result := TAPIResult=0;
  if not result then ReportError(TAPIResult)
    else FlineConfigInfoEntered := True;
end;
```

function lineShutdown TAPI.pas

Syntax

function lineShutdown(hLineApp: HLINEAPP): Longint; stdcall;

Description

This function shuts down the application's usage of the line abstraction of API.

Parameters

hLineApp: The application's usage handle (HLINEAPP) for the line API

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, and LINEERR_UNINITIALIZED.

See Also

lineClose

Example

Listing 8-17 shows how to shut down TAPI.

Listing 8-17: Shutting down TAPI

```
function TTapiInterface.ShutdownLine: boolean;
begin
  TAPIResult := LineShutdown(fLineApp);
  result := (TAPIResult = 0);
  If result then
    begin
      OnSendTapiMessage('success!');
      result := True;
      TAPI_Initialized := False;
      TapiInterface.SetLineIsOpen(False);
      Exit;
    end
  else
    ReportError(TAPIResult);
end;
```


function lineGetCountry TAPI.pas

Syntax

function lineGetCountry(dwCountryID, dwAPIVersion: DWORD; lpLineCountryList: PLineCountryList): Longint; stdcall; // TAPI v1.4

Description

This function fetches the stored dialing rules and other information related to a specified country, the first country in the country list, or all countries.

Parameters

dwCountryID: A DWORD holding the country ID (not the country code) of the country for which information is to be obtained. If the value 1 is specified, information on the first country in the country list is obtained. If the value 0 is specified, information on all countries is obtained (which may require a great deal of memory—20 Kbytes or more).

dwAPIVersion: A DWORD indicating the highest version of TAPI supported by the application (not necessarily the value negotiated by lineNegotiateAPI-Version() on some particular line device).

lpLineCountryList: A pointer (PLineCountryList) to a location to which a LINECOUNTRYLIST structure will be loaded. Before you call lineGetCountry(), you should set the *dwTotalSize* field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_NOMEM, LINEERR_INIFILECORRUPT, LINEERR_OPERATIONFAILED, LINEERR_INVALCOUNTRYCODE, LINEERR_STRUCTURETOOSMALL, and LINEERR_INVALPOINTER.

See Also

LINECOUNTRYLIST, lineNegotiateAPIVersion

Example

Listing 8-18 shows how to call this function and retrieve the number of countries in the current country list.

Listing 8-18: Retrieving the number of countries in the current country list

```
function TTapiInterface.GetCountryInfo(ACountry : DWord): boolean;
begin
  TapiResult :=   lineGetCountry(ACountry, fHiVersion,
    fPLineCountryList);
  result := TapiResult=0;
  if result then
    NumCountries := fPLineCountryList^.dwNumCountries
  else
    begin
      ReportError(TAPIResult);
      NumCountries := 0;
    end;
end;
```

structure LINECOUNTRYLIST   TAPI.pas

The LINECOUNTRYLIST structure describes a list of countries. A structure of this type is returned by the function lineGetCountry(). This structure cannot be extended. The structure is defined as follows in TAPI.pas:

```
PLineCountryList = ^TLineCountryList;
linecountrylist_tag = packed record
  dwTotalSize,                           // TAPI v1.4
  dwNeededSize,                          // TAPI v1.4
  dwUsedSize,                            // TAPI v1.4
  dwNumCountries,                        // TAPI v1.4
  dwCountryListSize,                     // TAPI v1.4
  dwCountryListOffset: DWORD;            // TAPI v1.4
end;
TLineCountryList = linecountrylist_tag;
LINECOUNTRYLIST = linecountrylist_tag;
```

The fields of the LINECOUNTRYLIST structure are described in Table 8-25.

Table 8-25: Fields of the LINECOUNTRYLIST structure

Field	Meaning
dwTotalSize	This field specifies the total size in bytes allocated to this data structure.
dwNeededSize	This field specifies the size in bytes for this data structure that is needed to hold all the returned information.
dwUsedSize	This field specifies the size in bytes of the portion of this data structure that contains useful information.
dwNumCountries	This field specifies the number of LINECOUNTRYENTRY structures present in the array denominated by dwCountryListSize and dwCountryListOffset.
dwCountryListSize	This field specifies the size in bytes of an array of LINECOUNTRYENTRY elements, which provide the information on each country.
dwCountryListOffset	This field specifies the offset in bytes from the beginning of this data structure of an array of LINECOUNTRYENTRY elements, which provide the information on each country.

structure LINECOUNTRYENTRY TAPI.pas

The LINECOUNTRYENTRY structure provides the information for a single country entry. An array of 1 or more of these structures is returned as part of the LINECOUNTRYLIST structure returned by the function lineGetCountry. This structure cannot be extended. It is defined as follows in TAPI.pas:

```
PLineCountryEntry = ^TLineCountryEntry;
linecountryentry_tag = packed record
  dwCountryID,                           // TAPI v1.4
  dwCountryCode,                         // TAPI v1.4
  dwNextCountryID,                       // TAPI v1.4
  dwCountryNameSize,                     // TAPI v1.4
  dwCountryNameOffset,                   // TAPI v1.4
  dwSameAreaRuleSize,                    // TAPI v1.4
  dwSameAreaRuleOffset,                  // TAPI v1.4
  dwLongDistanceRuleSize,                // TAPI v1.4
  dwLongDistanceRuleOffset,              // TAPI v1.4
  dwInternationalRuleSize,               // TAPI v1.4
  dwInternationalRuleOffset: DWORD;      // TAPI v1.4
end;
TLineCountryEntry = linecountryentry_tag;
LINECOUNTRYENTRY = linecountryentry_tag;
```

The fields of the LINECOUNTRYENTRY structure are described in Table 8-26.

Table 8-26: Fields of the LINECOUNTRYENTRY structure

Field	Meaning
dwCountryID	This field specifies the country ID of the entry. The country ID is an internal identifier which allows multiple entries to exist in the country list with the same country code (for example, all countries in North America and the Caribbean share country code 1, but they require separate entries in the list).
dwCountryCode	This field specifies the actual country code of the country represented by the entry (that is, the digits that would be dialed in an international call). Only this value should ever be displayed to users (country IDs should never be displayed, as they would be confusing).

Field	Meaning
dwNextCountryID	This field specifies the country ID of the next entry in the country list. Because country codes and IDs are not assigned in any regular numeric sequence, the country list is a single linked list, with each entry pointing to the next. The last country in the list has a dwNextCountryID value of 0. When the LINECOUNTRYLIST structure is used to obtain the entire list, the entries in the list will be in sequence as linked by their dwNextCountryID fields.
dwCountryNameSize	This field specifies the size in bytes of a NULL-terminated string giving the name of the country.
dwCountryNameOffset	This field specifies the offset in bytes from the beginning of the LINECOUNTRYLIST structure of a NULL-terminated string giving the name of the country.
dwSameAreaRuleSize	This field specifies the size in bytes of a NULL-terminated ASCII string containing the dialing rule for direct-dialed calls to the same area code.
dwSameAreaRuleOffset	This field specifies the offset in bytes from the beginning of the LINECOUNTRYLIST structure of a NULL-terminated ASCII string containing the dialing rule for direct-dialed calls to the same area code.
dwLongDistanceRuleSize	This field specifies the size in bytes of a NULL-terminated ASCII string containing the dialing rule for direct-dialed calls to other areas in the same country.
dwLongDistanceRuleOffset	This field specifies the offset in bytes from the beginning of the LINECOUNTRYLIST structure of a NULL-terminated ASCII string containing the dialing rule for direct-dialed calls to other areas in the same country.
dwInternationalRuleSize	This field specifies the size in bytes of a NULL-terminated ASCII string containing the dialing rule for direct-dialed calls to other countries.
dwInternationalRuleOffset	This field specifies the offset in bytes from the beginning of the LINECOUNTRYLIST structure of a NULL-terminated ASCII string containing the dialing rule for direct-dialed calls to other countries.

function lineGetIcon TAPI.pas

Syntax

```
function lineGetIcon(dwDeviceID: DWORD; lpszDeviceClass: PChar; lphIcon:
PHICON): Longint; stdcall;
```

Description

This function allows an application to retrieve a service line device-specific (or provider-specific) icon for display to the user.

Parameters

dwDeviceID: DWORD indicating the line device whose icon is requested

lpszDeviceClass: A pointer (LPCSTR) to a NULL-terminated string that identifies a device class name. This device class allows the application to select a specific sub-icon applicable to that device class. This parameter is optional and can be left NULL or empty, in which case the highest-level icon associated with the line device rather than a specified media stream device would be selected.

lphIcon: A pointer (PHICON) to a memory location in which the handle to the icon is returned.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BAD-DEVICEID, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALDEVICECLASS, LINE-ERR_UNINITIALIZED, LINEERR_NOMEM, and LINEERR_NODEVICE.

See Also

lineGetID

Example

Listing 8-19 shows how to get a line device's icon.

Listing 8-19: Getting a line device's icon

```
function TTapiInterface.GetLineIcon: boolean;
begin
  if fPLineIcon=Nil then
    fPLineIcon := AllocMem(1000);
  TapiResult := lineGetIcon(0, Nil, fPLineIcon);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

function lineSetAppSpecific TAPI.pas

Syntax

function lineSetAppSpecific(hCall: HCALL; dwAppSpecific: DWORD): Longint; stdcall;

Description

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

Parameters

hCall: A handle (HCALL) to the call whose application-specific field needs to be set. The application must be an owner of the call. The call state of *hCall* can be any state.

dwAppSpecific: A DWORD holding the new content of the *dwAppSpecific* field for the call's LINECALLINFO structure. This value is not interpreted by the Telephony API.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_NOTOWNER, LINEERR_OPERATIONUNAVAIL, or LINEERR_OPERATIONFAILED.

See Also

LINE_CALLINFO, LINECALLINFO, lineGetCallInfo

Example

Listing 8-20 shows how to call the lineSetAppSpecific() function.

Listing 8-20: Calling the lineSetAppSpecific() function

```
function TTapiInterface.SetDevSpecificInfo(AppSpecificInfo : DWord): boolean;
begin
  TapiResult := lineSetAppSpecific(fCall, AppSpecificInfo);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

function lineSetCurrentLocation TAPI.pas

Syntax

function lineSetCurrentLocation(hLineApp: HLINEAPP; dwLocation: DWORD): Longint; stdcall;

Description

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

Parameters

hLineApp: The application handle (HLINEAPP) returned by lineInitializeEx(). If an application has not yet called the lineInitializeEx() function, it can set the *hLineApp* parameter to NIL.

dwLocation: A DWORD specifying a new value for the CurrentLocation entry in the [Locations] section in the registry. It must contain a valid permanent ID of a Location entry in the [Locations] section, as obtained from lineGetTranslateCaps(). If it is valid, the CurrentLocation entry is updated.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INIFILECORRUPT, LINEERR_NOMEM, LINEERR_INVALAPPHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALLOCATION, LINEERR_RESOURCEUNAVAIL, LINEERR_NODRIVER, and LINEERR_UNINITIALIZED.

See Also

lineGetTranslateCaps, lineInitializeEx

Example

Listing 8-21 shows how to change the current location.

Listing 8-21: Changing the current location

```
function TTapiInterface.SetCurrentLocation(ALocation : DWord): boolean;
begin
  TapiResult := lineSetCurrentLocation(fLineApp, ALocation);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```


Summary

In this chapter we have taken a detailed look at initializing and configuring TAPI. While we have covered most of the essential topics, there is one that we have postponed until the next chapter, handling TAPI messages. The reason is simple—this is one of the most important and involved of TAPI topics and deserves its own chapter. After that, we'll be prepared to explore the two most common TAPI tasks, placing and accepting phone calls.

Chapter 9

Handling TAPI Line Messages

In the previous chapter we discussed initializing and configuring TAPI. While we have covered most of the essential topics, there is one that we have postponed until this chapter—handling TAPI messages. As you are no doubt aware, Windows is an event-driven operating system. To maintain its hardware independence, TAPI relies on messages sent from the Windows operating system to indicate changing hardware states. In this chapter we will examine all of the messages that relate to TAPI lines.

Line Callback

Messages enable application programmers to react to changes and inform the user about those developments. As with many other technologies and Windows' interfaces to those technologies, TAPI provides a callback mechanism so that Windows can send these messages back to your application. The callback routine prototype is declared as follows in TAPI.pas:

function TLineCallback *TAPI.pas*

Syntax

TLineCallback = procedure(hDevice, dwMessage: DWORD; dwInstance, dwParam1, dwParam2, dwParam3: DWORD_PTR) stdcall; LINECALLBACK = TLineCallback;

Description

This function serves as a placeholder for the application-supplied function name.

Parameters

hDevice: A DWORD that serves as a handle to either a line device or a call associated with the callback. You can determine the specific nature of this handle—line handle or call handle—by the context provided by *dwMessage*.

Even though this parameter refers to a handle, applications must use the DWORD type for this parameter because using the THandle type may generate an error.

dwMessage: A line or call device message

dwInstance: Callback instance data passed back to the application in the callback. Note that this DWORD is not interpreted by TAPI.

dwParam1: One parameter (DWORD_PTR) for the message

dwParam2: A second parameter (DWORD_PTR) for the message

dwParam3: A third parameter (DWORD_PTR) for the message

Return Value

No return value

You will always need to define your own callback routine when invoking TAPI. We'll discuss that process presently. The LINE_ messages carry the information. They tend to be expanded in each new version of TAPI and are declared as constant values in TAPI.PAS as follows:

```
const
  LINE_ADDRESSSTATE          = 0;
  LINE_CALLINFO              = 1;
  LINE_CALLSTATE             = 2;
  LINE_CLOSE                 = 3;
  LINE_DEVSPECIFIC           = 4;
  LINE_DEVSPECIFICFEATURE    = 5;
  LINE_GATHERDIGITS          = 6;
  LINE_GENERATE              = 7;
  LINE_LINEDEVSTATE          = 8;
  LINE_MONITORDIGITS         = 9;
  LINE_MONITORMEDIA          = 10;
  LINE_MONITORTONE           = 11;
  LINE_REPLY                 = 12;
  LINE_REQUEST               = 13;
  LINE_CREATE                = 19;         // TAPI v1.4
  LINE_AGENTSPECIFIC         = 21;         // TAPI v2.0
  LINE_AGENTSTATUS           = 22;         // TAPI v2.0
  LINE_APPNEWCALL            = 23;         // TAPI v2.0
  LINE_PROXYREQUEST          = 24;         // TAPI v2.0
  LINE_REMOVE                = 25;         // TAPI v2.0
  LINE_AGENTSESSIONSTATUS    = 27;         // TAPI v2.2
  LINE_QUEUESTATUS           = 28;         // TAPI v2.2
  LINE_AGENTSTATUSEX         = 29;         // TAPI v2.2
  LINE_GROUPSTATUS           = 30;         // TAPI v2.2
  LINE_PROXYSTATUS           = 31;         // TAPI v2.2
  LINE_APPNEWCALLHUB         = 32;         // TAPI v3.0
  LINE_CALLHUBCLOSE          = 33;         // TAPI v3.0
  LINE_DEVSPECIFICEX         = 34;         // TAPI v3.0
```

As mentioned above and in the TAPI Help file, with each initialization option (such as the Hidden Windows one that we use in our sample code) your application must specify a way to handle TAPI messages. Here we use a callback routine to handle TAPI messages. Such a callback routine generally consists of a large case statement that can respond to all of the possible messages from the particular device. These may be used to notify your application when asynchronous events occur. They are sent to an application via the message notification mechanism the application specified when it called the lineInitializeEx() function. Since we are not doing this in a Delphi form unit, but rather in our TAPI class unit, we send custom messages back to the calling application so that it may take actions or display information. Those messages are defined in our TAPIIntf.pas as follows:

```
Const
  WM_TapiErrReceived = WM_User + 117;
  WM_TapiMsgReceived = WM_User + 118;
  WM_TapiIncomingCall = WM_User + 119;
```

Each of these messages is fired from within the callback function, where we also store error or status messages in string variables. While the prototype of the callback routine is very simple, its implementation is not. Often it includes several long case statements at several levels. Below we show the callback routines for both line and phone devices:

```
procedure ALineCallBack(hDevice, dwMessage, dwInstance, dwParam1,
  dwParam2, dwParam3 : DWORD); stdcall;
begin
  case dwMessage of
    LINE_ADDRESSSTATE:
      case dwParam2 of    //
        LINEADDRESSSTATE_OTHER:
          TapiInterface.OnSendTapiMessage
            ('Address-status items other than the common ones '+
          'have changed.');
        LINEADDRESSSTATE_DEVSPECIFIC: TapiInterface.OnSendTapiMessage
          ('A device-specific ' +
          'item of the address status has changed');
        LINEADDRESSSTATE_INUSEZERO: TapiInterface.OnSendTapiMessage
          ('address has changed to idle ' +
          ' is now in use by zero stations');
        LINEADDRESSSTATE_INUSEONE:
         TapiInterface.OnSendTapiMessage
           ('Address has changed from state idle or from being used by many ' +
          'bridged stations to the state of being used by just one station; ');
        LINEADDRESSSTATE_INUSEMANY: TapiInterface.OnSendTapiMessage
          ('monitored or bridged address has ' +
          'changed from the state of being used by one station to that of ' +
          'being used by more than one station; ');
        LINEADDRESSSTATE_NUMCALLS: TapiInterface.OnSendTapiMessage
          ('the number of calls on  '+
          'the address has changed for some reason');
        LINEADDRESSSTATE_FORWARD: TapiInterface.OnSendTapiMessage
          ('forwarding status of the ' +
          'address has changed');
        LINEADDRESSSTATE_TERMINALS: TapiInterface.OnSendTapiMessage
          ('terminal settings ' +
          'for the address have changed');
        LINEADDRESSSTATE_CAPSCHANGE: TapiInterface.OnSendTapiMessage
          ('One or more LINEADDRESSCAPS ' +
          'fields  have changed');
```

```
    Else  TapiInterface.OnSendTapiMessage
        ('Undefined LINEADDRESSSTATE change');
    end; // dwParam2 case
  LINE_CALLINFO:
    begin
      TapiInterface.OnSendTapiMessage
        ('Call information about a specified call has changed');
        case dwParam1 of    //
          LINECALLINFOSTATE_OTHER  :
            TapiInterface.OnSendTapiMessage
              ('Additional informational items have changed');
          LINECALLINFOSTATE_DEVSPECIFIC :
            TapiInterface.OnSendTapiMessage
              ('Device-specific field of the call-information record ' +
               'has changed');
          LINECALLINFOSTATE_BEARERMODE :
            TapiInterface.OnSendTapiMessage
              ('bearer mode field of the call-information record ' +
               'has changed');
          LINECALLINFOSTATE_RATE :
            TapiInterface.OnSendTapiMessage
              ('rate field of the call-information record has changed');
          LINECALLINFOSTATE_MEDIAMODE:
            TapiInterface.OnSendTapiMessage
              ('media mode field of the call-information record ' +
               'has changed');
          LINECALLINFOSTATE_APPSPECIFIC:
            TapiInterface.OnSendTapiMessage
              ('Application-specific field of the call-information ' +
               'record has changed');
          LINECALLINFOSTATE_CALLID:
            TapiInterface.OnSendTapiMessage
              ('Call ID field of the ' +
               'call-information record has changed');
          LINECALLINFOSTATE_RELATEDCALLID:
            TapiInterface.OnSendTapiMessage
              ('related call ID field of the call-information record ' +
                'has changed');
          LINECALLINFOSTATE_ORIGIN:
            TapiInterface.OnSendTapiMessage
              ('Origin field of the ' +
               'call-information record has changed');
          LINECALLINFOSTATE_REASON:
            TapiInterface.OnSendTapiMessage
              ('Reason field of the call-information record has changed');
          LINECALLINFOSTATE_COMPLETIONID:
            TapiInterface.OnSendTapiMessage
              ('Completion ID field of the call-information ' +
               'record has changed');
          LINECALLINFOSTATE_NUMOWNERINCR:
            TapiInterface.OnSendTapiMessage
              ('Number of owner fields in the call-information ' +
               'record was increased');
          LINECALLINFOSTATE_NUMOWNERDECR:
            TapiInterface.OnSendTapiMessage
              ('Number of owner fields in the call-information ' +
               'record was decreased');
          LINECALLINFOSTATE_NUMMONITORS:
            TapiInterface.OnSendTapiMessage
              ('Number of monitors fields in the call-information ' +
```

```
              'record has changed');
          LINECALLINFOSTATE_TRUNK:
            TapiInterface.OnSendTapiMessage
              ('Trunk field of the call information record has changed');
          LINECALLINFOSTATE_CALLERID :
            TapiInterface.OnSendTapiMessage
              ('one of the callerID-related fields of the call ' +
               'information record has changed');
          LINECALLINFOSTATE_CALLEDID:
            TapiInterface.OnSendTapiMessage
              ('one of the calledID-related fields of the call ' +
               'information record has changed');
          LINECALLINFOSTATE_CONNECTEDID:  TapiInterface.OnSendTapiMessage
            ('one of the connectedID-related fields of the call '
             + 'information record has changed');
          LINECALLINFOSTATE_REDIRECTIONID:
            TapiInterface.OnSendTapiMessage
              ('one of the redirectionID-related fields of the ' +
                'call information record has changed');
          LINECALLINFOSTATE_REDIRECTINGID:
            TapiInterface.OnSendTapiMessage
              ('one of the redirectingID-related fields of the ' +
               'call information record has changed');
          LINECALLINFOSTATE_DISPLAY:
            TapiInterface.OnSendTapiMessage
              ('Display field of call information record has changed');
          LINECALLINFOSTATE_USERUSERINFO:
            TapiInterface.OnSendTapiMessage
              ('User-to-user information of call information record ' +
               'has changed');
          LINECALLINFOSTATE_HIGHLEVELCOMP:
            TapiInterface.OnSendTapiMessage
              ('high-level compatibility field of the call ' +
               'information record has changed');
          LINECALLINFOSTATE_LOWLEVELCOMP:
            TapiInterface.OnSendTapiMessage
              ('the low-level compatibility field of the call ' +
               'information record has changed');
          LINECALLINFOSTATE_CHARGINGINFO:
            TapiInterface.OnSendTapiMessage
              ('the charging information of the call information ' +
               'record has changed');
          LINECALLINFOSTATE_TERMINAL:
            TapiInterface.OnSendTapiMessage
              ('the terminal mode ' +
               'information of the call information record has changed');
          LINECALLINFOSTATE_DIALPARAMS:
            TapiInterface.OnSendTapiMessage
              ('the dial parameters of the call information record ' +
               'has changed');
          LINECALLINFOSTATE_MONITORMODES:  TapiInterface.OnSendTapiMessage
            ('one or more call ' +
             'information fields has changed');
        Else   TapiInterface.OnSendTapiMessage
            ('Other LINECALLINFOSTATE information has changed');
        end;    // case
    end;
  LINE_CALLSTATE:
    begin //reports asynchronous responses
    case dwParam1 of
```

```
        LINECALLSTATE_IDLE:
          begin
            TapiInterface.CallState := csIdle;
            TapiInterface.OnSendTapiMessage(
              'The call is idle - no call actually exists.');
                  end;
        LINECALLSTATE_OFFERING:
          begin
            TapiInterface.CallState := csOffering;
            TapiInterface.CurrentCall := dwParam1;
            if dwParam3<>LINECALLPRIVILEGE_OWNER then
              if NOT TapiInterface.SetCallPrivilege
                (TapiInterface.CurrentCall, cplOwner) then
                TapiInterface.OnSendTapiMessage
                ('Cannot accept call because we don''t '+
                 'have owner privileges .')
              else
                begin
                  TapiInterface.OnSendTapiMessage
                    ('Attempting to accept incoming call');
                       lineAccept(TapiInterface.CurrentCall, Nil, 0);
                  SendMessage(MainInstance, WM_TapiIncomingCall, 0, 0);
                end;
          end;
        LINECALLSTATE_ACCEPTED:
          begin
            TapiInterface.CallState := csAccepted;
            TapiInterface.OnSendTapiMessage(
              'The call was offering and has been accepted.');
            if TapiInterface.App.MessageBox('Do you want to accept this call?',
               'Incoming Phone Call', MB_OKCANCEL + MB_ICONQUESTION)=IDOK  then
               lineAnswer(TapiInterface.CurrentCall, Nil, 0);
          end;
        LINECALLSTATE_DIALTONE:
          begin
            TapiInterface.CallState := csDialtone;
            TapiInterface.OnSendTapiMessage('The call is receiving a dial tone.');
            TapiInterface.PlaceCall;
          end;
        LINECALLSTATE_DIALING:
          begin
            TapiInterface.CallState := csDialing;
            TapiInterface.OnSendTapiMessage('Dialing ' +
              TapiInterface.PhoneNumber);
          end;
        LINECALLSTATE_RINGBACK:
          begin
            TapiInterface.CallState := csRingback;
            TapiInterface.OnSendTapiMessage
              ('The call is receiving ringback.');
          end;
        LINECALLSTATE_BUSY:
          begin // note
            TapiInterface.CallState := csBusy;
            case dwParam2 of
              LINEBUSYMODE_STATION:
                TapiInterface.OnSendTapiMessage(
                  'Busy signal; called party''s station is busy.');
              LINEBUSYMODE_TRUNK:
                TapiInterface.OnSendTapiMessage(
```

```
              'Busy signal; trunk or circuit is busy.');
          LINEBUSYMODE_UNKNOWN:
            TapiInterface.OnSendTapiMessage(
              'Busy signal; specific mode is currently unkown');
          LINEBUSYMODE_UNAVAIL:
            TapiInterface.OnSendTapiMessage(
              'Busy signal; specific mode is unavailable');
        else
          TapiInterface.OnSendTapiMessage(
            'The call is receiving an unidentifiable busy tone.');
        end;
        TapiInterface.ShutdownLine;
      end;
    LINECALLSTATE_SPECIALINFO:
      begin
        TapiInterface.CallState := csSpecial;
        TapiInterface.OnSendTapiMessage(
           'Special information is sent by the network.');
      end;
    LINECALLSTATE_CONNECTED:
      begin
        TapiInterface.CallState := csConnected;
        TapiInterface.OnSendTapiMessage
          ('The call has been established and the connection is made.');
        TapiInterface.OnSendTapiMessage('LCB (LINE_CALLSTATE): ' +
           'The call has been established and the connection is made.');
      end;
    LINECALLSTATE_PROCEEDING:
      begin
        TapiInterface.CallState := csProceeding;
        TapiInterface.OnSendTapiMessage
          ('Dialing has completed and the call is proceeding.');
        Exit;
      end;
    LINECALLSTATE_ONHOLD:
      begin
        TapiInterface.CallState := csOnhold;
        TapiInterface.OnSendTapiMessage
          ('The call is on hold by the switch.');
      end;
    LINECALLSTATE_CONFERENCED:
      begin
        TapiInterface.CallState := csConferenced;
        TapiInterface.OnSendTapiMessage
          ('The call is ' +
          'currently a member of a multi-party conference call.');
      end;
    LINECALLSTATE_ONHOLDPENDCONF:
      begin
        TapiInterface.CallState := csOnholdconf;
        TapiInterface.OnSendTapiMessage
          ('The call is currently ' +
          'on hold while it is being added to a conference.');
      end;
    LINECALLSTATE_ONHOLDPENDTRANSFER :
      begin
        TapiInterface.CallState := csOnholdPendTransfer;
        TapiInterface.OnSendTapiMessage
          ('The call is currently ' +
          'on hold while a transfer is pending.');
```

```
      end;
    LINECALLSTATE_DISCONNECTED:
      begin
        TapiInterface.CallState := csDisconnected;
        TapiInterface.OnSendTapiMessage
          ('The line has been disconnected.');
        case dwParam2 of
          LINEDISCONNECTMODE_NORMAL:
            TapiInterface.OnSendTapiMessage
              (#9 + 'This is a "normal" disconnect request.');
          LINEDISCONNECTMODE_UNKNOWN:
            TapiInterface.OnSendTapiMessage
              (#9+'The reason for the disconnect request is unknown.');
          LINEDISCONNECTMODE_REJECT:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The remote user has rejected the call.');
          LINEDISCONNECTMODE_PICKUP:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The call was picked up from elsewhere.');
          LINEDISCONNECTMODE_FORWARDED:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The call was forwarded by the switch.');
          LINEDISCONNECTMODE_BUSY:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The remote user''s station is busy.');
          LINEDISCONNECTMODE_NOANSWER:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The remote user''s station does not answer.');
          LINEDISCONNECTMODE_BADADDRESS:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The destination address in invalid.');
          LINEDISCONNECTMODE_UNREACHABLE:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The remote user could not be reached.');
          LINEDISCONNECTMODE_CONGESTION:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The network is congested.');
          LINEDISCONNECTMODE_INCOMPATIBLE:
            TapiInterface.OnSendTapiMessage(#9 +
              'The remote user''s station equipment is incompatible');
          LINEDISCONNECTMODE_UNAVAIL:
            TapiInterface.OnSendTapiMessage
              (#9 + 'The reason for the disconnect is unavailable');
        Else TapiInterface.OnSendTapiMessage
              (#9 + 'The reason is not known');
        end;
      end;
    LINECALLSTATE_UNKNOWN:
      begin
        TapiInterface.CallState := csUnknown;
        TapiInterface.OnSendTapiMessage
         ('The state of the call is not known.');
      end;
  else
    begin
      TapiInterface.CallState := csUnknown;
      TapiInterface.OnSendTapiMessage
       ('The state of the call is not known.');
    end;
  end;
```

```
    end;
  LINE_LINEDEVSTATE:
    case dwParam1 of // incomplete list
       LINEDEVSTATE_RINGING:
          TapiInterface.OnSendTapiMessage
            ('(Ringing) Ring, ring, ring...');
       LINEDEVSTATE_CONNECTED:
          TapiInterface.OnSendTapiMessage
            ('Connected...');
       LINEDEVSTATE_DISCONNECTED:
          TapiInterface.OnSendTapiMessage
            ('Disconnected...');
       LINEDEVSTATE_MSGWAITON:
          TapiInterface.OnSendTapiMessage
            ('"message waiting" indicator is turned on.');
       LINEDEVSTATE_MSGWAITOFF:
          TapiInterface.OnSendTapiMessage(
            '"message waiting" indicator is turned off.');
       LINEDEVSTATE_NUMCOMPLETIONS:
          TapiInterface.OnSendTapiMessage
            ('The number of outstanding' +
             '  call completions on the line device has changed.');
       LINEDEVSTATE_INSERVICE:
          TapiInterface.OnSendTapiMessage
            ('The line is connected to TAPI');
       LINEDEVSTATE_OUTOFSERVICE:
          TapiInterface.OnSendTapiMessage
            ('The line is out of service');
       LINEDEVSTATE_MAINTENANCE:
          TapiInterface.OnSendTapiMessage
            ('Line maintenance heing performed');
       LINEDEVSTATE_OPEN:
          TapiInterface.OnSendTapiMessage
            ('Line opened by another application');
       LINEDEVSTATE_CLOSE:
          TapiInterface.OnSendTapiMessage
            ('Line closed by another application');
       LINEDEVSTATE_NUMCALLS:
          TapiInterface.OnSendTapiMessage
            ('Number of calls on line has changed');
       LINEDEVSTATE_TERMINALS:
          TapiInterface.OnSendTapiMessage
            ('Terminal settings have changed');
       LINEDEVSTATE_ROAMMODE:
          TapiInterface.OnSendTapiMessage
            ('Cellular roaming mode has changed');
       LINEDEVSTATE_BATTERY:
          TapiInterface.OnSendTapiMessage
            ('Cellular battery level has changed');
       LINEDEVSTATE_SIGNAL:
          TapiInterface.OnSendTapiMessage
            ('Cellular signal has changed');
       LINEDEVSTATE_DEVSPECIFIC:
          TapiInterface.OnSendTapiMessage
            ('Device-specific information has changed');
       LINEDEVSTATE_LOCK:
          TapiInterface.OnSendTapiMessage
            ('Lock status of line has changed');
       LINEDEVSTATE_CAPSCHANGE:
          TapiInterface.OnSendTapiMessage
```

```
              ('LCB (LINE_LINEDEVSTATE): capabilities of line have changed');
         LINEDEVSTATE_TRANSLATECHANGE:
            TapiInterface.OnSendTapiMessage
              ('Capabilities of line have changed');
         LINEDEVSTATE_REINIT: // line device has changed or been modified
            if (dwParam2 = 0) then
            begin
               TapiInterface.OnSendTapiMessage
                 ('Shutdown required');
               TapiInterface.ShutdownLine;
            end;
         LINEDEVSTATE_OTHER:
            TapiInterface.OnSendTapiMessage
              ('Other line device state.');
      else TapiInterface.OnSendTapiMessage
              ('Other line device state.');
      end;  // inner case
    LINE_REPLY:
      if (dwParam2 = 0) then
         TapiInterface.OnSendTapiMessage
           ('LineMakeCall completed successfully')
      else
         TapiInterface.OnSendTapiMessage
           ('LineMakeCall failed');
{$IFDEF TAPI14}
    LINE_CREATE:
       TapiInterface.OnSendTapiMessage('Line Acreated');
{$ENDIF}
{$IFDEF TAPI20}
    LINE_AGENTSPECIFIC:
      TapiInterface.OnSendTapiMessage
        ('status of an ACD agent on a currently open line has changed');
    LINE_AGENTSTATUS:
      begin
        TapiInterface.OnSendTapiMessage
          ('Status of an ACD agent on a currently open line has changed');
          if LINEAGENTSTATE_GROUP in [dwParam2] then
            TapiInterface.OnSendTapiMessage
              ('The Group List in LINEAGENTSTATUS has been updated. ');
          if  LINEAGENTSTATE_STATE in [dwParam2] then
            TapiInterface.OnSendTapiMessage
              ('The dwState member in LINEAGENTSTATUS has been updated.');
          if  LINEAGENTSTATE_NEXTSTATE in [dwParam2] then
            TapiInterface.OnSendTapiMessage
              ('The dwNextState member in LINEAGENTSTATUS has been updated. ');
          if  LINEAGENTSTATE_ACTIVITY in [dwParam2] then
            TapiInterface.OnSendTapiMessage('The ActivityID, ActivitySize, or ActivityOffset
               members in ' + 'LINEAGENTSTATUS has been updated. ');
          if LINEAGENTSTATE_ACTIVITYLIST  in [dwParam2] then
            TapiInterface.OnSendTapiMessage
              ('The List member in LINEAGENTACTIVITYLIST has been updated. ' +
              'The application can call lineGetAgentActivityList to get the ' +
              'updated list.');
          if  LINEAGENTSTATE_GROUPLIST in [dwParam2] then
            TapiInterface.OnSendTapiMessage
              ('The List member in LINEAGENTGROUPLIST has been updated. ' +
              'The application can call lineGetAgentGroupList to get the ' +
              'updated list.');
          if LINEAGENTSTATE_CAPSCHANGE  in [dwParam2] then
            TapiInterface.OnSendTapiMessage
```

```
              ('The capabilities in LINEAGENTCAPS have been updated. ' +
              'The application can call lineGetAgentCaps to get the updated ' +
              'list. ');
          if LINEAGENTSTATE_VALIDNEXTSTATES  in [dwParam2] then
            TapiInterface.OnSendTapiMessage
              ('The dwValidNextStates member in LINEAGENTSTATUS has been ' +
               'updated. ');
      end;
    LINE_APPNEWCALL:
      TapiInterface.OnSendTapiMessage
        ('A new call handle has been created spontaneously');
    LINE_PROXYREQUEST:
      TapiInterface.OnSendTapiMessage
        ('Request sent to a registered proxy function handler');
    LINE_REMOVE:
      TapiInterface.OnSendTapiMessage
        ('A device has been removed from this line');
{$ENDIF}
{$IFDEF TAPI22}
    LINE_AGENTSESSIONSTATUS:
      TapiInterface.OnSendTapiMessage
        ('The status of an ACD agent session has changed.');
    LINE_QUEUESTATUS:
      TapiInterface.OnSendTapiMessage
        ('The status of an ACD queue has changed');
    LINE_AGENTSTATUSEX:
      TapiInterface.OnSendTapiMessage
        ('The status of an ACD agent has changed');
    LINE_GROUPSTATUS:
      TapiInterface.OnSendTapiMessage
        ('The status of an ACD group has changed');
    LINE_PROXYSTATUS:
      TapiInterface.OnSendTapiMessage('The available proxies have changed');
{$ENDIF}
{$IFDEF TAPI30}
    LINE_APPNEWCALLHUB:
      TapiInterface.OnSendTapiMessage('A new call hub has been created.');
    LINE_CALLHUBCLOSE:
      TapiInterface.OnSendTapiMessage('A call hub has been closed.');
    LINE_DEVSPECIFICEX:
      TapiInterface.OnSendTapiMessage
        ('A device-specific event has occurred on a line, address, or call');
    else
      TapiInterface.OnSendTapiMessage
        ('An unspecified event has occurred on a line, address, or call');
{$ENDIF}
  end; // outer case
end;

// Callback to handle TAPI messages from the phone device
// For future development
procedure APhoneCallBack(hDevice, dwMessage, dwInstance, dwParam1,
  dwParam2, dwParam3 : DWORD); stdcall;
begin
  begin // Write TAPI results to string list for use
        // by Delphi components in TAPIForm unit
    case dwMessage of
    PHONE_BUTTON:
      begin
      case dwParam2 of
```

```
        PHONEBUTTONMODE_CALL:
          TapiInterface.OnSendTapiMessage
            ('The button is assigned to a call appearance.');
        PHONEBUTTONMODE_FEATURE:
          TapiInterface.OnSendTapiMessage
            ('The button is assigned to requesting features from the switch,'
          + 'such as hold, conference, and transfer.');
        PHONEBUTTONMODE_KEYPAD:
          TapiInterface.OnSendTapiMessage
            ('The button is one of the twelve keypad buttons,' +
          '''0'' through ''9'', ''*'', and ''#''.');
        PHONEBUTTONMODE_LOCAL:
          TapiInterface.OnSendTapiMessage
            ('The button is a local function button, such as mute or '
            + 'volume control.');
        PHONEBUTTONMODE_DISPLAY:
          TapiInterface.OnSendTapiMessage
            ('The button is a "soft" button associated with the phone''s display.'
            + ' A phone set can have zero or more display buttons.');

      end;    // case
      end;
    PHONE_CLOSE: TapiInterface.OnSendTapiMessage
      ('Phone device has been closed');
    PHONE_CREATE: TapiInterface.OnSendTapiMessage
      ('A New Phone Device has been created');
    PHONE_DEVSPECIFIC: TapiInterface.OnSendTapiMessage
      ('A device-specific event has occurred');
    PHONE_REMOVE: TapiInterface.OnSendTapiMessage
      ('A phone device has been removed from the system');
    PHONE_REPLY:
      begin
        if dwParam2=0 then
          TapiInterface.OnSendTapiMessage
            ('Async Function Call successful')
        else
          TapiInterface.OnSendTapiMessage
            ('Async Function Call not successful')
      end;
    PHONE_STATE:
      begin
        case dwParam1 of    //
          PHONESTATE_OTHER: TapiInterface.OnSendTapiMessage
            ('Phone-status items other than expected ' +
             'ones have changed.');
          PHONESTATE_CONNECTED: TapiInterface.OnSendTapiMessage
            ('The connection between the phone device ' +
             'and TAPI established.');
          PHONESTATE_DISCONNECTED: TapiInterface.OnSendTapiMessage
            ('Connection between phone device and TAPI broken.');
          PHONESTATE_OWNER: TapiInterface.OnSendTapiMessage
            ('Number of owners for the phone device has changed.');
          PHONESTATE_MONITORS: TapiInterface.OnSendTapiMessage
            ('Number of monitors for the phone device has changed.');
          PHONESTATE_DISPLAY: TapiInterface.OnSendTapiMessage
            ('The display of the phone has changed.');
          PHONESTATE_LAMP: TapiInterface.OnSendTapiMessage
            ('A lamp of the phone has changed.');
          PHONESTATE_RINGMODE: TapiInterface.OnSendTapiMessage
            ('The ring mode of the phone has changed.');
```

```
          PHONESTATE_RINGVOLUME: TapiInterface.OnSendTapiMessage
            ('The ring volume of the phone has changed.');
          PHONESTATE_HANDSETHOOKSWITCH: TapiInterface.OnSendTapiMessage
            ('The handset hookswitch state has changed.');
          PHONESTATE_HANDSETVOLUME: TapiInterface.OnSendTapiMessage
            ('The handset''s speaker volume setting ' +
            'has changed.');
          PHONESTATE_HANDSETGAIN: TapiInterface.OnSendTapiMessage
            ('The handset''s microphone gain ' +
             'setting has changed.');
          PHONESTATE_SPEAKERHOOKSWITCH: TapiInterface.OnSendTapiMessage
            ('The speakerphone''s hookswitch ' +
             'state has changed.');
          PHONESTATE_SPEAKERVOLUME: TapiInterface.OnSendTapiMessage
            ('The speakerphone''s speaker volume' +
             ' setting has changed.');
          PHONESTATE_SPEAKERGAIN: TapiInterface.OnSendTapiMessage
            ('The speakerphone''s microphone gain ' +
             'setting state has changed.');
          PHONESTATE_HEADSETHOOKSWITCH: TapiInterface.OnSendTapiMessage
            ('The headset''s hookswitch state has changed.');
          PHONESTATE_HEADSETVOLUME: TapiInterface.OnSendTapiMessage
            ('The headset''s speaker volume setting has changed.');
          PHONESTATE_HEADSETGAIN: TapiInterface.OnSendTapiMessage
            ('The headset''s microphone gain setting has changed.');
          PHONESTATE_SUSPEND: TapiInterface.OnSendTapiMessage
            ('The application''s use of the phone ' +
             'device is temporarily suspended.');
          PHONESTATE_RESUME: TapiInterface.OnSendTapiMessage
            ('The application''s use of the phone device' +
             'has resumed after having been suspended for some time.');
          PHONESTATE_DEVSPECIFIC: TapiInterface.OnSendTapiMessage
            ('The phone''s device-specific information' +
             ' has changed.');
          PHONESTATE_REINIT: TapiInterface.OnSendTapiMessage
            ('Items have changed in the configuration of phone devices.');
          PHONESTATE_CAPSCHANGE: TapiInterface.OnSendTapiMessage
            ('One or more of PHONECAPS'' fields has changed.');
          PHONESTATE_REMOVED: TapiInterface.OnSendTapiMessage
            ('A device is being removed from the system ' +
             'by the service provider');
        end;    // case
      end;
    end;
  end;
end;
```

The TAPI Help file provides very detailed information about each message. We have included most of that information here. First, we will provide an overview. We group messages into two different tables: The 20 older messages (TAPI 2.0 and earlier) are in Table 9-1 and the newer messages are in Table 9-2. After that, we will provide detailed information about each one, its use, and its parameters.

Table 9-1: Older TAPI messages (Version 2.0 and earlier)

Message	Meaning
LINE_ADDRESSSTATE	This message will be sent to an application when the status of an address changes on a currently open line. You can call the lineGetAddressStatus() function to determine the current status of the address.
LINE_AGENTSPECIFIC	This message will be sent to an application when the status of an ACD agent changes on a currently open line. You can call the lineGetAgentStatus() function to determine the current status of the agent.
LINE_AGENTSTATUS	This message will be sent to an application when the status of an ACD agent changes on a currently open line.You can call the lineGetAgentStatus() function to determine the current status of the agent.
LINE_APPNEWCALL	This message will be sent to an application to inform it when a new call handle has been spontaneously created on its behalf. This does not include situations in which the handle is created through a TAPI call from an application. In that case the handle will be returned through a pointer parameter passed to the function.
LINE_CALLINFO	This message will be sent to an application when the call information about the specified call has changed. You can call the lineGetCallInfo() function to determine the current call information.
LINE_CALLSTATE	This message will be sent to an application when the status of the specified call has changed. Typically, several such messages will be received during the lifetime of a call. Windows notifies applications of new incoming calls using this message; new calls are initially in the offering state. You can call the lineGetCallStatus() function to retrieve more detailed information about the current status of the call.
LINE_CLOSE	This message will be sent to an application when the specified line device has been forcibly closed. The line device handle or any call handles for calls on the line will no longer be valid once this message has been sent.
LINE_CREATE	This message will be sent to an application to inform it that a new line device had been created.
LINE_DEVSPECIFIC	This message will be sent to an application to notify it about device-specific events occurring on a line, address, or call. The specific meaning of the message and the interpretation of the parameters are device specific.
LINE_DEVSPECIFICFEATURE	This message will be sent to an application to notify it about device-specific events occurring on a line, address, or call. The specific meaning of the message and the interpretation of the parameters are device specific.
LINE_GATHERDIGITS	This message will be sent to an application when the current buffered digit-gathering request has been terminated or canceled. You may examine the digit buffer after this message has been received by an application.
LINE_GENERATE	This message will be sent to an application to notify it that the current digit or tone generation has terminated. Only one such generation request can be in progress on a given call at any time. This message is also sent when either digit or tone generation is canceled.
LINE_LINEDEVSTATE	This message will be sent to an application when the state of a line device has changed. You can call the lineGetLineDevStatus() function to determine the new status of the line.
LINE_MONITORDIGITS	This message will be sent to an application when a digit is detected. The lineMonitorDigits() function controls the process of sending this message.

Message	Meaning
LINE_MONITORMEDIA	This message will be sent to an application when a change in the call's media mode is detected. The lineMonitorMedia() function controls the process of sending this message.
LINE_MONITORTONE	This message will be sent to an application when a tone is detected. The lineMonitorTones() function controls the process of sending this message.
LINE_PROXYREQUEST	This message sends a request to a registered proxy function handler.
LINE_REMOVE	This message will be sent to an application to inform it of the removal (deletion from the system) of a line device. Generally, this is not used for temporary removals, such as extraction of PCMCIA devices. Rather, it is used only in the case of permanent removals in which the service provider would no longer report the device when TAPI was reinitialized.
LINE_REPLY	This message will be sent to an application to report the results of function calls that completed asynchronously.
LINE_REQUEST	This message will be sent to an application to report the arrival of a new request from another application (see Chapter 8, "Line Devices and Essential Operations" for an example).

Table 9-2: Newer TAPI messages (Version 2.2 and later)

Message	Meaning
LINE_AGENTSESSIONSTATUS	This message will be sent when the status of an ACD agent session changes on an agent handler for which an application currently has an open line. This message is generated using the lineProxyMessage() function.
LINE_QUEUESTATUS	This message will be sent when the status of an ACD queue changes on an agent handler for which an application currently has an open line. This message is generated using the lineProxyMessage() function.
LINE_AGENTSTATUSEX	This message will be sent when the status of an ACD agent changes on an agent handler for which an application currently has an open line. This message is generated using the lineProxyMessage() function.
LINE_GROUPSTATUS	This message will be sent when the status of an ACD group changes on an agent handler for which an application currently has an open line. This message is generated using the lineProxyMessage() function.
LINE_PROXYSTATUS	This (listed incorrectly as LINE_QUEUESTATUS in MS Help) message will be sent when the available proxies change on a line that an application currently has open.
TAPI LINE_APPNEWCALLHUB	This message will be sent to inform an application when a new call hub has been created.
TAPI LINE_CALLHUBCLOSE	This message will be sent when a call hub has been closed.
TAPI LINE_DEVSPECIFICEX	This message will be sent to notify an application about device-specific events occurring on a line, address, or call. The meaning of the message and the interpretation of the parameters are device specific.

A message always contains a handle to the object whose behavior it is reporting. The object could be a phone device, a line device, or a call. How does an application determine the type of handle? Actually, it can do this rather easily by examining the message type. As you can tell from Tables 9-1 and 9-2, messages can carry out a number of functions. Often they will notify an application about a change in an object's status. These kinds of messages always provide the object's handle and indicate which status item has changed. Your application can obtain the object's full status by calling one of the "get status" functions.

What actually happens when an event occurs? Invariably, messages may be sent to zero, one, or more applications. The target applications for a message will be determined by various factors. Among these are the following:

- The meaning of the message; the purpose it is attempting to fulfill
- The target application's level of privilege in relationship to the telephony object
- The initiator (application) of the particular request to which the message is responding
- Special message masking set by your application

Issues Involving Messages

There are a number of issues concerning messages, mainly restrictions on where they will be sent. Here are some of the main restrictions:

- The Windows operating system will send asynchronous reply messages only to the application that originated the request; such messages cannot be masked.
- Windows will send messages that signal the completion of digit or tone generation or the gathering of digits only to the application that initiated the particular task.
- Windows will send messages that indicate a change in line or address states to all applications that have opened the line, provided that the message has been enabled via lineSetStatusMessages().
- The operating system will send messages that indicate changes to a call's state or changes to other information regarding a call to all applications that have a handle to the call.
- The system will send messages that signal a digit detection, tone detection, or media mode detection to any application (one or more) that requested monitoring of the particular event.

There are other issues of backward compatability and timing that we'll discuss with certain messages.

As we've implied, there's an intimate relationship between these messages and the callback routine. One of the complex issues is that each message requires different values to be entered or returned in the callback routine. In some cases, a particular parameter could have a large number of such values. To provide a complete reference, we have included this information under the particular messages in this section.

LINE_ADDRESSSTATE Message

Windows will send the LINE_ADDRESSSTATE message to an application when the status of an address on a currently open line has changed. It will send this message to any application that has opened the particular line device and enabled this message. You can control and queue the sending of this message for the various status items by using the lineGetStatusMessages() and lineSetStatusMessages() functions. Address status reporting is disabled by default.

Table 9-3: Parameter values for the LINE_ADDRESSSTATE message

Parameter	Value(s)
hDevice	Line device handle
dwInstance	Callback instance supplied when the line was opened
dwParam1	ID of the address whose status changed
dwParam2	The address state that changed. It can be a combination of one or more of the following values: LINEADDRESSSTATE_OTHER indicates that address-status items other than those listed below have changed. It is recommended that an application check the current address status to determine which items have changed. LINEADDRESSSTATE_DEVSPECIFIC indicates that the device-specific item of the address status has changed. LINEADDRESSSTATE_INUSEZERO indicates that the address has changed to idle (e.g., it is now in use by zero stations). LINEADDRESSSTATE_INUSEONE indicates that the address has changed from the idle state or from being used by many bridged stations to the state of being used by just one station. LINEADDRESSSTATE_INUSEMANY indicates that the monitored or bridged address has changed from the state of being used by one station to that of being used by more than one station. LINEADDRESSSTATE_NUMCALLS indicates that the number of calls on the address has changed (as a result of events such as a new inbound call, an outbound call on the address, or a call changing its hold status). LINEADDRESSSTATE_FORWARD indicates that the forwarding status of the address has changed, including the number of rings for determining a no answer condition (an application should check the address status to determine details about the address's current forwarding status). LINEADDRESSSTATE_TERMINALS indicates that the terminal settings for the address have changed. LINEADDRESSSTATE_CAPSCHANGE indicates that one or more of the fields in the LINEADDRESSCAPS structure for the address have changed because of configuration changes made by the user or other circumstances. You should call the lineGetAddressCaps() function to read the updated structure. Applications that supportTAPI versions less than 1.04 will receive a LINEDEVSTATE_REINIT message, requiring them to shut down and reinitialize their connection to TAPI in order to obtain the updated information.
dwParam3	Not used

LINE_AGENTSPECIFIC Message

Windows will send a LINE_AGENTSPECIFIC message to an application when the status of an ACD agent on a currently open line has changed. You can call the lineGetAgentStatus() function to determine the current status of the agent. This message will not be sent to applications that support older versions of TAPI.

Table 9-4: Parameter values for the LINE_AGENTSPECIFIC message

Parameter	Value(s)
hDevice	An application's handle to the line device
dwInstance	Callback instance supplied when the line was opened
dwParam1	Index into the array of handler extension IDs in the LINEAGENTCAPS structure of the handler extension with which the message is associated
dwParam2	Specific to the handler extension. Generally, this value will be used to cause an application to call the lineAgentSpecific() function in order to gather further details of the message.
dwParam3	Specific to the handler extension

LINE_AGENTSTATUS Message

Windows will send a LINE_AGENTSTATUS message to an application when the status of an ACD agent changes on a currently open line. You can call the lineGetAgentStatus() function to determine the current status of the agent. This message will not be sent to applications that support older versions of TAPI.

Table 9-5: Parameter values for the LINE_AGENTSTATUS message

Parameter	Value(s)
hDevice	An application's handle to the line device on which the agent status has changed
dwInstance	Callback instance supplied when opening the line associated with the call
dwParam1	Identifier of the address on the line on which the agent status has changed
dwParam2	Specifies the agent status that has changed; can be a combination of LINEAGENTSTATE_ constant values (see Table 9-6)
dwParam3	If dwParam2 includes the LINEAGENTSTATUS_STATE bit, this parameter indicates the new value of the dwState member in LINEAGENTSTATUS. Otherwise, this parameter is set to 0.

Table 9-6: LINEAGENTSTATE_ flags

Flag	Meaning
LINEAGENTSTATE_ GROUP	The group list in LINEAGENTSTATUS has been updated.
LINEAGENTSTATE_ STATE	The dwState member in LINEAGENTSTATUS has been updated.
LINEAGENTSTATE_ NEXTSTATE	The dwNextState member in LINEAGENTSTATUS has been updated.

Flag	Meaning
LINEAGENTSTATE_ ACTIVITY	One of the following members in LINEAGENTSTATUS has been updated: ActivityID, ActivitySize, or ActivityOffset.
LINEAGENTSTATE_ ACTIVITYLIST	The list member in LINEAGENTACTIVITYLIST has been updated. An application can call lineGetAgentActivityList() to get the updated list.
LINEAGENTSTATE_ GROUPLIST	The list member in LINEAGENTGROUPLIST has been updated. An application can call lineGetAgentGroupList() to get the updated list.
LINEAGENTSTATE_ CAPSCHANGE	The capabilities in LINEAGENTCAPS have been updated. An application can call lineGetAgentCaps() to get the updated list.
LINEAGENTSTATE_ VALIDSTATES	The dwValidStates member in LINEAGENTSTATUS has been updated.
LINEAGENTSTATE_ VALIDNEXTSTATES	The dwValidNextStates member in LINEAGENTSTATUS has been updated.

LINE_APPNEWCALL Message

Generally, a call handle is created in response to an application's calling a TAPI function. Windows returns that handle through a pointer parameter passed to the function, but sometimes TAPI spontaneously creates a new call handle. The documentation is silent on when this might happen, but we assume that it would be in response to an incoming call of which the telephony application(s) are not aware. To accept such a call, an application must have a handle to that call. In newer versions of TAPI (since version 2), Windows will send a LINE_APP-NEWCALL message to an application to inform it when it has created such a call handle. The parameters of this message (see Table 9-7) provide enough information for a telephony application to create a new call object in the correct context. In particular, those parameters include the handle (*hLine*) to the line device on which the call was created and the identifier (*dwAddressID*) representing the address on the line on which the call appears. These are stored in the first and third parameters, respectively. Finally, this message will always be followed immediately by a LINE_CALLSTATE message indicating the initial state of the call.

For older applications—those which negotiated a TAPI version prior to TAPI 2—the process is a bit more difficult. For these applications, Windows will send only a LINE_CALLSTATE to them. After receiving that particular message, these applications will need to create a new call object (setting its *dwParam3* to a nonzero value). Older applications will not immediately know the call handle and the other information the newer message provides. To get this information, they must call the lineGetCallInfo() function. That function will return the *hLine* and *dwAddressID* associated with the call. However, there are two more issues these older TAPI versions must deal with. In addition to calling the lineGet-CallInfo() function, you must scan all known call handles in order to determine that the call is indeed a new call; finally, you must also make certain that what appears to be a new call handle is not, in fact, one that your application has just deallocated. For more information, see the TAPI Help file.

Table 9-7: Parameter values for the LINE_APPNEWCALL message

Parameter	Value(s)
hDevice	An application's handle to the line device on which the call has been created
dwInstance	The callback instance supplied when opening the call's line
dwParam1	Identifier of the address on the line on which the call appears
dwParam2	An application's handle to the new call
dwParam3	An application's privilege to the new call (LINECALLPRIVILEGE_OWNER or LINECALLPRIVILEGE_MONITOR)

LINE_CALLINFO Message

Windows will send a LINE_CALLINFO message to an application when information about a specified call has changed. This could occur when you call one of the following functions: lineOpen(), lineClose(), lineShutdown(), lineSetCallPrivilege(), lineGetNewCalls(), or lineGetConfRelatedCalls(). TAPI will create an individual LINECALLINFO structure for every inbound and outbound call that contains the basic static information about the call.

You can call the lineGetCallInfo() function to determine the current call information. TAPI will send a LINE_CALLINFO message with an indication of NumOwnersIncr, NumOwnersDecr, and/or NumMonitorsChanged to all applications that already have a handle for that call. This situation can occur when another application is changing ownership or monitoring status of a call. Your telephony applications could use this information when receiving a handle for a call through the LINE_CALLSTATE message or notification through a LINE_CALLINFO message that parts of the call information structure have changed. These messages supply the handle for the call as a parameter.

Be aware that Windows does not send LINE_CALLINFO messages when a notification of a new call is provided in a LINE_CALLSTATE message. The reason is that the call information already reflects the correct number of owners and monitors at the time the LINE_CALLSTATE messages are sent. LINE_CALLINFO messages are also suppressed when TAPI offers a call to monitoring applications through the LINECALLSTATE_UNKNOWN mechanism. Also, be aware that an application that causes a change in the number of owners or monitors (for example, by invoking lineDeallocateCall() or lineSetCallPrivilege()) will not receive a message itself indicating that the change has been made. No LINE_CALLINFO messages will be sent for a call after that call has entered the idle state. TAPI does not report changes in the number of owners and monitors when applications deallocate their handles for the idle call.

Table 9-8: Parameter values for the LINE_CALLINFO message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	Callback instance supplied when opening the call's line
dwParam1	The call information item that has changed. It can be a combination of one or more of the LINECALLINFOSTATE_ values shown in Table 9-9.
dwParam2	Unused
dwParam3	Unused

Table 9-9: LINECALLINFOSTATE_ constants used in the dwParamI field of the LINE_CALLINFO message

Constant	Meaning
LINECALLINFOSTATE_ OTHER	This constant indicates that informational items other than those listed below have changed (in this case, you should check the current call information to determine which items have changed).
LINECALLINFOSTATE_ DEVSPECIFIC	This constant indicates that the device-specific field of the call information record has changed.
LINECALLINFOSTATE_ BEARERMODE	This constant indicates that the bearer mode field of the call information record has changed.
LINECALLINFOSTATE_ RATE	This constant indicates that the rate field of the call information record has changed.
LINECALLINFOSTATE_ MEDIAMODE	This constant indicates that the media mode field of the call information record has changed.
LINECALLINFOSTATE_ APPSPECIFIC	This constant indicates that an application-specific field of the call information record has changed.
LINECALLINFOSTATE_ CALLID	This constant indicates that the call ID field of the call information record has changed.
LINECALLINFOSTATE_ RELATEDCALLID	This constant indicates that the related call ID field of the call information record has changed.
LINECALLINFOSTATE_ ORIGIN	This constant indicates that the origin field of the call information record has changed.
LINECALLINFOSTATE_ REASON	This constant indicates that the reason field of the call information record has changed.
LINECALLINFOSTATE_ COMPLETIONID	This constant indicates that the completion ID field of the call information record has changed.
LINECALLINFOSTATE_ NUMOWNERINCR	This constant indicates that the number of owner fields in the call information record was increased.
LINECALLINFOSTATE_ NUMOWNERDECR	This constant indicates that the number of owner fields in the call information record was decreased.
LINECALLINFOSTATE_ NUMMONITORS	This constant indicates that the number of monitors fields in the call information record has changed.
LINECALLINFOSTATE_ TRUNK	This constant indicates that the trunk field of the call information record has changed.
LINECALLINFOSTATE_ CALLERID	This constant indicates that one of the callerID-related fields of the call information record has changed.
LINECALLINFOSTATE_ CALLEDID	This constant indicates that one of the calledID-related fields of the call information record has changed.

Constant	Meaning
LINECALLINFOSTATE_CONNECTEDID	This constant indicates that one of the connectedID-related fields of the call information record has changed.
LINECALLINFOSTATE_REDIRECTIONID	This constant indicates that one of the redirectionID-related fields of the call information record has changed.
LINECALLINFOSTATE_REDIRECTINGID	This constant indicates that one of the redirectingID-related fields of the call information record has changed.
LINECALLINFOSTATE_DISPLAY	This constant indicates that the display field of the call information record has changed.
LINECALLINFOSTATE_USERUSERINFO	This constant indicates that the user-to-user information of the call information record has changed.
LINECALLINFOSTATE_HIGHLEVELCOMP	This constant indicates that the high-level compatibility field of the call information record has changed.
LINECALLINFOSTATE_LOWLEVELCOMP	This constant indicates that the low-level compatibility field of the call information record has changed.
LINECALLINFOSTATE_CHARGINGINFO	This constant indicates that the charging information of the call information record has changed.
LINECALLINFOSTATE_TERMINAL	This constant indicates that the terminal mode information of the call information record has changed.
LINECALLINFOSTATE_DIALPARAMS	This constant indicates that the dial parameters of the call information record has changed.
LINECALLINFOSTATE_MONITORMODES	This constant indicates that one or more of the digit, tone, or media monitoring fields in the call information record has changed.

LINE_CALLSTATE Message

Windows will send a LINE_CALLSTATE message to an application when the status of the specified call has changed. Windows uses this message to notify applications of new incoming calls, always starting off in the offering state. Typically, an application will receive several such messages during the lifetime of a call.

You can call the lineGetCallStatus() function to retrieve more detailed information about the current status of the call. Windows will send this message to any application that has a handle for that call. The LINE_CALLSTATE message will also notify those applications that have assumed responsibility to monitor calls on a line about the existence and the state of outbound calls. The lineGetCallStatus() function returns the dynamic status of a call, while the lineGetCallInfo() function returns primarily static information about a call. Such status information includes the current call's state, detailed mode information, and a list of the available API functions an application can invoke on the call while the call is in this state. You may want an application to request this information when it receives notification about a call state change through the LINE_CALLSTATE message.

These outbound calls may be established in one of two ways—by other telephony applications or manually by the user (for example, on an attached phone device). The call state of such calls will reflect the actual state of the call, which will not be in the offering state. By examining the call state, an application can determine if the call is an inbound call that it needs to answer.

A LINE_CALLSTATE message with an unknown call state may be sent to a monitoring application by another requesting application. This process will generally be the result of a successful call to any of the following line_ functions: lineMakeCall(), lineForward(), lineUnpark(), lineSetupTransfer(), linePickup(), lineSetupConference(), or linePrepareAddToConference(). The communication process is fairly complex. At the same time that a requesting application is sent a LINE_REPLY (success) for a requested operation, any of the other applications monitoring the line will be sent the LINE_CALLSTATE (unknown) message. The LINE_CALLSTATE message uses information provided by the service provider to inform requesting and monitoring applications about the actual call state of the newly generated call. This process will occur shortly after the previously discussed messages.

Note that this particular message cannot be disabled. A LINE_CALLSTATE (unknown) message will be sent to monitoring applications only if a call to the lineCompleteTransfer() function causes telephone calls to be resolved into a three-way conference.

There are backward compatibility issues affecting this message. Older applications using earlier TAPI versions will not be expecting any particular value in the *dwParam2* parameter of a LINECALLSTATE_CONFERENCED message. In such cases, TAPI will pass the parent call the hConfCall value in *dwParam2* regardless of theTAPI version of an application receiving the message. In the case of a conference call that was initiated by the service provider, the older application will generally not be aware that the parent call has become a conference call. The exception to this—the case in which it might be aware—would occur if it were to spontaneously examine other information, such as by calling the lineGetConfRelatedCalls() function.

Table 9-10: Parameter values for the LINE_CALLSTATE message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	The callback instance supplied when opening the call's line
dwParam1	The new call state. This parameter must be one, and only one, of the LINECALLSTATE_ values shown in Table 9-11.
dwParam2	Call state-dependent information. If dwParam1 is LINECALLSTATE_BUSY, dwParam2 contains details about the busy mode. This parameter uses the LINEBUSYMODE_ constants shown in Table 9-12. If dwParam1 is LINECALLSTATE_CONNECTED, dwParam2 contains details about the connected mode using one of the following LINECONNECTEDMODE_ constants: LINECONNECTEDMODE_ACTIVE indicates that the call is connected at the current station (the current station is a participant in the call).

Parameter	Value(s)
dwParam2 (cont.)	LINECONNECTEDMODE_INACTIVE indicates that the call is active at one or more other stations, but the current station is not a participant in the call. If dwParam1 is LINECALLSTATE_DIALTONE dwParam2 contains the details about the dial tone mode using the following LINEDIALTONEMODE_ constants: LINEDIALTONEMODE_NORMAL indicates that this is a “normal” dial tone, which typically is a continuous tone. LINEDIALTONEMODE_SPECIAL indicates that this is a special dial tone indicating that a certain condition is currently in effect. LINEDIALTONEMODE_INTERNAL indicates that this is an internal dial tone, as within a PBX. LINEDIALTONEMODE_EXTERNAL indicates that this is an external (public network) dial tone. LINEDIALTONEMODE_UNKNOWN indicates that the dial tone mode is currently unknown but may become known later. LINEDIALTONEMODE_UNAVAIL indicates that the dial tone mode is unavailable and will not become known. If dwParam1 is LINECALLSTATE_OFFERING, dwParam2 contains details about the connected mode using the following LINEOFFERINGMODE_ constants: LINEOFFERINGMODE_ACTIVE indicates that the call is alerting at the current station (will be accompanied by LINEDEVSTATE_RINGING messages), and if any application is set up to automatically answer, it may do so. LINEOFFERINGMODE_INACTIVE indicates that the call is being offered at more than one station, but the current station is not alerting (for example, it may be an attendant station where the offering status is advisory, such as blinking a light). If dwParam1 is LINECALLSTATE_SPECIALINFO, dwParam2 contains the details about the special information mode using the following LINESPECIALINFO_ constants: LINESPECIALINFO_NOCIRCUIT indicates that this special information tone precedes a “no circuit” or emergency announcement (trunk blockage category). LINESPECIALINFO_CUSTIRREG indicates that this special information tone precedes one of the following: a vacant number, an Alarm Indication Signal (AIS), a Centrex number change with a nonworking station, an access code that was not dialed or dialed in error, or a manual intercept operator message (customer irregularity category). LINESPECIALINFO_REORDER indicates that this special information tone precedes a reorder announcement (equipment irregularity category). LINESPECIALINFO_UNKNOWN indicates that specifics about the special information tone are currently unknown but may become known later. LINESPECIALINFO_UNAVAIL indicates that specifics about the special information tone are unavailable and will not become known. If dwParam1 is LINECALLSTATE_DISCONNECTED, dwParam2 will contain details about the disconnect mode using the following LINEDISCONNECTMODE_ constants: LINEDISCONNECTMODE_NORMAL, which is a “normal” disconnect request by the remote party, indicates that the call was terminated normally. LINEDISCONNECTMODE_UNKNOWN indicates that the reason for the disconnect request is unknown. LINEDISCONNECTMODE_REJECT indicates that the remote user has rejected the call. LINEDISCONNECTMODE_PICKUP indicates that the call was picked up from elsewhere. LINEDISCONNECTMODE_FORWARDED indicates that the call was forwarded by the switch.

Parameter	Value(s)
dwParam2 (cont.)	LINEDISCONNECTMODE_BUSY indicates that the remote user's station is busy. LINEDISCONNECTMODE_NOANSWER indicates that the remote user's station does not answer. LINEDISCONNECTMODE_NODIALTONE indicates that a dial tone was not detected within a service provider-defined timeout, at a point during dialing when one was expected, such as at a "W" in the dialable string (note that this situation can also occur without a service provider-defined timeout period or a value specified in the dwWaitForDialTone member of the LINEDIALPARAMS structure). LINEDISCONNECTMODE_BADADDRESS indicates that the destination address in invalid. LINEDISCONNECTMODE_UNREACHABLE indicates that the remote user could not be reached. LINEDISCONNECTMODE_CONGESTION indicates that the network is congested. LINEDISCONNECTMODE_INCOMPATIBLE indicates that the remote user's station equipment is incompatible for the type of call requested. LINEDISCONNECTMODE_UNAVAIL indicates that the reason for the disconnect is unavailable and will not become known later. If dwParam1 is LINECALLSTATE_CONFERENCED, dwParam2 will contain the hConfCall of the parent call of the conference of which the subject hCall is a member. If the call specified in dwParam2 was not previously considered by an application to be a parent conference call (hConfCall), the application must make this change in status as a result of this message. If the application does not have a handle to the parent call of the conference (because it has previously called lineDeallocateCall() on that handle) dwParam2 will be set to NULL.
dwParam3	If zero, this parameter indicates that there has been no change in an application's privilege for the call. If nonzero, it specifies an application's privilege to the call. This will occur in the following situations: (1) The first time that an application is given a handle to this call. (2) When an application is the target of a call handoff (even if an application already was an owner of the call). This parameter uses the following LINECALL-PRIVILEGE_ constants: LINECALLPRIVILEGE_MONITOR indicates that an application has monitor privilege. LINECALLPRIVILEGE_OWNER indicates that an application has owner privilege.

Table 9-II: LINECALLSTATE_ constants used with the dwParamI field of the LINE_CALLSTATE message

Constant	Meaning
LINECALLSTATE_IDLE	This constant indicates that the call is idle—no call actually exists.
LINECALLSTATE_ OFFERING	This constant indicates that the call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the offering state does not automatically alert the user (alerting is done by the switch instructing the line to ring; it does not affect any call states).
LINECALLSTATE_ ACCEPTED	This constant indicates that the call was offered and has been accepted; this indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also indicates that alerting to both parties has started.
LINECALLSTATE_ DIALTONE	This constant indicates that the call is receiving a dial tone from the switch, which means that the switch is ready to receive a dialed number.
LINECALLSTATE_ DIALING	This constant indicates that destination address information (a phone number) is being sent to switch over the call (note that lineGenerateDigits() does not place the line into the dialing state).

Constant	Meaning
LINECALLSTATE_ RINGBACK	This constant indicates that the call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.
LINECALLSTATE_BUSY	This constant indicates that the call is receiving a busy tone. Busy tone indicates that the call cannot be completed; either a circuit (trunk) or the remote party's station are in use.
LINECALLSTATE_ SPECIALINFO	This constant indicates that special information will be sent by the network (such information is typically sent when the destination cannot be reached).
LINECALLSTATE_ CONNECTED	This constant indicates that the call has been established and the connection is made (information is able to flow over the call between the originating address and the destination address).
LINECALLSTATE_ PROCEEDING	This constant indicates that dialing has completed and the call is proceeding through the switch or telephone network.
LINECALLSTATE_ ONHOLD	This constant indicates that the call is on hold by the switch.
LINECALLSTATE_ CONFERENCED	This constant indicates that the call is currently a member of a multiparty conference call.
LINECALLSTATE_ ONHOLDPENDCONF	This constant indicates that the call is currently on hold while it is being added to a conference.
LINECALLSTATE_ON-HOLDPENDTRANSFER	(Not LINECALLSTATE_ONHOLDPENTRANSFER as indicated in one Microsoft Help File) This constant indicates that the call is currently on hold awaiting transfer to another number.
LINECALLSTATE_ DISCONNECTED	This constant indicates that the remote party has disconnected from the call.
LINECALLSTATE_ UNKNOWN	This constant indicates that the state of the call is not known (this may be due to limitations of the call progress detection implementation).

Table 9-12: LINEBUSYMODE_ constants used with the dwParam2 field of the LINE_CALLSTATE message

Constant	Meaning
LINEBUSYMODE_ STATION	This constant indicates that the busy signal indicates that the called party's station is busy (this is usually signaled by means of a "normal" busy tone).
LINEBUSYMODE_ TRUNK	This constant indicates that the busy signal indicates that a trunk or circuit is busy. This is usually signaled with a "long" busy tone.
LINEBUSYMODE_ UNKNOWN	This constant indicates that the busy signal's specific mode is currently unknown but may become known later.
LINEBUSYMODE_ UNAVAIL	This constant indicates that the busy signal's specific mode is unavailable and will not become known.

LINE_CLOSE Message

Windows will send a LINE_CLOSE message to an application when the specified line device has been forcibly closed. You might wonder why this would occur. One reason is to prevent a single application from monopolizing a line device for too long. Another might be in response to a user modifying the configuration of the line or its driver. In some cases, a service provider may forcibly close the line device despite the user's desire to immediately effect configuration changes. What about reopening a line after it has been so rudely closed? The ability to reopen a line immediately after it has been forcibly closed will depend on the specific telephony device. In any case, the line device handle or any call handles for calls on the line will no longer be valid once this message has been sent.

Table 9-13: Parameter values for the LINE_CLOSE message

Parameter	Value(s)
hDevice	A handle to the line device that was closed. This handle is no longer valid.
dwInstance	The callback instance supplied when opening the line
dwParam1	Not used
dwParam2	Not used
dwParam3	Not used

LINE_CREATE Message

Windows will send a LINE_CREATE message to an application to inform it that a new line device has been created. Applications supporting TAPI version 1.4 or above are candidates to receive the LINE_CREATE message. Older applications—those negotiating a TAPI version of 1.3 or less—will be sent a LINE_LINEDEVSTATE message specifying LINEDEVSTATE_REINIT. This situation requires these older applications to shut down their use of TAPI and call lineInitialize() again to obtain the new number of devices. Unlike previous versions of TAPI, however, newer versions do not require all applications to shut down before allowing applications to reinitialize. Reinitialization can take place immediately when a new device is created; however, complete shutdown is required when a service provider is removed from the system. This message informs an application of the existence of a new device along with its new device ID. An application can then decide if it wants to attempt to work with the new device. This message will be sent to all applications supporting this or subsequent versions of TAPI that have called either lineInitialize() or lineInitalizeEx(). This notification includes applications that do not have any line devices open at the time.

Table 9-14: Parameter values for the LINE_CREATE message

Parameter	Value(s)
hDevice	Not used
dwInstance	Not used
dwParam1	The dwDeviceID of the newly created device
dwParam2	Not used
dwParam3	Not used

LINE_DEVSPECIFIC Message

Windows will send a LINE_DEVSPECIFIC message to an application to notify it about device-specific events occurring on a line, address, or call. The specific meaning of the message and the interpretation of the parameters are device specific. This message is used by a service provider in conjunction with the lineDevSpecific() function.

Table 9-15: Parameter values for the LINE_DEVSPECIFIC message

Parameter	Value(s)
hDevice	A handle to either a line device or call. This is device specific.
dwInstance	The callback instance supplied when opening the line
dwParam1	Device specific
dwParam2	Device specific
dwParam3	Device specific

LINE_DEVSPECIFICFEATURE Message

Windows will send a LINE_DEVSPECIFICFEATURE message to an application to notify it about device-specific events occurring on a line, address, or call. The meaning of the message and the interpretation of the parameters are device specific. The LINE_DEVSPECIFICFEATURE message is used by a service provider in conjunction with the lineDevSpecificFeature() function.

Table 9-16: Parameter values for the LINE_DEVSPECIFICFEATURE message

Parameter	Value(s)
hDevice	A handle to either a line device or call. This is device specific.
dwInstance	The callback instance supplied when opening the line
dwParam1	Device specific
dwParam2	Device specific
dwParam3	Device specific

LINE_GATHERDIGITS Message

Windows will send a LINE_GATHERDIGITS message to an application when the current buffered digit-gathering request has terminated or been canceled. You can examine the digit buffer after the application has received this message. Note that this message will be sent only to the application that initiated the digit gathering on the telephony call using the lineGatherDigits() function.

If the lineGatherDigits() function is used to cancel a previous request to gather digits, Windows will send a LINE_GATHERDIGITS message to an application with *dwParam1* set to LINEGATHERTERM_CANCEL. This indicates that the originally specified buffer contains the digits gathered up to the point of cancellation. Because the time stamp specified by *dwParam3* may have been generated on a computer other than the one on which an application is running, you should use this only for comparison to other similarly time-stamped messages generated on the same line device. Those messages include LINE_GENERATE, LINE_MONITORDIGITS, LINE_MONITORMEDIA, and LINE_MONITORTONE.

With this information, you can ascertain the relative timing or separation between events. The TAPI Help file points out that the tick count can "wrap around" after approximately 49.7 days and recommends that you take this into account when performing calculations. If the service provider used does not generate the time stamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a time stamp at the point closest to the service provider generating the event so that the synthesized time stamp will be as accurate as possible.

Table 9-17: Parameter values for the LINE_GATHERDIGITS message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	The callback instance supplied when opening the line
dwParam1	Provides the reason why digit gathering was terminated. This parameter must be one and only one of the following LINEGATHERTERM_ constants: LINEGATHERTERM_BUFFERFULL indicates that the requested number of digits has been gathered (e.g., the buffer is full). LINEGATHERTERM_TERMDIGIT indicates that one of the termination digits matched a received digit (the matched termination digit is the last digit in the buffer). LINEGATHERTERM_FIRSTTIMEOUT indicates that the first digit timeout expired (the buffer contains no digits). LINEGATHERTERM_INTERTIMEOUT indicates that the inter-digit timeout expired. The buffer contains at least one digit. LINEGATHERTERM_CANCEL indicates that the request was canceled by this application, by another application, or because the call terminated.
dwParam2	Not used
dwParam3	The "tick count" (number of milliseconds since Windows started) at which the digit gathering completed. For TAPI versions prior to 2.0, this parameter is unused.

LINE_GENERATE Message

Windows will send a LINE_GENERATE message to an application to notify it that the current digit or tone generation has terminated. Be aware that only one such generation request can be in progress on a given call at any time. Windows will also send this message when either digit or tone generation is canceled. It will send the LINE_GENERATE message only to the application that requested the digit or tone generation.

Because the time stamp specified by *dwParam3* may have been generated on a computer other than the one on which the application is running, you should use this only for comparison to other similarly time-stamped messages generated on the same line device. Those messages include LINE_GATHERDIGITS, LINE_MONITORDIGITS, LINE_MONITORMEDIA, and LINE_MONITOR-TONE.

As with previous similar messages, with this information, you can ascertain the relative timing or separation between events. The TAPI Help file points out that the tick count can "wrap around' after approximately 49.7 days and recommends that you take this into account when performing calculations. If the service provider used does not generate the time stamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a time stamp at the point closest to the service provider generating the event so that the synthesized time stamp will be as accurate as possible.

Table 9-18: Parameter values for the LINE_GENERATE message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	The callback instance supplied when opening the line
dwParam1	This parameter provides the reason why digit or tone generation was terminated. This parameter must be one and only one of the following LINEGENERATETERM_ constants: LINEGENERATETERM_DONE indicates that the requested number of digits have been generated, or the requested tones have been generated for the requested duration. LINEGENERATETERM_CANCEL indicates that the digit or tone generation request was canceled by this application, by another application, or because the call terminated.
dwParam2	Not used
dwParam3	The "tick count" (number of milliseconds since Windows started) at which the digit or tone generation completed. For versions prior to TAPI 2.0, this parameter is unused.

LINE_LINEDEVSTATE Message

Windows will send a LINE_LINEDEVSTATE message to an application when the state of a line device has changed. You can call the lineGetLineDevStatus() function to determine the new status of the line. Applications can use this function to query a line device about its current line status. Note that the status information returned applies globally to all addresses on the line device. Similarly, use lineGetAddressStatus() to determine the status information about a specific address on a line. You can control the process of sending the LINE_LINEDEVSTATE message by using the lineSetStatusMessages() function. Using this function, an application can specify the status item changes about which it wants to be notified. By default, all status reporting will be disabled with the exception of LINEDEVSTATE_REINIT, which can never be disabled. This message will be 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 valid combinations of these.

Table 9-19: Parameter values for the LINE_LINEDEVSTATE message

Parameter	Value(s)
hDevice	A handle to the line device. This parameter is NULL when dwParam1 is LINEDEVSTATE_REINIT.
dwInstance	The callback instance supplied when opening the line. If the dwParam1 parameter is LINEDEVSTATE_REINIT, the dwCallbackInstance parameter is not valid and is set to zero.
dwParam1	This parameter indicates that the line device status item has changed. The parameter can be a combination of the LINEDEVSTATE_ constants explained in Table 9-20.
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 of one to dwNumRingModes, where dwNumRingModes is a line device capability. If dwParam1 is LINEDEVSTATE_REINIT, and the message was issued by TAPI as a result of translation of a newTAPI message into a REINIT message, then dwParam2 will contain the dwMsg parameter of the original message (for example, LINE_CREATE or LINE_LINEDEV-STATE). If dwParam2 is zero, the REINIT message is a "real" REINIT message that requires an application to call the lineShutdown() function at its earliest convenience.
dwParam3	The interpretation of this parameter depends on the value of dwParam1 and, in some cases, also on the value of dwParam2. If dwParam1 is LINEDEVSTATE_RINGING, dwParam3 will contain the ring count for this ring event, with those counts starting at zero. If dwParam1 is LINEDEVSTATE_REINIT and TAPI issues a message responding to the translation of a new API message into a REINIT error message, dwParam3 will contain the dwParam1 parameter of the original message depending on the value of dwParam2. In this case, if dwParam2 is LINE_LINEDEVSTATE, dwParam3 will be LINEDEVSTATE_TRANSLATECHANGE or some other LINEDEVSTATE_ value; if dwParam2 is LINE_CREATE, dwParam3 will contain the new device ID.

Table 9-20: LINEDEVSTATE_ constants used with the dwParam1 field of the LINE_LINEDEVSTATE message

Constant	Meaning
LINEDEVSTATE_ OTHER	This constant indicates that device status items other than those listed below have changed (in this case, you should check the current device status to determine which items have changed).
LINEDEVSTATE_ RINGING	This constant indicates that the switch tells the line to alert the user (service providers notify applications on each ring cycle by sending messages containing this constant).
LINEDEVSTATE_ CONNECTED	This constant indicates that the line was previously disconnected but is now connected to TAPI again.
LINEDEVSTATE_ DISCONNECTED	This constant indicates that this line was previously connected and is now disconnected from TAPI.
LINEDEVSTATE_ MSGWAITON	This constant indicates that the "message waiting" indicator is turned on.
LINEDEVSTATE_ MSGWAITOFF	This constant indicates that the "message waiting" indicator is turned off.
LINEDEVSTATE_ NUMCOMPLETIONS	This constant indicates that the number of outstanding call completions on the line device has changed.
LINEDEVSTATE_ INSERVICE	This constant indicates that the line is connected to TAPI, a situation that occurs when TAPI is first activated or when the line wire is physically plugged in and in service at the switch while TAPI is active.

Constant	Meaning
LINEDEVSTATE_ OUTOFSERVICE	This constant indicates that the line is out of service at the switch or physically disconnected, resulting in TAPI not being able to operate on the line device.
LINEDEVSTATE_ MAINTENANCE	This constant indicates that maintenance is being performed on the line at the switch resulting in TAPI not being able to operate on the line device.
LINEDEVSTATE_ OPEN	This constant indicates that the line has been opened by another application.
LINEDEVSTATE_ CLOSE	This constant indicates that the line has been closed by another application.
LINEDEVSTATE_ NUMCALLS	This constant indicates that the number of calls on the line device has changed.
LINEDEVSTATE_ TERMINALS	This constant indicates that the terminal settings have changed.
LINEDEVSTATE_ ROAMMODE	This constant indicates that the roaming state of the line device has changed.
LINEDEVSTATE_ BATTERY	This constant indicates that the battery level of a cellular phone has changed significantly.
LINEDEVSTATE_ SIGNAL	This constant indicates that the signal level of a cellular phone has changed significantly.
LINEDEVSTATE_ DEVSPECIFIC	This constant indicates that the line's device-specific information has changed.
LINEDEVSTATE_ REINIT	This constant indicates that items have changed in the configuration of line devices (in order to discover these changes, an application should reinitialize its use of TAPI and set the dwDevice parameter to NULL for this state change, as it applies to any of the lines in the system).
LINEDEVSTATE_ LOCK	This constant indicates that the locked status of the line device has changed (see the description of the LINEDEVSTATUSFLAGS_LOCKED bit of the LINEDEVSTATUS-FLAGS_ constants).
LINEDEVSTATE_ CAPSCHANGE	This constant indicates that one or more of the fields in the LINEDEVCAPS structure for the address have changed due to configuration changes made by the user or other circumstances (an application should use lineGetDevCaps() to read the updated structure).
LINEDEVSTATE_ CONFIGCHANGE	This constant indicates that configuration changes have been made to one or more of the media devices associated with the line device (an application may call lineGetDevConfig() to read the updated information).
LINEDEVSTATE_ TRANSLATECHANGE	This constant indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the LINETRANSLATECAPS structure have changed (you should call lineGetTranslateCaps() to read the updated structure).
LINEDEVSTATE_ COMPLCANCEL	This constant indicates that the call completion identified by the completion ID contained in parameter dwParam2 of the LINE_LINEDEVSTATE message has been externally canceled and is no longer considered valid (if that value were to be passed in a subsequent call to lineUncompleteCall(), the function would fail with LINEERR_INVALCOMPLETIONID).

Constant	Meaning
LINEDEVSTATE_REMOVED	This constant indicates that the device is being removed from the system by the service provider (most likely through user action, a control panel, or similar utility). A LINE_LINEDEVSTATE message with this value will normally be immediately followed by a LINE_CLOSE message on the device. Subsequent attempts to access the device prior to TAPI being reinitialized will result in LINEERR_NODEVICE being returned to an application. If a service provider sends a LINE_LINEDEVSTATE message containing this value to TAPI, TAPI will pass it along to applications that have negotiated TAPI version 1.4 or above; applications negotiating a previous TAPI version will not receive any notification.

LINE_MONITORDIGITS Message

Windows will send a LINE_MONITORDIGITS message to an application when a digit has been detected. The lineMonitorDigits() function controls the process of sending this message. To be a candidate to receive this message, an application must have enabled digit monitoring.

Because the time stamp specified by *dwParam3* may have been generated on a computer other than the one on which an application is running, you should use this only for comparison to other similarly time-stamped messages generated on the same line device. Those messages include LINE_GATHERDIGITS, LINE_GENERATE, LINE_MONITORMEDIA, and LINE_MONITORTONE.

As with previous similar messages, with this information you can ascertain the relative timing or separation between events. The TAPI Help file points out that the tick count can "wrap around" after approximately 49.7 days and recommends that you take this into account when performing calculations. If the service provider used does not generate the time stamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a time stamp at the point closest to the service provider generating the event so that the synthesized time stamp will be as accurate as possible.

Table 9-21: Parameter values for the LINE_MONITORDIGITS message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	The callback instance supplied when opening the call's line
dwParam1	The low-order byte contains the last digit received in ASCII.
dwParam2	The digit mode that was detected. This parameter must be one and only one of the following LINEDIGITMODE_ constants: LINEDIGITMODE_PULSE directs TAPI to detect digits as audible clicks that are the result of rotary pulse sequences (valid digits for pulse are 0 through 9). LINEDIGITMODE_DTMF directs TAPI to detect digits as DTMF tones (valid digits for DTMF are 0 through 9, A, B, C, D, *, and #). LINEDIGITMODE_DTMFEND directs TAPI to detect and provide application notification of DTMF down edges (valid digits for DTMF are 0 through 9, A, B, C, D, *, and #).
dwParam3	The "tick count" (number of milliseconds since Windows started) at which the specified digit was detected. For TAPI versions prior to version 2.0, this parameter is unused.

LINE_MONITORMEDIA Message

Windows will send a LINE_MONITORMEDIA message to an application when a change in the call's media mode has been detected. The lineMonitorMedia() function controls the process of sending this message. To be a candidate to receive this message, an application must have enabled media monitoring

Because the time stamp specified by *dwParam3* may have been generated on a computer other than the one on which an application is running, you should use this only for comparison to other similarly time-stamped messages generated on the same line device. Those messages include LINE_GATHERDIGITS, LINE_MONITORDIGITS, LINE_GENERATE, and LINE_MONITORTONE.

As with previous similar messages, with this information you can ascertain the relative timing or separation between events. The TAPI Help file points out that the tick count can "wrap around" after approximately 49.7 days and recommends that you take this into account when performing calculations. If the service provider used does not generate the time stamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a time stamp at the point closest to the service provider generating the event so that the synthesized time stamp will be as accurate as possible.

Table 9-22: Parameter values for the LINE_MONITORMEDIA message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	The callback instance supplied when opening the line
dwParam1	The new media mode. This parameter must be one and only one of the following LINEMEDIAMODE_ constants: LINEMEDIAMODE_INTERACTIVEVOICE indicates the presence of voice energy has been detected and the call is treated as an interactive call with humans on both ends. LINEMEDIAMODE_AUTOMATEDVOICE indicates the presence of voice energy has been detected and the call is locally handled by an automated application. LINEMEDIAMODE_DATAMODEM indicates a data modem session has been detected. LINEMEDIAMODE_G3FAX indicates a group 3 fax has been detected. LINEMEDIAMODE_TDD indicates a TDD (Telephony Devices for the Deaf) session has been detected. LINEMEDIAMODE_G4FAX indicates a group 4 fax has been detected. LINEMEDIAMODE_DIGITALDATA indicates digital data has been detected. LINEMEDIAMODE_TELETEX indicates a teletex session has been detected. Teletex is one of the telematic services. LINEMEDIAMODE_VIDEOTEX indicates a videotex session has been detected. Videotex is one the telematic services. LINEMEDIAMODE_TELEX indicates a telex session has been detected. Telex is one the telematic services. LINEMEDIAMODE_MIXED indicates a mixed session has been detected. Mixed is one of the telematic services. LINEMEDIAMODE_ADSI indicates an ADSI (Analog Display Services Interface) session has been detected. LINEMEDIAMODE_VOICEVIEW indicates the media mode of the call is VoiceView.
dwParam2	Not used

Parameter	Value(s)
dwParam3	The "tick count" (number of milliseconds since Windows started) at which the specified media was detected. For TAPI versions prior to 2.0, this parameter is unused.

LINE_MONITORTONE Message

Windows will send a LINE_MONITORTONE message to an application when a tone is detected. The lineMonitorTones() function controls the process of sending this message. Because the time stamp specified by *dwParam3* may have been generated on a computer other than the one on which an application is running, you should use this only for comparison to other similarly time-stamped messages generated on the same line device. Those messages include LINE_GATHERDIGITS, LINE_MONITORDIGITS, LINE_MONITORMEDIA, and LINE_GENERATE.

As with previous similar messages, with this information you can ascertain the relative timing or separation between events. The TAPI Help file points out that the tick count can "wrap around" after approximately 49.7 days and recommends that you take this into account when performing calculations. If the service provider used does not generate the time stamp (for example, if it was created using an earlier version of TAPI), then TAPI will provide a time stamp at the point closest to the service provider generating the event so that the synthesized time stamp will be as accurate as possible.

Table 9-23: Parameter values for the LINE_MONITORTONE message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	The callback instance supplied when opening the call's line
dwParam1	An application-specific dwAppSpecific field of the LINEMONITORTONE structure for the tone that was detected
dwParam2	Not used
dwParam3	The "tick count" (number of milliseconds since Windows started) at which the specified media was detected. For TAPI versions prior to 2.0, this parameter is unused.

LINE_PROXYREQUEST Message

Windows uses the LINE_PROXYREQUEST message to send a request to a registered proxy function handler. This message will be sent only to the first application that registered to handle proxy requests of the type being delivered. An application should process the request contained in the proxy buffer and call lineProxyResponse() to return data or deliver results. Processing of the request should be done within the context of an application's TAPI callback function, only if it can be performed immediately without waiting for response from any other entity.

If an application needs to communicate with other entities, the request should be queued within an application and you should exit the callback function. This approach will ensure that there is no delay in an application's receipt of further TAPI messages. The Microsoft Help file provides examples that might result in blocking of messages.

When the LINE_PROXYREQUEST is sent to the proxy handler, TAPI has already returned a positive *dwRequestID* function result to the original application and has unblocked the calling thread to continue execution. At that point, an application is awaiting a LINE_REPLY message, which is automatically generated when the proxy handler application calls lineProxyResponse().

Do not free the memory pointed to by lpProxyRequest; TAPI will take care of that during the execution of the lineProxyResponse() function. You must call lineProxyResponse() once and only once for each LINE_PROXYREQUEST message.

If an application receives a LINE_CLOSE message while it has pending proxy requests, it should call lineProxyResponse() for each pending request, passing in an appropriate *dwResult* value (such as LINEERR_OPERATIONFAILED).

Table 9-24: Parameter values for the LINE_PROXYREQUEST message

Parameter	Value(s)
hDevice	An application's handle to the line device on which the agent status has changed
dwInstance	The callback instance supplied when opening the call's line
dwParam1	Pointer to a LINEPROXYREQUEST structure containing the request to be processed by the proxy handler application
dwParam2	Reserved
dwParam3	Reserved

LINE_REMOVE Message

Windows will send a LINE_REMOVE message to an application to inform it of the removal (deletion from the system) of a line device. Generally, this is not used for temporary removals, such as extraction of PCMCIA devices. Rather, it is used only in the case of permanent removals in which the service provider would no longer report the device when TAPI was reinitialized.

As in the case of other messages, there are backward compatibility issues. Applications supporting TAPI version 2.0 or above will be sent a LINE_REMOVE message informing them that the device has been removed from the system. The LINE_REMOVE message will have been preceded by a LINE_CLOSE message on each line handle if an application had the line open. This message will be sent to all applications supporting TAPI version 2.0 or above that have called lineInitializeEx(), including those that do not have any line devices open at the time.

Older applications will be sent a LINE_LINEDEVSTATE message specifying LINEDEVSTATE_REMOVED, followed by a LINE_CLOSE message. Unlike the LINE_REMOVE message, however, these older applications can receive these messages only if they have the line open when it is removed. If they do not have the line open, their only indication that the device was removed would be receiving a LINEERR_NODEVICE error when attempting to access the device.

Following the removal of a device, any attempt to access it by its device ID will result in a LINEERR_NODEVICE error. After all TAPI applications have been shut down and TAPI has been reinitialized, the removed device will no longer occupy a device ID. After a LINE_REMOVE message is received from a service provider, no further calls will be made to that service provider using that line device ID.

Table 9-25: Parameter values for the LINE_REMOVE message

Parameter	Value(s)
hDevice	Reserved; set to 0
dwInstance	Reserved; set to 0
dwParam1	Identifier of the line device that was removed
dwParam2	Reserved; set to 0
dwParam3	Reserved; set to 0

LINE_REPLY Message

Windows will send a LINE_REPLY message to an application to report the results of function calls that completed asynchronously. Functions that operate asynchronously will return a positive request ID value to an application along with a reply message to identify the request that was completed. The other parameter for the LINE_REPLY indicates success or failure. Possible errors are the same as those defined by the corresponding function. This message cannot be disabled. In some cases, an application may fail to receive the LINE_REPLY message corresponding to a call to an asynchronous function. This occurs if the corresponding call handle is deallocated before the message has been received.

Table 9-26: Parameter values for the LINE_REPLY message

Parameter	Value(s)
hDevice	Not used
dwInstance	Returns an application's callback instance
dwParam1	The request ID for which this is the reply
dwParam2	The success or error indication. You should cast this parameter into a LONG. Zero indicates success; a negative number indicates an error.
dwParam3	Not used

LINE_REQUEST Message

As with the previous message, Windows will send a LINE_REQUEST message to an application to report the results of function calls that completed asynchronously. This message will be sent to the highest priority application that has registered for the corresponding request mode. This message indicates the arrival of an Assisted Telephony request (see Chapter 8) of the specified request mode. If *dwParam1* is LINEREQUESTMODE_MAKECALL or LINEREQUESTMODE_MEDIACALL, an application can call the lineGet-Request() function (using the corresponding request mode) to receive the request. If *dwParam1* is LINEREQUESTMODE_DROP, the message will contain all of the information the request recipient needs in order to perform the request.

Table 9-27: Parameter values for the LINE_REQUEST message

Parameter	Value(s)
hDevice	Not used
dwInstance	The registration instance of an application specified in lineRegisterRequest-Recipient()
dwParam1	The request mode of the newly pending request. This parameter uses one of the following LINEREQUESTMODE_ constants: LINEREQUESTMODE_MAKECALL indicates a tapiRequestMakeCall request. LINEREQUESTMODE_DROP indicates to drop the call. LINEREQUESTMODE_MEDIACALL indicates to make a media call.
dwParam2	If dwParam1 is set to LINEREQUESTMODE_DROP, dwParam2 will contain the hWnd of an 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 an application requesting the drop. Otherwise, dwParam3 is unused.

LINE_AGENTSESSIONSTATUS Message

Windows will send a LINE_AGENTSESSIONSTATUS message when the status of an ACD agent session changes on an agent handler for which an application currently has an open line. This message is generated using the lineProxyMessage() function.

Table 9-28: Parameter values for the LINE_AGENTSESSIONSTATUS message

Parameter	Value(s)
hDevice	An application's handle to the line device on which the agent session status has changed
dwInstance	The callback instance supplied when opening the line
dwParam1	A handle of the agent session whose status has changed
dwParam2	Specifies the agent session status that has changed; can be one or more of the LINE_AGENTSESSIONSTATUS constants
dwParam3	If dwParam2 includes the LINEAGENTSTATUSEX_STATE bit, dwParam3 indicates the new value of the agent state, which is one of the LINEAGENTSTATEEX_ constants. Otherwise, dwParam3 is set to zero.

LINE_QUEUESTATUS Message

Windows will send a LINE_QUEUESTATUS message when the status of an ACD queue changes on an agent handler for which an application currently has an open line. This message is generated using the lineProxyMessage() function.

Table 9-29: Parameter values for the LINE_QUEUESTATUS message

Parameter	Value(s)
hDevice	An application's handle to the line device. This relates to the agent handler.
dwInstance	The callback instance supplied when opening the line
dwParam1	The identifier of the queue whose status has changed
dwParam2	Specifies the queue status that has changed; can be one or more of the LINEQUEUESTATUS_ constants
dwParam3	Reserved; set to zero

LINE_AGENTSTATUSEX Message

Windows will send a LINE_AGENTSTATUSEX message when the status of an ACD agent changes on an agent handler for which an application currently has an open line. This message is generated using the lineProxyMessage() function.

Table 9-30: Parameter values for the LINE_AGENTSTATUSEX message

Parameter	Value(s)
hDevice	An application's handle to the line device. This relates to the agent handler.
dwInstance	The callback instance supplied when opening the line
dwParam1	The handle of the agent whose status has changed
dwParam2	Specifies the queue status that has changed; can be one or more of the LINEQUEUESTATUS_ constants
dwParam3	If dwParam2 includes the LINEAGENTSTATUSEX _STATE bit, the dwParam3 field will indicate the new value of the agent state, which will be one of the LINEAGENTSTATEEX_ constants. Otherwise, dwParam3 will be set to zero.

LINE_GROUPSTATUS Message

Windows will send a LINE_GROUPSTATUS message when the status of an ACD group changes on an agent handler for which an application currently has an open line. This message is generated using the lineProxyMessage() function.

Table 9-31: Parameter values for the LINE_GROUPSTATUS message

Parameter	Value(s)
hDevice	An application's handle to the line device. This relates to the agent handler.
dwInstance	The callback instance supplied when opening the line
dwParam1	Reserved; set to zero
dwParam2	Specifies the group status that has changed. An application can invoke lineGetGroupList() to determine the changes in available groups. The dwParam2 parameter is one or more of the LINEGROUPSTATUS_ constants.
dwParam3	Reserved; set to zero

LINE_PROXYSTATUS Message

Windows will send a LINE_PROXYSTATUS (listed incorrectly as "LINE_QUEUESTATUS" in MS Help) message when the available proxies change on a line that an application currently has open. TAPISRV generates this message during a lineOpen() function using LINEPROXYSTATUS_OPEN and LINEPROXYSTATUS_ALLOPENFORACD or a lineClose() function using LINEPROXYSTATUS_CLOSE (all LINEPROXYSTATUS_ constants).

Table 9-32: Parameter values for the LINE_PROXYSTATUS message

Parameter	Value(s)
hDevice	An application's handle to the line device. This relates to the agent handler.
dwInstance	The callback instance supplied when opening the line
dwParam1	Specifies the queue status that has changed; can be one or more of the LINEPROXYSTATUS_ constants
dwParam2	If dwParam1 is set to LINEPROXYSTATUS_OPEN or LINEPROXYSTATUS_CLOSE, dwParam2 indicates the related proxy request type, which is one of the following: LINEPROXYREQUEST_SETAGENTGROUP, LINEPROXYREQUEST_SETAGENTSTATE, LINEPROXYREQUEST_SETAGENTACTIVITY, LINEPROXYREQUEST_GETAGENTCAPS, LINEPROXYREQUEST_GETAGENTSTATUS, LINEPROXYREQUEST_AGENTSPECIFIC, LINEPROXYREQUEST_GETAGENTACTIVITYLIST, LINEPROXYREQUEST_GETAGENTGROUPLIST, LINEPROXYREQUEST_CREATEAGENT, LINEPROXYREQUEST_SETAGENTMEASUREMENTPERIOD, LINEPROXYREQUEST_GETAGENTINFO, LINEPROXYREQUEST_CREATEAGENTSESSION, LINEPROXYREQUEST_GETAGENTSESSIONLIST, LINEPROXYREQUEST_SETAGENTSESSIONSTATE, LINEPROXYREQUEST_ GETAGENTSESSIONINFO, LINEPROXYREQUEST_GETQUEUELIST, LINEPROXYREQUEST_SETQUEUEMEASUREMENTPERIOD, LINEPROXYREQUEST_GETQUEUEINFO, LINEPROXYREQUEST_GETGROUPLIST, or LINEPROXYREQUEST_SETAGENTSTATEEX; otherwise, dwParam2 is set to zero.

Parameter	Value(s)
dwParam3	Reserved; set to zero

LINE_APPNEWCALLHUB Message

Windows will send a LINE_APPNEWCALLHUB message to inform an application when a new call hub has been created.

Table 9-33: Parameter values for the LINE_APPNEWCALLHUB message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	The callback instance supplied when opening the call's line
dwParam1	The tracking level on the new hub, as defined by one of the LINECALLHUBTRACKING_ constants
dwParam2	Reserved; should be set to 0
dwParam3	Not used; should be set to 0

LINE_CALLHUBCLOSE Message

Windows will send a LINE_CALLHUBCLOSE message when a call hub has been closed. Since this message originates with TAPI and not with a service provider, there is no corresponding TSPI message.

Table 9-34: Parameter values for the LINE_CALLHUBCLOSE message

Parameter	Value(s)
hDevice	A handle to the call
dwInstance	The callback instance supplied when opening the call's line
dwParam1	Reserved; set to 0
dwParam2	Reserved; set to 0
dwParam3	Reserved; set to 0

LINE_DEVSPECIFICEX Message

Windows will send a LINE_DEVSPECIFICEX message to notify an application about device-specific events occurring on a line, address, or call. The meaning of the message and the interpretation of the parameters are device specific. The LINE_DEVSPECIFICEX message is used by a service provider in conjunction with the lineDevSpecific() function. Its meaning is device specific.

Table 9-35: Parameter values for the LINE_DEVSPECIFICEX message

Parameter	Value(s)
hDevice	A handle to either a line device or call. This parameter is device specific.
dwInstance	The callback instance supplied when opening the line
dwParam1	Device specific

Parameter	Value(s)
dwParam2	Device specific
dwParam3	Device specific

LINEPROXYREQUEST_ Constants

The LINEPROXYREQUEST_ constants are defined as follows in TAPI.PAS:

```
LINEPROXYREQUEST_SETAGENTGROUP                 = $00000001;  // TAPI v2.0
LINEPROXYREQUEST_SETAGENTSTATE                 = $00000002;  // TAPI v2.0
LINEPROXYREQUEST_SETAGENTACTIVITY              = $00000003;  // TAPI v2.0
LINEPROXYREQUEST_GETAGENTCAPS                  = $00000004;  // TAPI v2.0
LINEPROXYREQUEST_GETAGENTSTATUS                = $00000005;  // TAPI v2.0
LINEPROXYREQUEST_AGENTSPECIFIC                 = $00000006;  // TAPI v2.0
LINEPROXYREQUEST_GETAGENTACTIVITYLIST          = $00000007;  // TAPI v2.0
LINEPROXYREQUEST_GETAGENTGROUPLIST             = $00000008;  // TAPI v2.0
LINEPROXYREQUEST_CREATEAGENT                   = $00000009;  // TAPI v2.2
LINEPROXYREQUEST_SETAGENTMEASUREMENTPERIOD     = $0000000A;  // TAPI v2.2
LINEPROXYREQUEST_GETAGENTINFO                  = $0000000B;  // TAPI v2.2
LINEPROXYREQUEST_CREATEAGENTSESSION            = $0000000C;  // TAPI v2.2
LINEPROXYREQUEST_GETAGENTSESSIONLIST           = $0000000D;  // TAPI v2.2
LINEPROXYREQUEST_SETAGENTSESSIONSTATE          = $0000000E;  // TAPI v2.2
LINEPROXYREQUEST_GETAGENTSESSIONINFO           = $0000000F;  // TAPI v2.2
LINEPROXYREQUEST_GETQUEUELIST                  = $00000010;  // TAPI v2.2
LINEPROXYREQUEST_SETQUEUEMEASUREMENTPERIOD     = $00000011;  // TAPI v2.2
LINEPROXYREQUEST_GETQUEUEINFO                  = $00000012;  // TAPI v2.2
LINEPROXYREQUEST_GETGROUPLIST                  = $00000013;  // TAPI v2.2
LINEPROXYREQUEST_SETAGENTSTATEEX               = $00000014;  // TAPI v2.2
```

These constants are used in TAPI version 2.0 and later, and they occur in two contexts. First, they can be used in an array of DWORD values in the LINECALLPARAMS structure passed in with lineOpen() when the LINEOPENOPTION_PROXY option is specified to indicate which functions an application is willing to handle; second, they can be used in the LINEPROXYREQUEST structure passed to the handler application by a LINEPROXYREQUEST_ message to indicate the type of request that is to be processed and the format of the data in the buffer. Table 9-36 shows each of these constants and the function with which it is associated.

Table 9-36: LINEPROXYREQUEST_ constants

Constant	Associated Function
LINEPROXYREQUEST_AGENTSPECIFIC	lineAgentSpecific()
LINEPROXYREQUEST_CREATEAGENT	lineCreateAgent()
LINEPROXYREQUEST_CREATEAGENTSESSION	lineCreateAgentSession()
LINEPROXYREQUEST_GETAGENTACTIVITYLIST	lineGetAgentActivityList()
LINEPROXYREQUEST_GETAGENTCAPS	lineGetAgentCaps()
LINEPROXYREQUEST_GETAGENTGROUPLIST	lineGetAgentGroupList()
LINEPROXYREQUEST_GETAGENTINFO	lineGetAgentInfo()
LINEPROXYREQUEST_GETAGENTSESSIONINFO	lineGetAgentSessionInfo()
LINEPROXYREQUEST_GETAGENTSESSIONLIST	lineGetAgentSessionList()
LINEPROXYREQUEST_GETAGENTSTATUS	lineGetAgentStatus()
LINEPROXYREQUEST_GETGROUPLIST	lineGetGroupList()
LINEPROXYREQUEST_GETQUEUEINFO	lineGetQueueInfo()
LINEPROXYREQUEST_GETQUEUELIST	lineGetQueueList()
LINEPROXYREQUEST_SETAGENTACTIVITY	lineSetAgentActivity()
LINEPROXYREQUEST_SETAGENTGROUP	lineSetAgentGroup()

Functions Related to Message Handling

There are several functions associated with messages. Let's take a look at them and their related structures.

function lineGetMessage TAPI.pas

Syntax

function lineGetMessage(hLineApp: HLINEAPP; var lpMessage: TLineMessage; dwTimeout: DWORD): Longint; stdcall; // TAPI v2.0

Description

This function returns the next TAPI message that is queued for delivery to an application that is using the Event Handle notification mechanism (see lineInitializeEx() for further details).

Parameters

hLineApp: The handle (HLINEAPP) returned by lineInitializeEx(). The application must have set the LINEINITIALIZEEXOPTION_USEEVENT option in the *dwOptions* member of the LINEINITIALIZEEXPARAMS structure.

var lpMessage: A pointer (TLineMessage) to a LINEMESSAGE structure. Upon successful return from this function, the structure will contain the next message that had been queued for delivery to the application.

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

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALAPPHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, and LINEERR_NOMEM.

See Also

lineInitializeEx, LINEINITIALIZEEXPARAMS, LINEMESSAGE, lineShutdown

Example

None

structure LINEINITIALIZEEXPARAMS TAPI.pas

The LINEINITIZALIZEEXPARAMS structure describes parameters supplied when making calls using LINEINITIALIZEEX. It is defined as follows in TAPI.pas:

```
PLineInitializeExParams = ^TLineInitializeExParams;
 lineinitializeexparams_tag = packed record
   dwTotalSize,                                      // TAPI v2.0
   dwNeededSize,                                     // TAPI v2.0
   dwUsedSize,                                       // TAPI v2.0
   dwOptions: DWORD;                                 // TAPI v2.0
   Handles: TTAPIHandleUnion;
   dwCompletionKey: DWORD;                           // TAPI v2.0
 end;
 TLineInitializeExParams = lineinitializeexparams_tag;
 LINEINITIALIZEEXPARAMS = lineinitializeexparams_tag;
```

The fields of this structure are described in Table 9-37.

Table 9-37: Fields of the LINEINITIALIZEEXPARAMS structure

Field	Meaning
dwTotalSize	The total size, in bytes, allocated to this data structure
dwNeededSize	The size, in bytes, for this data structure that is needed to hold all the returned information
dwUsedSize	The size, in bytes, of the portion of this data structure that contains useful information
dwOptions	One of the LINEINITIALIZEEXOPTION_ constants (see Table 9-38) that specifies the event notification mechanism that the application desires to use
handles	If dwOptions specifies LINEINITIALIZEEXOPTION_USEEVENT, TAPI returns the event handle in this field.

Field	Meaning
dwCompletionKey	If dwOptions specifies LINEINITIALIZEEXOPTION_USECOMPLETION-PORT, the application must specify in this field the handle of an existing completion port opened using CreateIoCompletionPort(). If dwOptions specifies LINEINITIALIZEEXOPTION_USECOMPLETIONPORT, the application must specify in this field a value that is returned through the lpCompletionKey parameter of GetQueuedCompletionStatus() to identify the completion message as a telephony message.

LINEINITIALIZEEXOPTION_ Constants

The LINEINITIALIZEEXOPTION_ constants specify which event notification mechanism to use when initializing a session. They are described in Table 9-38.

Table 9-38: LINEINITIALIZEEXOPTION_ constants

Constant	Meaning
LINEINITIALIZEEXOPTION_ CALLHUBTRACKING	The application desires to use the call hub tracking event notification mechanism. This constant is exposed only to applications that negotiate a TAPI version of 3.0 or higher.
LINEINITIALIZEEXOPTION_ USECOMPLETIONPORT	The application desires to use the Completion Port event notification mechanism. This flag is exposed only to applications that negotiate a TAPI version of 2.0 or higher.
LINEINITIALIZEEXOPTION_ USEEVENT	The application desires to use the Event Handle event notification mechanism. This flag is exposed only to applications that negotiate a TAPI version of 2.0 or higher.
LINEINITIALIZEEXOPTION_ USEHIDDENWINDOW	The application desires to use the Hidden Window event notification mechanism. This flag is exposed only to applications that negotiate a TAPI version of 2.0 or higher.

structure LINEMESSAGE TAPI.pas

The LINEMESSAGE structure contains parameter values specifying a change in status of the line that the application currently has open. The lineGet-Message() function returns the LINEMESSAGE structure. (For information about parameter values passed in this structure, see Line Device Messages in the TAPI Help file.) It is defined as follows in TAPI.pas:

```
PLineMessage = ^TLineMessage;
linemessage_tag = packed record
  hDevice,                                    // TAPI v2.0
  dwMessageID,                                // TAPI v2.0
  dwCallbackInstance,                         // TAPI v2.0
  dwParam1,                                   // TAPI v2.0
  dwParam2,                                   // TAPI v2.0
  dwParam3: DWORD;                            // TAPI v2.0
end;
TLineMessage = linemessage_tag;
LINEMESSAGE = linemessage_tag;
```

The fields of the LINEMESSAGE structure are described in Table 9-39.

Table 9-39: Fields of the LINEMESSAGE structure

Field	Meaning
hDevice	A handle to either a line device or a call. The nature of this handle (line handle or call handle) can be determined by the context provided by dwMessageID.
dwMessageID	A line or call device message
dwCallbackInstance	Instance data passed back to the application, which was specified by the application in the dwCallBackInstance parameter of lineInitializeEx(). This DWORD is not interpreted by TAPI.
dwParam1	A parameter for the message
dwParam2	A parameter for the message
dwParam3	A parameter for the message

function lineGetStatusMessages TAPI.pas

Syntax

```
function lineGetStatusMessages(hLine: HLINE; var dwLineStates, dwAddressStates:
DWORD): Longint; stdcall;
```

Description

This function enables an application to query which notification messages the application is set up to receive for events related to status changes for the specified line or any of its addresses.

Parameters

hLine: A handle (HLINE) to the line device

var dwLineStates: A DWORD holding a bit array that identifies for which line device status changes a message is to be sent to the application. If a flag is TRUE, that message is enabled; if FALSE, it is disabled. Note that multiple flags can be set. This parameter uses the LINEDEVSTATE_ constants shown in Table 9-20.

dwAddressStates: A DWORD holding 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, it is disabled. Multiple flags can be set. This parameter uses the LINEADDRESSSTATE_ constants shown in Table 8-2.

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALLINEHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALPOINTER, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, and LINEERR_UNINITIALIZED.

See Also

LINE_CLOSE, LINE_LINEDEVSTATE, lineSetStatusMessages

Example

Listing 9-1 shows how your application can query which notification messages it is set up to receive for line or line address status events.

Listing 9-1: Querying which notification messages an application is set up to receive

```
function TTapiInterface.GetStatusMessages: boolean;
begin
  fLineStates :=  LINEDEVSTATE_OTHER or LINEDEVSTATE_RINGING or
  LINEDEVSTATE_CONNECTED or LINEDEVSTATE_NUMCOMPLETIONS or
  LINEDEVSTATE_DISCONNECTED;
  fAddressStates := LINEADDRESSSTATE_DEVSPECIFIC  or
  LINEADDRESSSTATE_OTHER  or LINEADDRESSSTATE_INUSEZERO or
  LINEADDRESSSTATE_INUSEONE  or LINEADDRESSSTATE_INUSEMANY or
  LINEADDRESSSTATE_NUMCALLS;
  TapiResult := lineGetStatusMessages(fLine,
    fLineStates, fAddressStates);
  fLineStatesSelected := fLineStates;
  fAddressStatesSelected := fAddressStates;
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```


function lineSetStatusMessages TAPI.pas

Syntax

function lineSetStatusMessages(hLine: HLINE; dwLineStates, dwAddressStates: DWORD): Longint; stdcall;

Description

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.

Parameters

hLine: A handle (HLINE) to the line device

dwLineStates: A DWORD holding a bit array that identifies for which line device status changes a message is to be sent to the application. This parameter uses the LINEDEVSTATE_ constants explained in Table 9-20.

dwAddressStates: A DWORD holding a bit array that identifies for which address status changes a message is to be sent to the application. This parameter uses the LINEADDRESSSTATE_ constants described in Table 8-2.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALADDRESSSTATE, LINEERR_OPERATIONFAILED, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINESTATE, LINEERR_UNINITIALIZED, LINEERR_NOMEM, and LINEERR_OPERATIONUNAVAIL.

See Also

LINE_CLOSE, LINE_LINEDEVSTATE, lineInitialize, lineInitializeEx, lineOpen

Example

Listing 9-2 shows how to call the lineSetStatusMessages() function.

Listing 9-2: Calling the lineSetStatusMessages() function

```
function TTapiInterface.SetStatusMessages(RequestedLineStates,
         RequestedAddressStates : DWord): boolean;
begin
  TapiResult := lineSetStatusMessages(fLine, RequestedLineStates,
         RequestedAddressStates);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

function lineSetCallPrivilege TAPI.pas

Syntax

function lineSetCallPrivilege(hCall: HCALL; dwCallPrivilege: DWORD): Longint; stdcall;

Description

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

Parameters

hCall: A handle (HCALL) to the call whose privilege is to be set. The call state of *hCall* can be any state.

dwCallPrivilege: A DWORD indicating the privilege the application wants to have for the specified call. Only a single flag can be set. This parameter uses the following LINECALLPRIVILEGE_ constants:
LINECALLPRIVILEGE_MONITOR indicates that the application requests monitor privilege to the call (these privileges allow the application to monitor state changes and query information and status about the call).
LINECALLPRIVILEGE_OWNER indicates that the application requests owner privilege to the call (these privileges allow the application to manipulate the call in ways that affect the state of the call).

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSTATE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALCALLPRIVILEGE, LINEERR_UNINITIALIZED, and LINEERR_NOMEM.

See Also

lineDrop

Example

Listing 9-3 shows how to use the lineSetCallPrivilege() function.

Listing 9-3: Using the lineSetCallPrivilege() function

```
function TTapiInterface.SetCallPrivilege(ACall : HCall; LevelRequested :
                                          TCallPrivilegeLevel): boolean;
begin
  if LevelRequested = cplOwner then
    TapiResult := lineSetCallPrivilege(ACall, LINECALLPRIVILEGE_OWNER)
  else
    TapiResult := lineSetCallPrivilege(ACall, LINECALLPRIVILEGE_MONITOR);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

In this chapter we have taken a detailed look at TAPI messages, but we have not yet looked at the essential TAPI functions of placing and receiving calls. We will cover those important topics in the remaining two chapters.

Chapter 10

Placing Outgoing Calls

In the last two chapters we laid a solid foundation for starting to work with TAPI. We have discussed the essential functions and structures used to initialize TAPI, open and close line devices, and handle TAPI messages. We are now ready to do something worthwhile—place outgoing calls.

We will examine two ways to place calls, one simple and the other more involved. Both of these phone calling approaches depend on a dialable phone number, one that is properly formatted so that TAPI can use it in placing an outgoing call. Once you have such a phone number, you may use either of the two programming approaches to allow your users to place that call. We'll start by considering the various types of phone number representations, or addresses, as they are called in TAPI. Then, we'll consider how Assisted Telephony provides a simple way to add call placing functionality to a wide variety of application types. Finally, we'll examine the standard, low-level manner of placing calls, one that depends on the foundation we have laid in the previous two chapters.

Canonical and Dialable Address Formats

In TAPI a phone number or address can exist in more than one format. The two common address formats are *canonical* and *dialable*. There is also a *displayable* address, which is the basic phone number without any special or control characters—the one you would actual display or the user would enter. Since phone number formats tend to vary from one country to another, there needs to be a standard international format for storing them. Enter the *canonical address* format. With it you can represent any phone number from anywhere in the world. Because of that, the canonical format is ideal for storing phone numbers in a database.

A canonical address or phone number is an ASCII string that contains certain characters with specific meanings in a specific order. All canonical addresses begin with the plus (+) character. This character has the function of identifying the string as a canonical address, nothing more. This is followed by the country code, a variable length string of digits delimited by a space character at the end.

Next is an optional area code, another variable length string of digits surrounded by parentheses, as in (301). Next is the subscriber number, the main phone number. This portion consists of the digits that represent that dialable number with possible formatting characters that we'll discuss presently under dialable addresses.

There may or may not be additional information in the canonical address. To indicate the presence of additional information after its end, the subscriber number will be followed by a pipe (|) character. That will be followed by the additional and optional parts that could include the sub-address portion or the name portion. The former could represent an e-mail address or an ISDN subaddress. The latter would simply be the name of the subscriber, a name that could be displayed. For additional information, see the TAPI Help file.

A dialable address is equally complex. It consists of the main portions we just discussed in relation to canonical addresses and more. Those elements are shown in Table 10-1 (see the TAPI Help file for additional information).

Table 10-1: Elements of a dialable address

<table>
<tr><th>Element</th><th>Meaning</th></tr>
<tr><td>Dialable number</td><td>A series of digits and modifier characters (0-9 A-D * # , ! W w P p T t @ $?) delimited by the dialable address string, the end of the string, or by one of the following characters: | ^ CRLF (#13#10).</td></tr>
<tr><td>!</td><td>This character indicates that a hookflash (one-half second onhook, followed by one-half second offhook before continuing) is to be inserted in the dial string.</td></tr>
<tr><td>P or p</td><td>This character indicates that dialing is done using the older pulse method on the digits that follow.</td></tr>
<tr><td>T or t</td><td>This character indicates that dialing is done using the newer tone (DTMF) dialing method on the digits that follow.</td></tr>
<tr><td>,</td><td>This character indicates that dialing is to be paused. The duration of a pause is device specific and can be retrieved from the line's device capabilities. You may use multiple commas to provide longer pauses.</td></tr>
<tr><td>W or w</td><td>This character indicates to wait for a dial tone until proceeding with dialing.</td></tr>
<tr><td>@</td><td>This character indicates to "wait for a quiet answer" (at least one ringback tone followed by several seconds of silence) before dialing the remainder of the dialable address.</td></tr>
<tr><td>$</td><td>This character indicates entering or dialing the billing information should wait for a "billing signal" (such as a credit card prompt tone).</td></tr>
<tr><td>?</td><td>This character indicates that the user will be prompted before continuing with dialing. The "?" character forces the provider to reject the string as invalid, alerting the application to break the string into pieces and prompt the user.</td></tr>
<tr><td>;</td><td>This character, if placed at the end of a partially specified dialable address string, indicates that the dialable number information is incomplete and that additional address information will be provided later. It is allowed only in the DialableNumber portion of an address.</td></tr>
<tr><td>|</td><td>This optional character indicates that the information following it up to the next + | ^ CRLF (or the end of the dialable address string) should be treated as subaddress information</td></tr>
<tr><td>Sub address</td><td>A variably sized string containing a subaddress and delimited by the next + | ^ CRLF or the end of the address string</td></tr>
</table>

Element	Meaning
^	This optional character indicates that the information following it up to the next CRLF or the end of the dialable address string should be treated as an ISDN name.
Name	A variably sized string containing name information and delimited by CRLF or the end of the dialable address string
CRLF	This optional character pair indicates that the current dialable number is following by another dialable number.

The dialable address is important since many TAPI functions need to use it. With such a dialable address, you can implement any of the functionality outlined in Table 10-1. For example, in our sample Call Manager, we allow the user to select pulsed dialing in a check box. If checked, we append a "p" to the beginning of the dialable address, which causes the modem to use the pulse-dialing method instead of tone dialing.

While the canonical and dialable address formats are similar, there are important differences. Canonical addresses are more universal and applicable to many telephone systems. On the other hand, dialable addresses enable you to actually send phone numbers as parameters to and from Windows TAPI functions. The latter takes into account local considerations and often includes additional digits needed by the local system. For example, you may have to add digits like 1, 8, or 9 for long distance and/or an outside line. In the sample code, we append a letter "p" at the beginning of the string if the Pulse Dialing box is checked on the sample Call Manager program that accompanies this chapter and the next.

Assisted Telephony

Later in this chapter, we'll begin to create a full-featured Call Manager application using many of the low-level TAPI functions we discussed in previous chapters. However, we don't always need that degree of sophistication. Sometimes all we need to provide is limited call placing functionality within a word processor, spreadsheet, database application, or personal information managers (PIM). Conveniently for developers, TAPI includes a small subset of functions called Assisted Telephony for just this kind of situation.

The goal of Assisted Telephony is to provide the basic functionality of placing voice calls or media calls in a Win32-based application. With Assisted Telephony, your application can essentially ignore the complex details of the full TAPI services we've discussed and will continue to expose in this and the next chapter. It extends basic telephony functionality to any type of application from which a user might want to place a phone call. For example, you could use the Assisted Telephony function tapiRequestMakeCall() to allow users of a spreadsheet application to automatically dial telephone numbers stored in that spreadsheet by simply double-clicking on the field containing the phone number.

Be aware that functionality beyond simple dialing (such as the transmission and reception of data) requires additional data-transfer APIs, including the communications functions of the Comm API. One note of caution: Since Assisted Telephony and full TAPI are used and implemented in different ways, you should not mix Assisted Telephony function calls and Telephony API function calls within the same application. While your users can place calls, they cannot accept incoming calls. For that you need the full TAPI to create a Call Manager. However, if you're looking for an easy-to-use means to allow your users to make phone calls without having to use the low-level TAPI functions, Assisted Telephony is the answer.

TIP: Never mix Assisted Telephony function calls and Telephony API function calls within the same application.

You need two kinds of applications to implement Assisted Telephony: Assisted Telephony clients and servers. The clients use Assisted Telephony by calling certain functions that have a prefix of "tapi." An example would be any application that includes a Dial button to execute a command that dials a phone number. On the other hand, an Assisted Telephony server is able to execute such Telephony API functions that have been requested by another (client) application calling a "tapi"-prefixed function. How can we be sure such an application will be available? That is generally not an issue. In fact, most modern computers come equipped with voice modems and include a Call Manager program that can access these TAPI services.

Every Assisted Telephony server must be registered with Windows, including any that you may write yourself. A server accomplishes this self-registration by calling the lineRegisterRequestRecipient() function. Once it has done this, it will be available for client applications that want to request its services. The Assisted Telephony functions (beginning with the prefix "tapi") are known as *request functions*. The Assisted Telephony applications that process these requests are known as *request recipients*. Now we'll examine some of the subtle details of Assisted Telephony.

When an application uses the Assisted Telephony services to initiate a request, that request is temporarily queued by TAPI. The request recipient application (server) that retrieves these requests will execute them on behalf of the Assisted Telephony application (client). You should call the tapiRequestMakeCall() function to establish a voice call. Note that a "requesting" or client application will not control the call; instead, the Call Manager application that assumes the role of server will control the call. If you need to control a call, you should use another approach (which we will discuss later in this chapter) instead of the Assisted Telephony approach we are discussing here.

With TAPI, a user may set different recipient applications or the same recipient application to handle each of these services. As we indicated above, an application becomes a request recipient by registering itself using the lineRegisterRequestRecipient() function, specifying TRUE for the value of the *bEnable* parameter. On the other hand, if you specify FALSE for this parameter, the function will unregister that application as a request recipient. A server application should do this when it has determined that its recipient duties are finished for the current session. When it calls the lineRegisterRequest-Recipient() function, the server application will select the services it wants to handle by specifying them in the *dwRequestMode* parameter of the function. One possible value for a request is LINEREQUESTMODE_MAKECALL, indicating that the application wants to handle tapiRequestMakeCall() requests.

If multiple applications register for the same service(s), a priority scheme will be used to select the application that will be the preferred one for handling requests. This priority scheme is the same as that used for call handoff and for the routing of incoming calls. It is based on a list of filenames in the Handoff-Priorities section of the Windows registry.

How does a client application request TAPI services? The process by which such an application may request services is shown in Figure 10-1. Here's how it works. First, an application must request some basic telephony service, such as placing a call. When TAPI receives an Assisted Telephony request, it first attempts to identify a request recipient—an application currently registered to process that particular type of request. If such a request recipient can be located, the request is then queued. The highest priority application that has registered itself for that request's service is sent a LINE_REQUEST message. That message notifies the request recipient that a new request has arrived, including information about the request's mode.

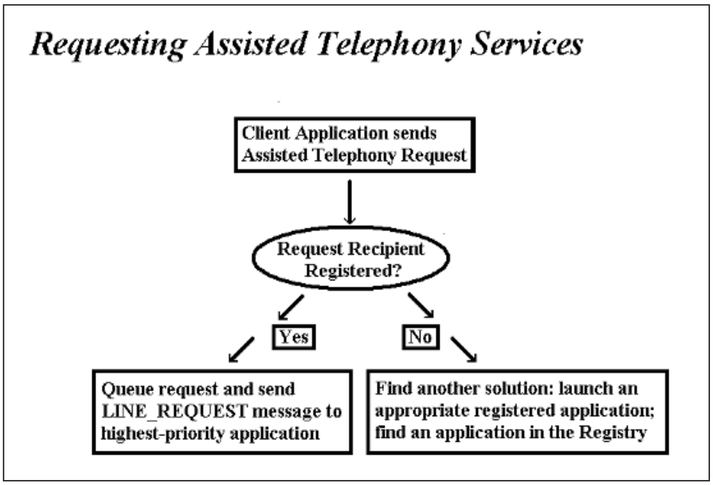


Figure 10-1: The process of requesting TAPI services

What if TAPI is unable to find an appropriate *request server*? If it cannot find a currently running application to process the request, it will try to launch an application that has been registered as having the necessary capabilities. This particular registration information, if it exists, will be stored in the Handoff-Priorities section of the Windows registry. TAPI will attempt to launch applications in the order in which they are listed in that section. If no application is currently registered, TAPI will not give up yet. It will continue by examining the list of request-processing applications in the associated entry within that section of the Windows registry.

Of course, there may be situations in which a server application cannot be found or used. If the associated line is missing, if there are no applications listed on it, or if none of the applications in the list can be launched, the request will be rejected with the TAPIERR_NOREQUESTRECIPIENT error. However, when a request recipient is launched (directly by TAPI or otherwise), it will accept the responsibility to call the lineRegisterRequestRecipient() function during the startup process in order to register itself as a request recipient.

As a good citizen among Windows technologies, TAPI will always relate to the registry using a systematic approach: If one or more applications are listed in the Windows registry entry, TAPI will begin with the first listed application (highest priority). It will attempt to launch that application by calling CreateProcess(). If that fails, it will then attempt to launch the next application in the list, continuing until there are no applications left to try.

When a request recipient (server) application has been launched successfully, TAPI will queue the request and return an indication of success. This will occur early in the process, before the request recipient deals with the request that caused it to be opened. After such an application has been launched, it will call the lineRegisterRequestRecipient() function, which in turn will cause a LINE_REQUEST message to be sent by Windows. This message signals that the request has been queued. If for some reason the launched application never becomes properly registered, any request that caused it to be opened will remain in the queue indefinitely or at least until an application becomes properly registered for that type of request.

To summarize the process, if TAPI finds an appropriate application already registered and running or is able to successfully launch one, it will then queue the request and send a LINE_REQUEST message to the server application. It will also return a success result for the function call to the Assisted Telephony application. Be aware that this success message will indicate only that the request has been accepted and queued; it will not necessarily indicate that it has been successfully executed.

TAPI Servers in Assisted Telephony

We've seen how TAPI locates an appropriate server application, but how do these server applications themselves work? When the server application is ready to process a request, it will call the function lineGetRequest(). By calling this function, the server will receive whatever information it needs, such as an address (dialable) to dial. The server will then process the request using the various telephony API functions (lineMakeCall(), lineDrop() and so on) that would otherwise be used to place the call. When you call lineGetRequest(), you are essentially removing the request from TAPI's radar screen. After that function call, the request parameters will be copied to an application-allocated request buffer. The size and interpretation of the contents of that buffer will vary depending on the request mode. Since these functions are part of Basic TAPI and not Assisted Telephony, we'll discuss them later in this chapter.

A TAPI server must fulfill certain responsibilities. Importantly, it must ensure that it uses the correct parameters when executing requests from a client using Assisted Telephony. When doing so, it will follow these steps:

1. The request recipient will receive a LINE_REQUEST message from Windows alerting it that requests can exist for it in the request queue. Essentially, this triggers the application to call the lineGetRequest() function and continue to call it until the queue is drained (if the request is to make a new call) or to drop an existing call. This message will not contain the parameters for the request, except in the case of a request to drop an existing call.
2. If the request is to make a new call, the Assisted Telephony server must first allocate the memory needed to store the needed information and then call the lineGetRequest() function to retrieve the full request information, including the request's parameters. After this, the server will have all the information it needs, such as the number to dial or the identification of the maker of the request.
3. Finally, the server executes the request by invoking the appropriate low-level TAPI function or set of functions.

Sometimes TAPI cannot launch a server application that is capable of performing the duties of a request recipient. When this happens, the Assisted Telephony call will fail, returning the TAPIERR_NOREQUESTRECIPIENT error.

What kind of information is processed during an Assisted Telephony request? How is that information processed by the various systems involved? The TAPI Help file provides the following description and makes certain recommendations:

- The default registry entry should list a Call Manager application in the priority list for tapiRequestMakeCall(). It would be helpful, but is not essential, for that call manager application to have a menu option that allows users to set it to the highest priority.
- When an Assisted Telephony recipient application has been launched automatically by TAPI, and assuming that it is the only TAPI application in the system, this action will initialize TAPI. It will go through all of the steps we described in Chapter 8. If the Assisted Telephony recipient application initializes and shuts down the line device before registering for Assisted Telephony requests, TAPI will be shut down as well, and the Assisted Telephony request will be lost. Assisted Telephony requests might also be lost if another TAPI application that is launched performs a TAPI initialization and shutdown.

Assisted Telephony Functions

Having examined the role of Assisted Telephony servers, we will now return our attention to Assisted Telephony clients and the specific functions they must call to request these services. There are four functions associated with Assisted Telephony: tapiRequestMakeCall(), tapiGetLocationInfo(), tapiRequestMediaCall(), and tapiRequestDrop(). Since the last two are obsolete and nonfunctional in Win32-based applications, you should avoid using them; although they are included in TAPI.pas for backward compatibility, we will not discuss them in this book. The first function, tapiRequestMakeCall(), will attempt to establish a voice call between the application user and a remote party specified by its phone number.

Here's how the process works: Windows will send the request to place the call to TAPI, which will then pass it to an application that is registered as a recipient of such requests—a Call Manager application. Note that after your application has made such a request, the call will be controlled entirely from the call manager application. Assisted Telephony applications cannot manage calls themselves. By using this function, the call manager application will handle the more complex telephony aspects and any needed user-interface operations. Therefore, any application for which you provide this kind of telephony support need not be modified in any substantial way. Without a doubt, Assisted Telephony is the easiest form of telephony programming. Use it whenever you can.

Using the Assisted Telephony functions is extremely straightforward. To enable your application to have a call placed by tapiRequestMakeCall(), you need only provide the call's destination phone number. TAPI will forward the request to the appropriate server application, which in turn will actually place the call on behalf of your application. As you may be aware, a default call control application is provided as part of Win32 Telephony. At the same time, users have the option to replace this with a call control application of their choice.

There are certain situations where you may encounter problems with Assisted Telephony. If you attempt to invoke tapiRequestMakeCall() when no call control application is running, the function will return the TAPIERR_NOREQUESTRECIPIENT error indication. If the call control application is not running, TAPI will attempt to launch the highest priority call control application (which is listed for tapiRequestMakeCall() in the registry). If you try to invoke this function when the Assisted TAPI request queue is full, it will return the TAPIERR_REQUESTQUEUEFULL error. Now we'll provide a reference for the functions themselves.

function tapiRequestMakeCall TAPI.pas

Syntax

function tapiRequestMakeCall(lpszDestAddress, lpszAppName, lpszCalledParty, lpszComment: LPCSTR): Longint; stdcall;

Description

This function requests that a voice call be established. For this to work, a Call Manager application must be responsible for establishing the call on behalf of the requesting application; the call will then be controlled by the user's Call Manager application.

Parameters

lpszDestAddress: An LPCSTR that points to a memory location where the NULL-terminated destination address of the call request is located. The address can be in canonical address format or the dialable address format. Note that the validity of the specified address will not be checked by this operation. The maximum length of the address is TAPIMAXDEST-ADDRESSSIZE characters, which includes the NULL terminator.

lpszAppName: An LPCSTR that points to a memory location where a user-friendly application name (NULL-terminated string) of the call request is stored. This pointer may be left NULL if the application does not wish to supply an application name. The maximum length of the address is TAPI-MAXAPPNAMESIZE characters, which includes the NULL terminator. Longer strings will be truncated.

lpszCalledParty: An LPCSTR that points to a memory location where the ASCII NULL-terminated called party name for the called party is located. This pointer may be left NULL if the application does not wish to supply this information. The maximum length of the string is TAPIMAXCALLED-PARTYSIZE characters, which includes the NULL terminator. Longer strings are truncated.

lpszComment: An LPCSTR that points to a memory location where the ASCII NULL-terminated comment about the call is located. This pointer may be left NULL if the application does not wish to supply a comment. The maximum length of the address is TAPIMAXCOMMENTSIZE characters, which includes the NULL terminator. Longer strings are truncated.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible error return values are TAPIERR_NOREQUESTRECIPIENT, TAPIERR_INVALDESTADDRESS, TAPIERR_REQUESTQUEUEFULL, and TAPIERR_INVALPOINTER.

Example

Listing 10-1 shows how to call the Assisted Telephony dialing function.

Listing 10-1: Calling the Assisted Telephony dialing function

```
function TTapiInterface.DialWithAssistedTelephony: boolean;
begin
  if PhoneNumber='' then
    begin
      ShowMessage('You need to enter a phone number');
      result := false;
      exit;
    end;
  // using assisted telephony
  TAPIResult := TapiRequestMakeCall(
    PChar(PhoneNumber), // the phone number
    '', // application name, optional, could use PChar(Application.Title)
    '', // optional, this is the name of the person being called
    ''); // optional comment
  result := TAPIResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

function tapiGetLocationInfo   TAPI.pas

Syntax

function tapiGetLocationInfo(lpszCountryCode, lpszCityCode: LPCSTR): Longint; stdcall;

Description

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

Parameters

lpszCountryCode: A pointer to a NULL-terminated ASCII string specifying the country code for the current location. You should allocate at least eight bytes of storage at this location to hold the string (TAPI will not return more than eight bytes, including the terminating NULL character). TAPI will return an empty string if the country code has not been set for the current location.

lpszCityCode: A pointer to a NULL-terminated ASCII string specifying the city (area) code for the current location. You should allocate at least eight bytes of storage at this location to hold the string (TAPI will not return more than eight bytes, including the terminating NULL). TAPI will return an empty string if the city code has not been set for the current location.

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. A possible return value is TAPIERR_REQUESTFAILED.

As we indicated, there are two Assisted Telephony functions that are no longer used. These are the tapiRequestMediaCall() function and the tapiRequest-Drop() function. Both are nonfunctional in Win32-based applications and obsolete for all classes of Windows-based applications. Microsoft advises not to use either function, and we have not covered them here. If you need to work with either function in support of an older TAPI application, see the TAPI Help file and the declarations in TAPI.pas for information. We have concluded our discussion of Assisted Telephony and the various types of phone numbers (addresses). Now we'll turn our attention to low-level TAPI functions used in placing a call.

Establishing a Call with Low-Level Line Functions

In previous chapters, we provided an overview of the process to initialize and close down TAPI and those to open and close line devices. Please be aware that an understanding of the material in those chapters is essential as a foundation for the functions and functionality we will be discussing from this point onward. Now we'll begin the process of examining some of the important functions used between opening and closing a line device. In the next chapter we'll complete that process and discuss handling incoming calls.

As we stated already, one of the most common tasks a telephony application can perform is placing a call. Once an application has opened the line device, it can place a call using the lineMakeCall() function. During this process, it must specify the address (phone number and area code) to be called in the *lpszDest-Address* parameter and the media mode (datamodem, in this case) desired in the *lpCallParams* parameter. This function will return a positive "request ID" if completed asynchronously or a negative error number if a problem has occurred. Negative return values describe specific error states. LINEERR_CALLUNAVAIL, for example, indicates that the line is probably in use (someone else already has an active call). If dialing completes successfully, messages will be sent to an application to inform it about the call's progress. Applications typically use these messages to display status reports to the user, as we demonstrate in our Call Manager.

Later, when the lineMakeCall() function has successfully set up the call, your application will receive a LINE_REPLY message (the asynchronous reply to lineMakeCall()). At this point, your application will not necessarily have established a connection to the remote destination station quite yet; rather, it has simply established a call at the local end, perhaps indicated by the presence of a dial tone. This LINE_REPLY message simply informs the application that the call handle returned by lineMakeCall() is valid.

As shown in Figure 10-2, a call can go through various states. Each of these states is reflected in a LINE_CALLSTATE message, which we discussed in Chapter 9. These states include dial tone present, dialing, ringback, and, if the connection succeeds, LINECALLSTATE_CONNECTED. (To see the complete list of call states, see the LINECALLSTATUS structure.) After your application receives this message indicating a successful connection, it can begin sending data.

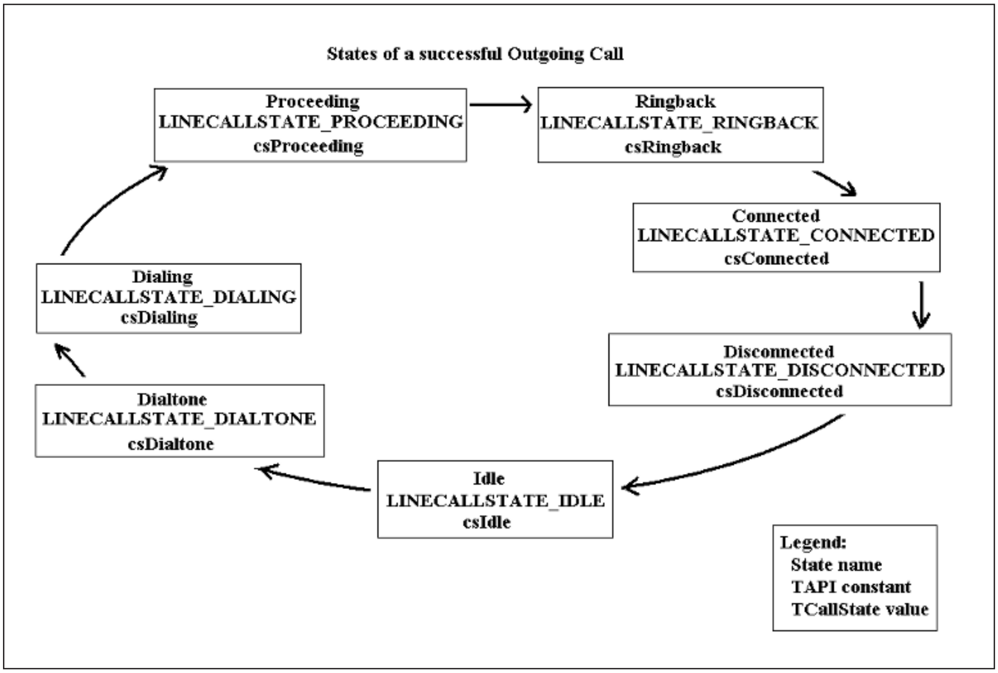


Figure 10-2: States of a successful outgoing call

What about data calls, calls that send files or other data over a phone line rather than enable a voice conversation? Interestingly, TAPI's programming model treats data calls in a manner similar to voice calls. This is demonstrated by the fact that this same function, lineMakeCall(), can be used to initiate calls of both types. If LINEBEARERMODE_DATA is specified in a field of the *lpCallParams* parameter of lineMakeCall(), the call will be set up to send data. To select speech transmission, you must use a different value. If you specify 0, TAPI will establish a default 3.1 kHz voice call—one that can support the speech, fax, and modem media modes.

Again, we must emphasize that the low-level call placing method we're about to discuss depends on the TAPI initialization functions we discussed in Chapter 8. Once your application has initialized TAPI, determined that a given line offers the needed set of capabilities, and opened that line, you can access various telephony functions for either incoming or outgoing calls on the line. (Of course, we take care of these details in all of the sample applications.) The usual way to place a call on that line is to call the lineMakeCall() function, specifying the line handle and a dialable destination address.

The first step is to dial the call (using a dialable address). When you call the lineMakeCall() function, it will first attempt to obtain a call appearance on a line address, and then it will wait for a dial tone. Finally, it will dial the specified address or phone number. The TAPI Help file defines a call appearance simply as a "connection to the switch over which a call can be made." Interestingly, once your application has established the connection, that call appearance will exist, even if the call itself has not yet been placed. After the call has been established, the call appearance will remain in existence until the call transitions to the idle state. If calls controlled by other applications exist on the line, these calls would normally be in an on hold state and would typically be forced to stay on hold until your application either dropped its current call or placed it on hold. If dialing is successful, a handle to a call with owner privileges will be returned to your application.

Before you call the lineMakeCall() function, you must set up the parameters for the call and store them in a LINECALLPARAMS data structure. The lineMakeCall() function has a parameter that points to this structure. Using this structure's fields, you can specify the quality of service you want to request from the network. You can also specify a variety of ISDN call setup parameters. If you neglect to provide a LINECALLPARAMS structure to lineMakeCall(), don't worry; TAPI will provide a default POTS voice-grade call with a set of default values.

TIP: Use LINECALLPARAMS to accurately keep track of and report call information (such as the identification of the called party).

The phone call's origination address will also be included in LINECALL-PARAMS. Using this field, your application can specify the address on the line where it wants the call to originate. It can do so by specifying an address ID, though in some configurations, it is more practical to identify the originating address by its position in a directory. As we stated previously, do not mix function calls of the line API with the functions of Assisted Telephony. The actions requested by lineMakeCall() would happen automatically in response to another application that requested that functionality by calling the Assisted Telephony function tapiRequestMakeCall().

Once dialing is complete and the call is in the process of being established, it passes through a number of different states. Windows will inform an application of these states (the progress of the call) using LINE_CALLSTATE messages. Relying on this mechanism, your application can keep track of these stages and determine if the call is actually reaching the called party.

TIP: A robust telephony application should base its behavior on the information received in these messages and not make assumptions about a call's state. In fact, you should consider passing this information along to an application's user in a status bar or memo control, as we do in our sample Call Manager application.

If you want your application to take special call setup parameters into consideration, you must supply them to lineMakeCall(). The TAPI Help file emphasizes that call setup parameters are required for the following actions:

- To request a special bearer mode, bandwidth, or media mode for a call
- To send user-to-user information (with ISDN)
- To secure the call
- To block the sending of a caller ID to the called party
- To take the phone off the hook automatically at the originator and/or the called party

Special Dialing Support

As we've emphasized, dialing a phone number is one of the most basic and essential telephony functions. As we've discussed, the lineMakeCall() function performs this task for simple calls, but what about more complex situations that involve dialing on an existing call appearance, such as transferring a call or adding a call to a conference? TAPI provides the lineDial() function for this purpose.

Here's how this process works. First, you must set up a call for transferring or conferencing. Second, TAPI will automatically allocate a consultation call, and you can call the lineDial() function to perform the actual dialing of this consultation call. Third, if needed, you may invoke lineDial() multiple times in multistage dialing if the line's device capabilities allow it. You may also include multiple addresses in a single dial string, but they must be separated by the CRLF (#13#10) character pair.

Of course, different service providers will support different functionality. Those that support inverse multiplexing could establish individual, physical calls with each of the addresses. They can return a single call handle to the aggregate of all calls to an application. In this scenario, all of the addresses would use the same country code. Those service providers that support inverse multiplexing may allow multiple addresses to be provided at once.

From TAPI's perspective, dialing is considered complete when the address has been passed to the service provider, not when the call is finally connected. As before, Windows informs an application of the progress of the call using LINE_CALLSTATE messages. If you want to provide the ability for the user to abort a call attempt while that call is in the process of being established, you should use the lineDrop() function.

If you want to indicate that dialing is complete, you can set the *lpszDestAddress* parameter of the lineDial() function to an empty string. However, you should do this only if that parameter (*lpszDestAddress*) in the previous calls to the lineMakeCall() and lineDial() had strings that were terminated with semicolons.

The lineDial() function can return various results to indicate success or failure. If it returns LINEERR_INVALADDRESS, no dialing took place. If it returns LINEERR_DIALBILLING, LINEERR_DIALQUIET, LINEERR_DIALDIALTONE, or LINEERR_DIALPROMPT, none of the usual actions performed by lineDial() have occurred; none of the dialable addresses before the offending character have been dialed, and therefore, no hookswitch state has changed.

function lineDial TAPI.pas

Syntax

```
function lineDial(hCall: HCALL; lpszDestAddress: LPCSTR; dwCountryCode:
DWORD): Longint; stdcall;
```

Description

This function dials the specified dialable number on the specified call.

Parameters

hCall: A handle (HCALL) to the call on which a number is to be dialed. The application must be an owner of the call. The call state of *hCall* can be any state except idle and disconnected.

lpszDestAddress: An LPCSTR holding the destination to be dialed using the standard dialable number format

dwCountryCode: A DWORD holding the country code of the destination. This is used by the implementation to select the call progress protocols for the destination address. If a value of zero is specified, a service provider-defined default call progress protocol is used.

Return Value

This function returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or a negative error number if an error has occurred. Possible return values are LINEERR_ADDRESSBLOCKED, LINEERR_INVALPOINTER, LINEERR_DIALBILLING, LINEERR_NOMEM, LINEERR_DIALDIALTONE, LINEERR_NOTOWNER, LINEERR_DIALPROMPT, LINEERR_OPERATIONFAILED, LINEERR_DIALQUIET, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_ UNINITIALIZED, and LINEERR_INVALCOUNTRYCODE.

See Also

LINE_CALLSTATE, LINE_REPLY, lineDrop, lineMakeCall

Example

Listing 10-2 shows how to use the lineDial() function when placing a phone call.

Listing 10-2: Placing a phone call with TAPI

```
function TTapiInterface.PlaceCall: boolean;
begin
  TapiResult := lineDial(FCall, '', 0);
  result := TapiResult>0;
  If NOT Result then
    ReportError(TapiResult)
  else
    OnSendTapiMessage('Number dialing initiated successfully');
end;
```

function lineMakeCall   TAPI.pas

Syntax

function lineMakeCall(hLine: HLINE; lphCall: PHCall; lpszDestAddress: LPCSTR; dwCountryCode: DWORD; CallParams: PLineCallParams): Longint; stdcall;

Description

This function places a call on the specified line to the specified destination address. Optionally, call parameters can be specified to request anything beyond the default call setup parameters.

Parameters

hLine: A handle (HLINE) to the open line device on which a call is to be originated

lphCall: A pointer (PHCall) to an HCALL handle. The handle is only valid after the LINE_REPLY message is received by the application indicating that the lineMakeCall() function successfully completed. Use this handle to identify the call when invoking other telephony operations on the call. The application will initially be the sole owner of this call. This handle is void if the function returns an error (synchronously or asynchronously by the reply message).

lpszDestAddress: A pointer (LPCSTR) 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 dial tone state until dialing begins. Service providers that have inverse multiplexing capabilities may allow an application to specify multiple addresses at once.

dwCountryCode: A DWORD indicating the country code of the called party. If a value of zero is specified, a default is used by the implementation.

CallParams: A pointer (PLineCallParams) to a LINECALLPARAMS structure. This structure allows the application to specify how it wants the call to be set up. If NULL is specified, a default 3.1 kHz voice call is established and an arbitrary origination address on the line is selected. This structure allows the application to select elements such as the call's bearer mode, data rate, expected media mode, origination address, blocking of caller ID information, and dialing parameters.

Return Value

This function returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or a negative error number if an error has occurred. Possible return values are LINEERR_ADDRESSBLOCKED, LINEERR_INVALLINEHANDLE, LINEERR_BEARERMODEUNAVAIL, LINEERR_INVALLINESTATE, LINEERR_CALLUNAVAIL, LINEERR_INVALMEDIAMODE, LINEERR_DIALBILLING, LINEERR_INVALPARAM, LINEERR_ DIALDIALTONE, LINEERR_INVALPOINTER, LINEERR_DIALPROMPT, LINEERR_INVALRATE, LINEERR_DIALQUIET, LINEERR_NOMEM, LINEERR_INUSE, LINEERR_OPERATIONFAILED, LINEERR_INVALADDRESS, LINEERR_OPERATIONUNAVAIL, LINEERR_INVALADDRESSID, LINEERR_RATEUNAVAIL, LINEERR_INVALADDRESSMODE,

LINEERR_RESOURCEUNAVAIL, LINEERR_INVALBEARERMODE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALCALLPARAMS, LINEERR_UNINITIALIZED, LINEERR_INVALCOUNTRYCODE, and LINEERR_USERUSERINFOTOOBIG.

See Also

LINE_CALLSTATE, LINE_REPLY, LINECALLPARAMS, LINEDEVSTATUS, lineDial, lineDrop, lineGetLineDevStatus

Example

Listing 10-3 shows how to place a call with the lineMakeCall() function.

Listing 10-3: Placing a call with the lineMakeCall() function

```
function TTapiInterface.RequestLine(var ATAPIResult: DWord): boolean;
begin
  App := @Application;
  if not fLineIsOpen then // if a line is open, no need to open one
    begin
      OpenLine(ATAPIResult, false);
      if ATAPIResult<>0 then
        begin
          result := false;
          ReportError(TAPIResult);
          Exit; // no point in continuing if we cannot open line
        end;
    end;
  // now place the call
  if PulseDialing then
  ATAPIResult := LineMakeCall(fLine, @FCall, PChar('p'+PhoneNumber),
    FCountryCode, FPLineCallParams)
  else
  ATAPIResult := LineMakeCall(fLine, @FCall, PChar(PhoneNumber), FCountryCode,
    FPLineCallParams);
  result := ATAPIResult>0;
  if result then OnSendTapiMessage('Placing phone call was successful')
    else ReportError(ATAPIResult);
end;
```

structure LINECALLPARAMS TAPI.pas

The LINECALLPARAMS structure describes parameters supplied when making calls using the lineMakeCall() and TSPI_lineMakeCall() functions. The LINECALLPARAMS structure is also used as a parameter in other operations, such as the lineOpen() function. This structure is defined as follows in TAPI.pas:

```
  PLineCallParams = ^TLineCallParams;
  linecallparams_tag = packed record {// Defaults:        }
    dwTotalSize,                     {// ---------        }
    dwBearerMode,                    {// voice            }
    dwMinRate,                       {// (3.1kHz)         }
    dwMaxRate,                       {// (3.1kHz)         }
    dwMediaMode,                     {// interactiveVoice }
    dwCallParamFlags,                {// 0                }
    dwAddressMode,                   {// addressID        }
```

```
    dwAddressID: DWORD;              {// (any available)  }
    DialParams: TLineDialParams;     {// (0, 0, 0, 0)     }
    dwOrigAddressSize,               {// 0                }
    dwOrigAddressOffset,
    dwDisplayableAddressSize,
    dwDisplayableAddressOffset,
    dwCalledPartySize,               {// 0 }
    dwCalledPartyOffset,
    dwCommentSize,                   {// 0 }
    dwCommentOffset,
    dwUserUserInfoSize,              {// 0  }
    dwUserUserInfoOffset,
    dwHighLevelCompSize,             {// 0  }
    dwHighLevelCompOffset,
    dwLowLevelCompSize,              {// 0  }
    dwLowLevelCompOffset,
    dwDevSpecificSize,               {// 0  }
    dwDevSpecificOffset: DWORD;
{$IFDEF TAPI20}
    dwPredictiveAutoTransferStates,                      // TAPI v2.0
    dwTargetAddressSize,                                 // TAPI v2.0
    dwTargetAddressOffset,                               // TAPI v2.0
    dwSendingFlowspecSize,                               // TAPI v2.0
    dwSendingFlowspecOffset,                             // TAPI v2.0
    dwReceivingFlowspecSize,                             // TAPI v2.0
    dwReceivingFlowspecOffset,                           // TAPI v2.0
    dwDeviceClassSize,                                   // TAPI v2.0
    dwDeviceClassOffset,                                 // TAPI v2.0
    dwDeviceConfigSize,                                  // TAPI v2.0
    dwDeviceConfigOffset,                                // TAPI v2.0
    dwCallDataSize,                                      // TAPI v2.0
    dwCallDataOffset,                                    // TAPI v2.0
    dwNoAnswerTimeout,                                   // TAPI v2.0
    dwCallingPartyIDSize,                                // TAPI v2.0
    dwCallingPartyIDOffset: DWORD;                       // TAPI v2.0
{$ENDIF}
{$IFDEF TAPI30}
    dwAddressType: DWORD;                                // TAPI v3.0
{$ENDIF}
  end;
  TLineCallParams = linecallparams_tag;
  LINECALLPARAMS = linecallparams_tag;
```

If your application requires device-specific extensions, you should use the DevSpecific (*dwDevSpecificSize* and *dwDevSpecificOffset*) variably sized area of this data structure. This structure is used as a parameter to the lineMakeCall() function we discussed earlier when setting up a call. You can use its fields to enable your application to specify the quality of service you want from the network or to set a variety of ISDN call setup parameters. As we indicated above, if you do not supply a LINECALLPARAMS structure when calling lineMakeCall(), TAPI will assume that a default POTS voice-grade call is being requested and use the default values.

Note that the fields *DialParams* through *dwDevSpecificOffset* will be ignored when an *lpCallParams* parameter is specified with the lineOpen() function. The fields *dwPredictiveAutoTransferStates* through *dwCallingPartyIDOffset* will be available only to applications that open the line device with a TAPI version of 2.0 or higher. The *dwAddressType* field will be available only to applications that open the line device with a TAPI version of 3.0 or later. The fields of this structure are defined in Table 10-2; for additional information on these fields, see the TAPI Help file.

Table 10-2: Fields of the LINECALLPARAMS structure

Field	Meaning
dwTotalSize	This field indicates the total size, in bytes, allocated to this data structure. This size should be big enough to hold all the fixed and variably sized portions of this data structure.
dwBearerMode	This field indicates the bearer mode for the call. This member uses one of the LINEBEARERMODE_ constants. If dwBearerMode is zero, the default value is LINEBEARERMODE_VOICE.
dwMinRate	This field indicates the minimum value of the data rate range requested for the call's data stream in bps (bits per second). When making a call, the service provider attempts to provide the highest available rate in the requested range.
dwMaxRate	This field indicates the maximum value of the data rate range requested for the call's data stream in bps (bits per second).
dwMediaMode	This field indicates the expected media type of the call. This member uses one of the LINEMEDIAMODE_ constants. If dwMediaMode is zero, the default value is LINEMEDIAMODE_INTERACTIVEVOICE.
dwCallParamFlags	This field holds flags that specify a collection of Boolean call setup parameters. This member uses one or more of the LINECALLPARAMFLAGS_ constants.
dwAddressMode	This field indicates the mode by which the originating address is specified using one of the LINEADDRESSMODE_ constants. Its value cannot be LINEADDRESSMODE_ADDRESSID for the lineOpen() function call.
dwAddressID	This field indicates the address identifier of the originating address if dwAddressMode is set to LINEADDRESSMODE_ADDRESSID.
DialParams	This field indicates the dial parameters to be used on this call of type LINEDIALPARAMS. When a value of zero is specified for this field, the default value for the field is used as indicated in the DefaultDialParams member of the LINEDEVCAPS structure.
dwOrigAddressSize	This field indicates the size, in bytes, of the variably sized field holding the originating address of this data structure. The format of this address is dependent on the dwAddressMode member.
dwOrigAddressOffset	This field indicates the offset, in bytes, from the beginning of the variably sized field holding the originating address of this data structure. The format of this address is dependent on the dwAddressMode member.
dwDisplayableAddressSize	This field indicates that the size of the displayable string is used for logging purposes. The content of these members is recorded in the dwDisplayable-AddressOffset and dwDisplayableAddressSize members of the call's LINECALLINFO message.
dwDisplayableAddressOffset	This field indicates the offset to the displayable string used for logging purposes. The content of these members is recorded in the dwDisplayableAddressOffset and dwDisplayableAddressSize members of the call's LINECALLINFO message.

Field	Meaning
dwCalledPartySize	This field is the size, in bytes, of the variably sized field holding called-party information from the beginning of this data structure. This information can be specified by the application that makes the call and is made available in the call's information structure for logging purposes. The format of this field is that of dwStringFormat, as specified in LINEDEVCAPS.
dwCalledPartyOffset	This field is the offset, in bytes, from the beginning of this data structure of the variably sized field holding called-party information. This information can be specified by the application that makes the call and is made available in the call's information structure for logging purposes. The format of this field is that of dwStringFormat, as specified in LINEDEVCAPS.
dwCommentSize	This field is the size, in bytes, of the variably sized field holding comments about the call. This information can be specified by the application that makes the call and is made available in the call's information structure for logging purposes. The format of this field is that of dwStringFormat, as specified in LINEDEVCAPS.
dwCommentOffset	This field is the offset, in bytes, of the variably sized field holding comments about the call from the beginning of this data structure. This information can be specified by the application that makes the call and is made available in the call's information structure for logging purposes. The format of this field is that of dwStringFormat, as specified in LINEDEVCAPS.
dwUserUserInfoSize	This field is the size, in bytes, of the variably sized field holding user-user information. The protocol discriminator field for the user-user information, if required, should appear as the first byte of the data pointed to by dwUserUserInfoOffset and must be accounted for in dwUserUserInfoSize.
dwUserUserInfoOffset	This field is the offset, in bytes, of the variably sized field holding user-user information from the beginning of this data structure.
dwHighLevelCompSize	This field is the size, in bytes, of the variably sized field holding high-level compatibility information.
dwHighLevelCompOffset	This field is the offset, in bytes, from the beginning of this data structure of the variably sized field holding high-level compatibility information.
dwLowLevelCompSize	This field is the size, in bytes, of the variably sized field holding low-level compatibility information.
dwLowLevelCompOffset	This field is the offset, in bytes, from the beginning of this data structure of the variably sized field holding low-level compatibility information.
dwDevSpecificSize	This field is the size, in bytes, of the variably sized field holding device-specific information.
dwDevSpecificOffset	This field is the offset, in bytes, from the beginning of this data structure of the variably sized field holding device-specific information.
dwPredictiveAutoTransferStates	This field indicates the LINECALLSTATE_ constants that would cause the call to be blind-transferred to the specified target address. It is set to zero if automatic transfer is not desired.
dwTargetAddressSize	This field is the size, in bytes, of a string specifying the target dialable address (not dwAddressID); used in the case of certain automatic actions.
dwTargetAddressOffset	This field is the offset from the beginning of LINECALLPARAMS of a string specifying the target dialable address (not dwAddressID); used in the case of certain automatic actions.

Field	Meaning
dwSendingFlowspecSize	This field is the total size, in bytes, of a WinSock2 FLOWSPEC structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in SendingFlowspec.len in a WinSock2 QOS structure. It specifies the quality of service desired in the sending direction on the call. The provider-specific portion following the FLOWSPEC structure must not contain pointers to other blocks of memory because TAPI does not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.
dwSendingFlowspecOffset	This field is the offset from the beginning of LINECALLPARAMS of a WinSock2 FLOWSPEC structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in SendingFlowspec.len in a WinSock2 QOS structure. It specifies the quality of service desired in the sending direction on the call. The provider-specific portion following the FLOWSPEC structure must not contain pointers to other blocks of memory because TAPI does not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.
dwReceivingFlowspecSize	This field is the total size, in bytes, of a WinSock2 FLOWSPEC structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in ReceivingFlowspec.len in a WinSock2 QOS structure. It specifies the quality of service desired in the receiving direction on the call. The provider-specific portion following the FLOWSPEC structure must not contain pointers to other blocks of memory because TAPI does not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.
dwReceivingFlowspecOffset	This field is the offset from the beginning of LINECALLPARAMS of a WinSock2 FLOWSPEC structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in ReceivingFlowspec.len in a WinSock2 QOS structure. It specifies the quality of service desired in the receiving direction on the call. The provider-specific portion following the FLOWSPEC structure must not contain pointers to other blocks of memory, because TAPI does not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.
dwDeviceClassSize	This field is the size, in bytes, of a NULL-terminated string (the size includes the NULL) that indicates the device class of the device whose configuration is specified in DeviceConfig. Valid device class strings are the same as those specified for the lineGetID function().
dwDeviceClassOffset	This field is the offset from the beginning of LINECALLPARAMS of a NULL-terminated string (the size includes the NULL) that indicates the device class of the device whose configuration is specified in DeviceConfig. Valid device class strings are the same as those specified for the lineGetID() function.
dwDeviceConfigSize	This field is the number of bytes of the opaque configuration data structure pointed to by dwDevConfigOffset. This value is returned in the dwStringSize member in the VarString structure returned by lineGetDevConfig(). If the size is zero, the default device configuration is used. This allows the application to set the device configuration before the call is initiated.
dwDeviceConfigOffset	This field is the offset from the beginning of LINECALLPARAMS of the opaque configuration data structure pointed to by dwDevConfigOffset. This value is returned in the dwStringSize member in the VarString structure returned by lineGetDevConfig(). If the size is zero, the default device configuration is used. This allows the application to set the device configuration before the call is initiated.

Field	Meaning
dwCallDataSize	This field is the size, in bytes, of the application-modifiable call data to be initially attached to the call.
dwCallDataOffset	This field is the offset from the beginning of LINECALLPARAMS of the application-settable call data to be initially attached to the call.
dwNoAnswerTimeout	This field is the number of seconds, after the completion of dialing, that the call should be allowed to wait in the PROCEEDING or RINGBACK states before it is automatically abandoned by the service provider with a LINECALLSTATE_DISCONNECTED and LINEDISCONNECTMODE_NOANSWER. A value of zero indicates that the application does not desire automatic call abandonment.
dwCallingPartyIDSize	This field is the size, in bytes, of a NULL-terminated string (the size includes the NULL) that specifies the identity of the party placing the call. If the content of the identifier is acceptable and a path is available, the service provider passes the identifier along to the called party to indicate the identity of the calling party.
dwCallingPartyIDOffset	This field is the offset from the beginning of LINECALLPARAMS of a NULL-terminated string (the size includes the NULL) that specifies the identity of the party placing the call. If the content of the identifier is acceptable and a path is available, the service provider passes the identifier along to the called party to indicate the identity of the calling party.
dwAddressType	This field is the address type used for the call. This member of the structure is available only if the negotiated TAPI version is 3.0 or higher.

LINECALLPARAMFLAGS_ Constants

The LINECALLPARAMFLAGS_ constants are defined in Table 10-3. They describe various status flags about a call.

Table 10-3: LINECALLPARAMFLAGS_ constants

Constant	Meaning
LINECALLPARAMFLAGS_ BLOCKID	This constant indicates that the originator identity should be concealed (block caller ID).
LINECALLPARAMFLAGS_ DESTOFFHOOK	This constant indicates that the called party's phone should be automatically taken offhook.
LINECALLPARAMFLAGS_IDLE	This constant indicates that the call should be originated on an idle call appearance and not join a call in progress. When using the lineMakeCall() function, if the LINECALLPARAMFLAGS_IDLE value is not set and there is an existing call on the line, the function breaks into the existing call if necessary to make the new call. If there is no existing call, the function makes the new call as specified.
LINECALLPARAMFLAGS_ NOHOLDCONFERENCE	This constant is used only in conjunction with lineSetupConference() and linePrepareAddToConference(). The address to be added to a conference with the current call is specified in the TargetAddress member in LINECALLPARAMS. The consultation call does not physically draw dial tone from the switch but will progress through various call establishment states (for example, dialing and proceeding). When the consultation call reaches the connected state, the conference is automatically established; the original call, which had remained in the connected state, enters the conferenced state; the consultation call enters the conferenced state; and the hConfCall enters the connected state.

Constant	Meaning
LINECALLPARAMFLAGS_ NOHOLDCONFERENCE (cont.)	If the consultation call fails (enters the disconnected state followed by idle), the hConfCall also enters the idle state, and the original call (which may have been an existing conference, in the case of linePrepareAddToConference()) remains in the connected state. The original party (or parties) never perceives the call as having gone onhold. This feature is often used to add a supervisor to an ACD agent call when necessary to monitor interactions with an irate caller.
LINECALLPARAMFLAGS_ ONESTEPTRANSFER	This constant is used only in conjunction with lineSetupTransfer(). It combines the operation of lineSetupTransfer() followed by lineDial() on the consultation call into a single step. The address to be dialed is specified in the TargetAddress member in LINECALLPARAMS. The original call is placed in onholdpending-tranfer state, as if lineSetupTransfer() were called normally, and the consultation call is established normally. The application must still call lineCompleteTransfer() to affect the transfer. This feature is often used when invoking a transfer from a server over a third-party call control link because such links frequently do not support the normal two-step process.
LINECALLPARAMFLAGS_ ORIGOFFHOOK	This constant indicates that the originator's phone should be automatically taken offhook.
LINECALLPARAMFLAGS_ PREDICTIVEDIAL	This constant is used only when placing a call on an address with predictive dialing capability (LINEADDRCAPFLAGS_PREDICTIVEDIALER is on in the dwAddrCapFlags member in LINEADDRESSCAPS). The bit must be on to enable the enhanced call progress and/or media device monitoring capabilities of the device. If this bit is not on, the call will be placed without enhanced call progress or media type monitoring, and no automatic transfer will be initiated based on call state.
LINECALLPARAMFLAGS_ SECURE	This constant indicates that the call should be set up as secure.

function lineTranslateAddress TAPI.pas

Syntax

function lineTranslateAddress(hLineApp: HLINEAPP; dwDeviceID, dwAPIVersion: DWORD; lpszAddressIn: LPCSTR; dwCard, dwTranslateOptions: DWORD; lpTranslateOutput: PLineTranslateOutput): Longint; stdcall;

Description

This function translates the specified address into another format.

Parameters

hLineApp: The application handle (HLINEAPP) returned by lineInitializeEx(). If an application has not yet called the lineInitializeEx() function, it can set the *hLineApp* parameter to NULL.

dwDeviceID: A DWORD holding the device ID for the line device upon which the call is intended to be dialed, so variations in dialing procedures on different lines can be applied to the translation process.

dwAPIVersion: A DWORD indicating the highest version of TAPI supported by the application (not necessarily the value negotiated by lineNegotiate-APIVersion() on some particular line device).

lpszAddressIn: A pointer (LPCSTR) to a NULL-terminated ASCII string containing the address from which the information is to be extracted for translation. It must be in either the canonical address format or an arbitrary string of dialable digits (non-canonical). This parameter must not be NULL. If *lpszAddressIn* contains a subaddress, 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: A DWORD indicating the credit card to be used for dialing. This field is only valid if the CARDOVERRIDE bit is set in *dwTranslateOptions*. This field specifies the permanent ID of a card entry in the [Cards] section in the registry (as obtained from lineTranslateCaps()), which should be used instead of the PreferredCardID specified in the definition of the CurrentLocation. It does not cause the PreferredCardID parameter of the current location entry in the registry to be modified; the override applies only to the current translation operation. This field is ignored if the CARDOVERRIDE bit is not set in *dwTranslateOptions*.

dwTranslateOptions: A DWORD indicating the associated operations to be performed prior to the translation of the address into a dialable string. This parameter uses the LINETRANSLATEOPTION_ constants explained in Table 10-4.

lpTranslateOutput: A pointer (PLineTranslateOutput) to an application-allocated memory area to contain the output of the translation operation of type LINETRANSLATEOUTPUT. Before you call lineTranslateAddress(), you should set the *dwTotalSize* field of this structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BAD-DEVICEID, LINEERR_INVALPOINTER, LINEERR_INCOMPATIBLE-APIVERSION, LINEERR_NODRIVER, LINEERR_INIFILECORRUPT, LINEERR_NOMEM, LINEERR_INVALADDRESS, LINEERR_OPERATION-FAILED, LINEERR_INVALAPPHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALCARD, LINEERR_STRUCTURETOOSMALL, and LINEERR_INVALPARAM.

See Also

LINECALLPARAMS, lineInitializeEx, lineNegotiateAPIVersion, LINE-TRANSLATEOUTPUT

Example

Listing 10-4 shows how to call the lineTranslateAddress() function.

Listing 10-4: Calling the lineTranslateAddress() function

```
procedure TTapiInterface.ShowLineTranslateDialog(APhoneNum : string; AHandle :
   THandle);
var
TempNumber : string;
begin
  TempNumber := '+1' + Copy(APhoneNum, 2, Length(APhoneNum)-1);
  if  FDev<0 then
  TapiResult := lineTranslateDialogA(fLineApp, 0, APIVersion, AHandle,
        LPCStr(TempNumber))
  else
  TapiResult := lineTranslateDialogA(fLineApp, FDev, APIVersion, AHandle,
        LPCStr(TempNumber));
  if TapiResult<>0 then
    ShowMessage('Could not show Line Translate Dialog Box')
  else
   lineTranslateAddress(fLineApp, FDev, APIVersion, PChar(fPhoneNumber), 0,
         LineTranslateOptions, @FTranslateOutput);
end;
```

Table 10-4: LINETRANSLATEOPTION_ constants used in the lineTranslateAddress() function's dwTranslateOptions parameter

Constant	Meaning
LINETRANSLATEOPTION_ CARDOVERRIDE	If this bit is set, dwCard specifies the permanent ID of a card entry in the [Cards] section in the registry (as obtained from lineTranslateCaps()), which should be used instead of the PreferredCardID specified in the definition of the CurrentLocation. It does not cause the PreferredCardID parameter of the current location entry in the registry to be modified; the override applies only to the current translation operation. The dwCard field is ignored if the CARDOVERRIDE bit is not set.
LINETRANSLATEOPTION_ CANCELCALLWAITING	If a Cancel Call Waiting string is defined for the location, setting this bit will cause that string to be inserted at the beginning of the dialable string. This is commonly used by data modem and fax applications to prevent interruption of calls by call waiting beeps. If no Cancel Call Waiting string is defined for the location, this bit has no effect. Note that applications using this bit are advised to also set the LINECALLPARAMFLAGS_SECURE bit in the dwCallParamFlags field of the LINECALLPARAMS structure passed in to lineMakeCall() through the lpCallParams parameter, so if the line device uses a mechanism other than dialable digits to suppress call interrupts, that mechanism will be invoked.
LINETRANSLATEOPTION_ FORCELOCAL	If the number is local but would have been translated as a long distance call (LINETRANSLATERESULT_INTOLLLIST bit set in the LINETRANSLATEOUTPUT structure), this option will force it to be translated as local. This is a temporary override of the toll list setting.
LINETRANSLATEOPTION_ FORCELD	If the address could potentially have been a toll call but would have been translated as a local call (LINETRANSLATERESULT_NOTINTOLLLIST bit set in the LINETRANSLATEOUTPUT structure), this option will force it to be translated as long distance. This is a temporary override of the toll list setting.

structure LINETRANSLATEOUTPUT TAPI.pas

The LINETRANSLATEOUTPUT structure describes the result of an address translation. It does not support extensions. It is defined as follows in TAPI.pas:

```
PLineTranslateOutput = ^TLineTranslateOutput;
linetranslateoutput_tag = packed record
  dwTotalSize,
  dwNeededSize,
  dwUsedSize,
  dwDialableStringSize,
  dwDialableStringOffset,
  dwDisplayableStringSize,
  dwDisplayableStringOffset,
  dwCurrentCountry,
  dwDestCountry,
  dwTranslateResults: DWORD;
end;
TLineTranslateOutput = linetranslateoutput_tag;
LINETRANSLATEOUTPUT  = linetranslateoutput_tag;
```

These fields are described in Table 10-5.

Table 10-5: Fields of the LINETRANSLATEOUTPUT structure

Field	Meaning
dwTotalSize	This field indicates the total size in bytes allocated to this data structure.
dwNeededSize	This field indicates the size in bytes for this data structure that is needed to hold all the returned information.
dwUsedSize	This field indicates the size in bytes of the portion of this data structure that contains useful information.
dwDialableStringSize	This field indicates the size in bytes of the NULL-terminated ASCII string that contains the translated output that can be passed to the lineMakeCall(), lineDial(), or other function requiring a dialable string. The output is always a NULL-terminated ASCII string (NULL is accounted for in Size). Ancillary fields, such as name and subaddress, are included in this output string if they were in the input string. This string may contain private information, such as calling card numbers. It should not be displayed to the user, in order to prevent inadvertent visibility to unauthorized persons.
dwDialableStringOffset	This field indicates the offset, in bytes, to the NULL-terminated ASCII string that contains the translated output that can be passed to the lineMakeCall(), lineDial(), or other function requiring a dialable string. The output is always a NULL-terminated ASCII string (NULL is accounted for in size). Ancillary fields, such as name and subaddress, are included in this output string if they were in the input string. This string may contain private information, such as calling card numbers. It should not be displayed to the user, in order to prevent inadvertent visibility to unauthorized persons.
dwDisplayableStringSize	This field indicates the translated output that can be displayed to the user for confirmation. It will be identical to DialableString, except calling card digits will be replaced with the "friendly name" of the card enclosed within bracket characters (for example, "[AT&T Card]"), and ancillary fields, such as name and subaddress, will be removed. It should normally be safe to display this string in call-status dialog boxes without exposing private information to unauthorized persons. This information is also appropriate to include in call logs.

Field	Meaning
dwDisplayableStringOffset	This field indicates the offset, in bytes, to the NULL-terminated ASCII string that contains the translated output that can be displayed to the user for confirmation. It will be identical to DialableString, except calling card digits will be replaced with the "friendly name" of the card enclosed within bracket characters (for example, "[AT&T Card]"), and ancillary fields, such as name and subaddress, will be removed. It should normally be safe to display this string in call-status dialog boxes without exposing private information to unauthorized persons. This information is also appropriate to include in call logs.
dwCurrentCountry	This field contains the the country code configured in CurrentLocation. This value may be used to control the display by the application of certain user interface elements, local call progress tone detection, and other purposes.
dwDestCountry	This field contains the destination country code of the translated address. This value may be passed to the dwCountryCode parameter of lineMakeCall() and other dialing functions (so that the call progress tones of the destination country, such as a busy signal, will be properly detected). This field is set to zero if the destination address passed to lineTranslateAddress() is not in canonical format.
dwTranslateResults	This field indicates the information derived from the translation process, which may assist the application in presenting user-interface elements. This field uses the LINETRANSLATERESULT_ constants described in Table 10-6.

Table 10-6: LINETRANSLATERESULT_ constants used with the dwTranslateResults field of the LINETRANSLATEOUTPUT structure

Constant	Meaning
LINETRANSLATERESULT_ CANONICAL	This constant indicates that the input string was in valid canonical format.
LINETRANSLATERESULT_ INTERNATIONAL	If this bit is on, the call is being treated as an international call (country code specified in the destination address is different from the country code specified for the CurrentLocation).
LINETRANSLATERESULT_ LONGDISTANCE	If this bit is on, the call is being treated as a long distance call (country code specified in the destination address is the same, but area code is different from those specified for the CurrentLocation).
LINETRANSLATERESULT_ LOCAL	If this bit is on, the call is being treated as a local call (country code and area code specified in the destination address are the same as those specified for the CurrentLocation).
LINETRANSLATERESULT_ INTOLLLIST	If this bit is on, the local call is being dialed as long distance because the country has toll calling and the prefix appears in the TollPrefixList of the CurrentLocation.
LINETRANSLATERESULT_ NOTINTOLLLIST	If this bit is on, the country supports toll calling, but the prefix does not appear in the TollPrefixList, so the call is dialed as a local call. Note that if both INTOLLIST and NOTINTOLLIST are off, the current country does not support toll prefixes and user-interface elements related to toll prefixes should not be presented to the user; if either such bit is on, the country does support toll lists, and the related user-interface elements should be enabled.
LINETRANSLATERESULT_ DIALBILLING	This constant indicates that the returned address contains a "$."
LINETRANSLATERESULT_ DIALQUIET	This constant indicates that the returned address contains a "@."
LINETRANSLATERESULT_ DIALDIALTONE	This constant indicates that the returned address contains a "W."

Constant	Meaning
LINETRANSLATERESULT_ DIALPROMPT	This constant indicates that the returned address contains a "?."

function lineTranslateDialog TAPI.pas

Syntax

function lineTranslateDialog(hLineApp: HLINEAPP; dwDeviceID, dwAPIVersion: DWORD; hwndOwner: HWND; lpszAddressIn: LPCSTR): Longint; stdcall; // TAPI v1.4

Description

This function displays an application modal dialog that allows the user to change the current location, adjust location and calling card parameters, and see the effect on a phone number about to be dialed.

Parameters

hLineApp: The application handle (HLINEAPP) returned by lineInitializeEx(). If an application has not yet called the lineInitializeEx() function, it can set the *hLineApp* parameter to NULL.

dwDeviceID: A DWORD indicating the device ID for the line device upon which the call is intended to be dialed, so variations in dialing procedures on different lines can be applied to the translation process

dwAPIVersion: A DWORD indicating the highest version of TAPI supported by the application (not necessarily the value negotiated by lineNegotiateAPIVersion() on the line device indicated by *dwDeviceID*)

hwndOwner: A handle (HWND) to a window to which the dialog is to be attached. It can be a NULL value to indicate that any window created during the function should have no owner window.

lpszAddressIn: A pointer (LPCSTR) to a NULL-terminated ASCII string containing a phone number that will be used in the lower portion of the dialog to show the effect of the user's changes to the location parameters. The number must be in canonical format; if non-canonical, the phone number portion of the dialog will not be displayed. This pointer can be left NULL, in which case the phone number portion of the dialog will not be displayed. If *lpszAddressIn* contains a subaddress, name field, or additional addresses separated from the first address by ASCII CR and LF characters, only the first address is used in the dialog.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_INVALPARAM, LINEERR_INCOMPATIBLEAPIVERSION, LINEERR_INVALPOINTER, LINEERR_INIFILECORRUPT, LINEERR_NODRIVER, LINEERR_INUSE, LINEERR_NOMEM, LINEERR_INVALADDRESS, LINEERR_INVALAPPHANDLE, and LINEERR_OPERATIONFAILED.

See Also

lineGetTranslateCaps, lineInitializeEx, lineNegotiateAPIVersion, lineTranslateAddress

Example

See Listing 10-4.

Summary

In this chapter we have explored the various means of placing calls with TAPI: high level and low level. We have also discussed the various types of addresses or phone numbers with which TAPI works. Equally important in a full-featured Call Manager application is accepting incoming phone calls. That is the topic of our next chapter, "Accepting Incoming Calls."

Chapter 11

Accepting Incoming Calls

In the last chapter we explored the high-level and low-level means of placing calls with TAPI along with the various types of addresses or phone numbers with which TAPI works. Equally important in a full-featured Call Manager application is the ability to accept incoming phone calls. We'll concentrate on that topic in this chapter. We'll begin by discussing the process that TAPI must go through in finding the right application to handle an incoming call. Then we'll discuss the process of answering such a call. Finally, we'll examine all of the functions that are used to support these activities.

Finding the Right Application

When a call arrives, we have no way of knowing what kind of call it may be (voice, data, or something else). As we discussed in Chapter 8, TAPI uses media modes to differentiate different kinds of calls. These media modes have a considerable impact on how TAPI deals with incoming calls.

When a call arrives, information about that call, including its media mode, is contained in its LINECALLINFO structure. If just one media mode bit (excluding the *unknown* bit) has been set in the structure's *dwMediaMode* field, TAPI will attempt to find a suitable telephony application to handle it. In doing so, it will follow a consistent procedure based on the current state of the system and information saved by the user in the registry. These steps are summarized succinctly in Figure 11-1. As enumerated in the TAPI Help file, these are the steps it takes to find and possibly launch an application to handle an incoming call:

1. The service provider notifies the TAPI dynamic-link library that a call is arriving.
2. TAPI examines the information in the HandoffPriorities section of the registry to discover which applications, if any, are interested in handling calls having this one's media mode. Often, this information is exposed through a Preferences option in an application's user interface.
3. TAPI considers the first appropriate application listed, reading left to right, as the highest priority application. If that application is currently running

and has the arriving call's line open for the requested media mode, it gets ownership of the call. If it is not running or does not have that line open, TAPI again uses the information in the registry to find an interested application in the correct state, to which it gives the call.

4. If none of the applications listed in the registry are in the proper state, TAPI looks for other applications that are currently executing and that also have the particular line open for that media mode (though they are not listed in the registry). The relative priority among these unlisted applications is arbitrary and not necessarily associated with the sequence in which they were launched or opened on the line.

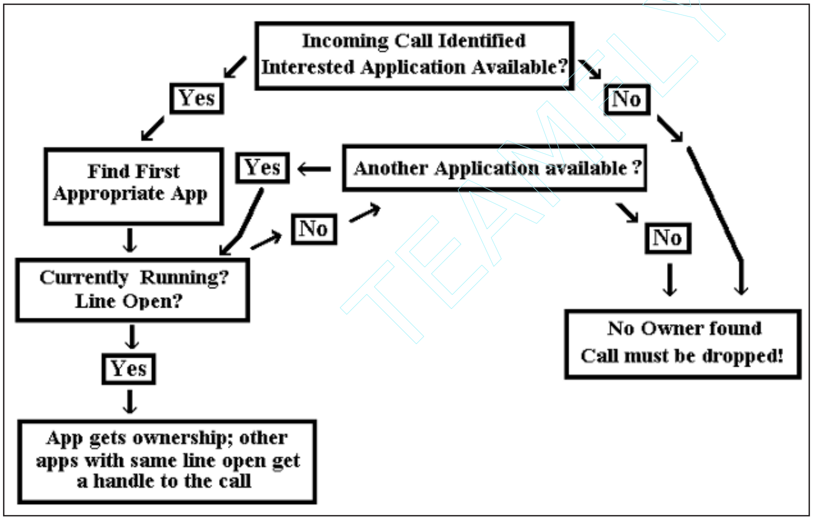


Figure 11-1: Finding and launching an application to handle an incoming call

If your application has the particular line open for monitoring, it also will receive a handle to all of the calls on that line. Because of that, your application could step up, claim ownership of the call by invoking the lineSetCallPrivilege() function, and go ahead and answer it. (We discuss this function and provide Delphi code in the reference section of this chapter.) Be aware that this behavior could result in problems in handling the call and the TAPI Help file discourages it.

If no application becomes an owner of the call, TAPI will eventually drop the call, but this will happen only if no appropriate owner can be found and the call state is neither idle nor offering. Of course, the calling party can also drop the call. (On an ISDN network, this event becomes known when a "call-disconnect" frame is received.) If the call is not explicitly dropped, it can go into an idle state after the expiration of a timeout. Such a timeout is usually based on the absence of ringing. (The service provider would need to assume that the calling party has dropped the call and implemented the timeout.) Because there were no applications that could take the call successfully, this situation usually means that the incoming call reached a wrong number.

Unknown Media Type

When a call is coming in, TAPI is constantly monitoring its progress. It checks the first LINE_CALLSTATE message delivered by the service provider to find out the type of media that is arriving. Sometimes, this is not indicated explicitly and the *unknown* media bit will be set. Here's how that works.

If TAPI discovers that the messages's *param3* LINEMEDIAMODE_UNKNOWN bit is set, it will begin a process of probing, as outlined in Figure 11-2. First, it will determine whether or not an application has already opened the specific line and is prepared to accept calls of the *unknown* media type. There are two possible cases: a suitable *unknown* application is running or one is not running. We'll examine each.

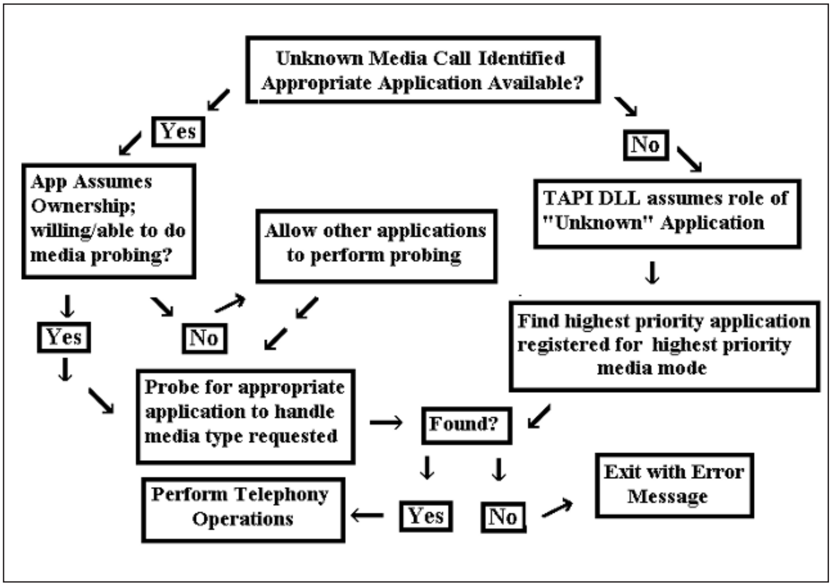


Figure 11-2: The probing process

In the first instance, at least one *unknown* application will be open and available on a line. In this scenario, the TAPI dynamic-link library will give an ownership handle for the incoming call to the highest priority *unknown* application. That application will receive a LINE_CALLSTATE message with *dwParam3* set to owner. As we indicated, it will also pass monitoring handles to the other applications that have the line open for monitoring.

This *unknown* application could then attempt to perform media determination itself or use the assistance of the other media-related applications. In the latter case, these other applications would perform probes for their supported media mode(s). The *unknown* application could also simply pass the call to another media application by calling the lineHandoff() function. The *unknown* application would want to examine LINECALLINFO's *dwMediaMode* field to determine the possible remaining media candidates. In doing so, it would select the highest priority media (see below) as the initial handoff target. It would call the lineHandoff() function, specifying the single highest priority destination media mode as the target.

In the second instance, if no *unknown* application has opened the line, the TAPI dynamic-link library itself will assume the role of an *unknown* application, passing an owner handle for the call to the highest priority application that is registered for the highest priority media mode whose flag is set in the LINECALLINFO structure's *dwMediaMode* field. TAPI will examine all of the appropriate applications in priority order until a suitable one is found.

What if no appropriate owner application can be found? In that case, the call will remain in the offering state until either a monitoring application becomes an owner by calling the lineSetCallPrivilege() function or until the call is abandoned by the calling party. In the latter case, it will simply transition to the idle state. After that, all monitoring applications will deallocate their handles to the call.

Prioritizing Media Modes

Having briefly examined the two general possibilities, we will now explore how TAPI prioritizes media modes. These media modes have a default order, beginning with the first one that will be tried during media type handoff to the last one that will be tried. The order is as follows:

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

If a handoff fails, the *unknown* application should clear that media mode flag in LINECALLINFO's *dwMediaMode* member and try the next one in the list. If the handoff indicates TARGETSELF, the current *unknown* application is itself the highest priority application for the media mode for which it was trying to hand off the call. Therefore, it should go ahead and do the probing itself.

If the handoff indicates SUCCESS, a different application is the highest priority application for the media mode for which the call was being handed off. The *unknown* application should deallocate the call handle or change its status to that of a monitor while the new owner takes control and proceeds with probing.

Responsibilities of the Receiving Application

A receiving application has certain responsibilities. Most importantly, it gains control of the call. If the probe is successful, it should set the correct media mode bit. If the probe fails, the application should clear the failed media mode bit in LINECALLINFO and hand the call off to the next highest priority application, which can give it a try. If no more media mode bits are set, the handoff will fail, since no suitable owner application could be found for the call.

In the end, the media mode may be identified through monitoring or successful probing, though the *unknown* bit may still be set in *dwMediaMode* in the data structure, LINECALLINFO. This situation is a bit fluid. The application that received the call cannot be absolutely sure that it is the highest priority application for the identified media mode. It is now the duty of that application to ensure that the call goes to the highest priority application. To do so, it must follow these steps:

1. It must call the lineSetMediaMode() function, which will write to the *dwMediaMode* field of the call, turning off the *unknown* bit and specifying the newly identified media mode bit.
2. It should call the lineHandoff() function to return the call to TAPI, which will assume the task of finding the highest priority application for that media mode.
3. As indicated above, if this application is itself the highest priority application for this media mode, it will receive a LINEERR_TARGETSELF return value (for the lineHandoff() function call). This tells the application: "No, you are already the highest priority application for that media mode; deal with it."

The application in question never loses control of the call, and it continues handling the call normally. If the call to the lineHandoff() function succeeds, there was a higher priority application for the identified media mode, and the application that called lineHandoff() should deallocate its handle or become a monitor, allowing the highest priority application to handle the call.

> **TIP:** Be aware of this: As long as the *unknown* bit remains set, a receiving application will still not know that the highest priority media mode is present on the call. Therefore, it must probe for it. It will consider the media mode to be present only if the *unknown* bit is off. Only then can it relate to the call as one of that media mode.

The TAPI Help file recommends that *unknown* applications use default priorities when probing for applications to accept calls of unknown media modes. They point out that this protects human callers from hearing unpleasant fax or modem signals. Specifically, they recommend probing first for voice, which will occur automatically if an application follows the order stated in the default media mode list described earlier.

If your application will be probing for high-priority media modes, the TAPI Help file recommends turning media monitoring on. This feature, invoked by calling the lineMonitorMedia() function, will detect signals that indicate particular media. The Help file provides an interesting example in which one application may be playing an outgoing "leave a message" voice message, while at the same time an incoming call starts sending a fax "calling" tone after which it waits for a handshake. In order to not lose the fax call, the local application would need to monitor for this tone while playing the voice message. Determining the lower priority media (the fax call) while actively probing for the higher priority media (voice) is not only a safer method, it also helps prevent the loss of a call. It is quite efficient since it can shorten the probing process.

Media Application Duties

When a suitable media application has been located and the call has been given to that application, the latter must assume certain duties. If that application receives the call as a handoff target, it should first check LINECALLINFO's *dwMediaMode* bit flags. If it finds that only a single media mode flag is set, the call will be considered to be officially of that media mode, and the application can act accordingly.

As you may have guessed, if the *unknown* flag and other media mode flags are set, the media mode of the call will still be officially *unknown*, with the assumption that it could operate using one of the media modes for which a flag is set in LINECALLINFO. In this case, the application should next probe for the highest priority media mode. This continued probing can follow different directions. If more than one bit is set in LINECALLINFO and the call has not been answered, the application must call the lineAnswer() function to continue probing. On the other hand, if the call has already been answered, the application can continue probing without having to first answer the call.

If the probe succeeds (for either the highest priority media mode or for another one), the application should set LINECALLINFO's *dwMediaMode* field to the particular media mode that the probe recognized. If the actual media mode is this expected media mode, the application can handle it. Otherwise (if it identifies another media mode), it must first attempt to hand off the call in case it is not the highest priority application for the detected media mode.

If the probe fails, the application should clear the flag for that media mode in *dwMediaMode* in LINECALLINFO and hand the call off to the *unknown* application. It should also deallocate its call handle or revert back to monitoring the call. If an attempt to hand off the call to the *unknown* application fails, no *unknown* application is running. It is then the responsibility of the application that currently owns the call to attempt to hand it off to the next highest priority media mode (while leaving LINECALLINFO's *dwMediaMode unknown* bit turned on so the process may continue). If that handoff fails, the application should turn off that media bit and attempt the next higher priority bit, until the handoff succeeds or until all of the bits are off except for the *unknown* bit.

If none of the media modes were determined to be the actual one, only the *unknown* flag will remain set in LINECALLINFO's *dwMediaMode* when the media application attempts to hand the call off to *unknown*. The final call to lineHandoff() will fail if the application is the only remaining owner of the call. This failure informs the application that it should drop the call and then deallocate the call's handle. At this point, the call is abandoned. Of course, you should use the information available to inform the user of the failure and indicate the reasons for it.

Accepting an Incoming Call

Now that we have examined the process of dealing with media modes, especially the *unknown* media mode, we are ready to discuss the process of accepting a call. If you're writing a Call Manager application (as we do in the code accompanying this book), you'll want your users to be able to receive calls as well as place calls. After an application has properly opened a line device, it will be notified whenever a call arrives on that line.

To properly open a line to receive incoming calls, your application must register a privilege other than a privilege of none. It must also indicate a media mode. If your application has opened a line with LINECALLPRIVILEGE_MONITOR, it will receive a LINE_CALLSTATE message for every call that arrives on the line. If it has opened a line with LINECALLPRIVILEGE_OWNER, it will receive a LINE_CALLSTATE message only if it has become an owner of the call or is the target of a directed handoff. In this notification, TAPI will give the handoff receiving application a handle to the incoming call. That application will keep this handle until it deallocates the call.

Using the mechanism you set up when you initialize TAPI, Windows will inform applications of call arrivals and all other call-state events using the LINE_CALLSTATE message. This message provides the call handle, an application's privilege to the call, and the call's new state. The call state for an unanswered inbound call will always be offering. You can call the lineGetCallInfo() function to obtain information about an offering call before accepting it.

This function call will also cause the call information in the LINECALLINFO data structure to be updated. By knowing the call state and other information, your application can determine whether or not it needs to answer the call. You'll recall that we stressed the importance of this structure at the beginning of this chapter. What kind of call information is stored in the LINECALLINFO structure? Among other things, it includes the information shown in Table 11-1.

Table 11-1: Information stored in the LINECALLINFO structure

Information	Description
Bearer mode, rate	This is the bearer mode (voice, data) and data rate (in bits per second) of the call for digital data calls.
Media mode	The current media mode of the call. Unknown is the mode specified if this information is unknown, and the other set bits indicate which media modes might possibly exist on the call. For more information, see Multiple-Application Programming in the TAPI Help file.
Call origin	Indicates whether the call originated from an internal caller, an external caller, or an unknown caller.
Reason for the call	Describes why the call is occurring. Possible reasons include a direct call, a call transferred from another number, a busy call forwarded from another number, a call unconditionally forwarded from another number, a call picked up from another number, a call completion request, a callback reminder (the reason for the call will be given as unknown if this information is not known), user-to-user information sent by the remote station (ISDN), and so on.

In addition to the reasons for the call listed in Table 11-1, there are several involving call identifiers, as shown in Table 11-2.

Table 11-2: Call reasons involving call identifiers

ID Type	Description
Caller-ID	Identifies the originating party of the call. This can be in a variety of (name or number) formats, determined by what the switch or network provides.
Called-ID	Identifies the party originally dialed by the caller
Connected-ID	Identifies the party to which the call was actually connected. This may be different from the called party if the call was diverted.
Redirection-ID	Identifies to the caller the number toward which diversion was invoked
Redirecting-ID	Identifies to the diverted-to user the party from which diversion was invoked

The LINE_CALLSTATE message also has the duty of notifying monitoring applications that a call has been established by other applications or manually by the user. One example, given in the TAPI Help file, concerns an attached phone device (if the telephony hardware and the service provider support monitoring of actions on external equipment). The call state of such calls reflects the actual state of the call, as follows: An inbound call for which ownership is given to another application is indicated to the monitor applications as initially being in the offering state. An outbound call placed by another application would normally first appear to the monitoring applications in the dial-tone state.

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

When it comes to providing information about a call, some telephony environments can provide information when the call is initially offered; others cannot. For example, if caller ID is not provided by the network until after the second ring, caller ID will be unknown at the time the call is first offered. When it does become known shortly thereafter, a LINE_CALLINFO message will notify the application about the change in the party ID information of the call.

Now we can discuss the specific details of accepting and answering calls. As before, we'll see differences on different types of networks. On a POTS network, the only reason for an application to call lineAccept() would be to inform other applications that it has accepted responsibility to present the call to the user. The lineAccept() function is discussed in detail later in this chapter. Similarly, on an ISDN line, the effect of accepting a call is simply to make other applications aware that some application has accepted responsibility for handling the call.

On an ISDN network, accepting a call also involves informing the switch that an application will present the call to the user. This is accomplished by alerting the user, either by ringing or by popping up a dialog box on the computer. If the LINEADDRCAPFLAGS_ACCEPTTOALERT bit is set, an application must call lineAccept() for the call or the call will not ring.

NOTE: If an application fails to call the lineAccept() function quickly enough (the timeout may be as short as three seconds on some ISDN networks), the network will assume that the station is powered off or disconnected and act accordingly. Under these circumstances, it will likely either deflect the call (if Forward is on and No Answer is activated) or send a disconnect message to the calling station.

Be aware that these terms are quite specific. Accepting a call is not the same as answering a call. With POTS, answering a call simply means to go offhook. On an ISDN line, it means to tell the switch to place the call in a connected state. Prior to answering, there is no physical connection for the call between the switch and the destination, though the call is connected from the caller to the switch.

You can program your telephony applications to wait a minimum number of rings before abandoning a call or answering it automatically to accept voice mail. You should use the lineGetNumRings() function to determine the number of times an inbound call on the given address should ring before the call is to be answered. Waiting a certain number of rings allows callers to be spared the charge of a call connection if it seems that the call will not be answered by the desired party (usually a person). This feature is sometimes called *toll-saver support*. Applications can use the functions lineGetNumRings() and lineSetNumRings() in combination to provide a mechanism to support toll-saver features for multiple independent applications. These two functions are discussed in the reference section at the end of this chapter.

Any application that receives a handle for a call in the offering state, along with a LINE_LINEDEVSTATE ringing message, should wait a number of rings equal to the number returned by lineGetNumRings() before answering the call in order to honor the toll-saver settings across all applications. The function lineGetNumRings() will return the minimum number of rings an application has specified with the function lineSetNumRings(). Because this number may vary dynamically, an application should call lineGetNumRings() each time it has the option to answer a call. In other words, it should check the number of rings whenever it is the owner of a call that is still in the offering state. A separate LINE_LINEDEVSTATE ringing message will be sent to an application for each ring cycle.

If the service provider is set to auto-answer calls, it will answer after a certain number of rings. Service providers do not have access to the minimum ring information established by lineSetNumRings() and will therefore make their own determination of when to automatically answer an incoming call. When a call has been answered by a service provider, it will be delivered initially to the owning application. Since it will already be in the connected state, an application need not be concerned with counting rings or answering the call. In our sample Call Manager application, we give the user the opportunity to set how many rings should transpire before automatically answering and possibly playing a recorded message.

You might wonder how an application takes ownership of a call. In general, when one application learns that another application wants ownership of a call, it will simply relinquish ownership of the call to that other application. Although there can be many co-owners of a call, multiple ownership should be a transitory state.

There is one case in which it is valid for an application to actively take ownership of a call owned by another application—when it is instructed to do so by the user interacting with a user interface. This would be appropriate, for example, if the user wanted to end a voice conversation, but keep the line open, transferring the call to a fax application to send a fax. Of course, the fax application would take ownership from the previous owner of the original application that had controlled the voice call. The TAPI Help file also describes a less polite method, whereby an application could forcibly become owner of a call.

Be aware that there are potential pitfalls. For example, there is no way to shield a call from another application's attempt to become its owner. Generally, there isn't any reason to do so. Once an application is informed that another application has become an owner, it should do the responsible thing—abandon its activities on the call and relinquish ownership. This makes sense, since such changes in ownership are almost always the result of explicit actions by the user.

At some point, an application will be finished with a call and want to relinquish it. An application can relinquish ownership of a call by calling lineSetCallPrivilege() to change its status to that of a monitor application. Or, it could simply call the lineDeallocateCall() function to indicate that it has no further interest in the call. Be aware, however, that you cannot give up your responsibilities if no other application is willing to assume them. If an application happens to be the sole owner of the call and cannot hand off ownership to another application, TAPI will not permit it to change to being a monitor or to deallocate its call handle. In this situation, the application has no choice but to drop the call.

Now we're ready to consider how to handle incoming calls and line privileges. The following is implicit within what we have discussed so far: An application cannot refuse ownership of a call for which it receives an owner handle. An application's relationship to a call—whether the application will receive owner or monitor privileges to the call—will be decided before the call arrives, when the application opens the line on which the call is established by the remote caller. Next we'll examine the details.

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

When an incoming call has been offered to an application and the latter is an owner of the call, the application can then answer the call using the lineAnswer() function. Once the application has answered the call, the latter's call state will typically transition to "connected," at which time information can be exchanged over the line.

We've laid the groundwork for dealing with incoming calls. Now we'll examine some of the more subtle details. Sometimes, you may have a situation in which multiple telephony applications are capable of running simultaneously. In this situation, TAPI must be able to identify an appropriate application to become the initial owner of each incoming call. In general, incoming calls reach their destination, or target application, in a process involving two or three steps: First, the service provider learns that a new call has arrived and passes it to the TAPI dynamic-link library. Second, TAPI initiates the process to give the call to an appropriate application. Finally, the applications themselves sometimes need to conduct a probing process, as we discussed in some detail above. In such a probing process, the call may be handed off between applications one or more times.

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

Sometimes, you may wish to log call information. An application can call the function lineGetCallInfo() in order to obtain information about a call. Although this function fills the LINECALLINFO structure with a large amount of data, applications may need to maintain additional items, such as the start and stop time of the call. You can also include call state information in a log, as we do in the Call Manager application. Those states are diagrammed in Figure 11-3.

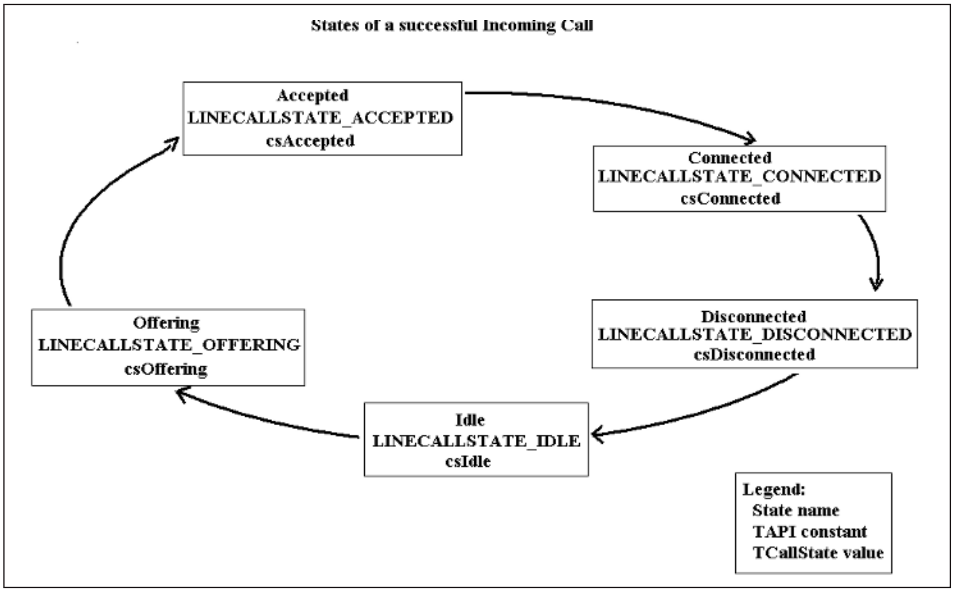


Figure 11-3: States of a successful incoming call: the probing process

In developing your applications, you might consider the TAPI Help file's suggestions:

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

Ending a Call

Besides placing and accepting calls, another very common task, of course, is ending a call. When the call has ended, your application will receive a LINE_CALLSTATE message that will inform it that the state of a line device has changed. Generally, this means that a remote disconnect has occurred. Of course, you may also disconnect a call at the local end (it "goes on-hook") by calling the lineDrop() function. Alternatively, your application itself may choose to end the call by invoking the lineDrop() function before it receives a remote-disconnect message. In previous chapters, we discussed closing a line and shutting down TAPI. It is important that you perform these tasks in a specific order. Here are the steps that you might use to end a call, close the line, and shut down TAPI (assuming the user wishes to end all telephony activity):

1. An application could first call lineDrop(), which will place the call in the idle state. The call will still exist with the application maintaining its handle to the call. If it needs to, the application could still examine the call information record.
2. An application might then call lineDeallocateCall() to release the call handle for the finished call. After this, the call will no longer exist.
3. An application is now ready to call lineClose() to close the line; it should do this only if it expects no more further calls on that line. After this, there will be no more incoming or outgoing calls on that line.
4. Before closing, an application should call lineShutdown() to end its use of TAPI's functions for the current session.

As we discussed in Chapter 8, when your application is finished using a line device, you should close the device by calling lineClose() for the line device handle. As we stated there, after you've closed the line, your application's handle for that line device will no longer be valid.

In the next section, we provide a reference for the additional basic TAPI functions, especially those that support dealing with incoming calls. We also discuss the structures and constants that are used with these functions. Each function reference includes Delphi code from our TAPI class. Every function is used in one of the sample applications.

Reference for Additional Basic TAPI Functions

function lineAccept　　*Tapi.pas*

Syntax

```
function lineAccept(hCall: HCALL; lpsUserUserInfo: LPCSTR; dwSize: DWORD):
Longint; stdcall;
```

Description

This function accepts the specified offered call. It may optionally send the specified user-to-user information to the calling party. It is typically used in telephony environments like Integrated Services Digital Network (ISDN) that support an alerting process associated with incoming calls independent from the initial offering of the call; it also can be used with non-ISDN systems. When a call arrives, it is initially in the offering state. During that brief time period the application may reject the call by using the lineDrop() function, redirect it by using the lineRedirect() function, answer it by using the lineAnswer() function, or accept it by using this function (lineAccept()). After a call has been accepted by an application, its call state typically transitions to that of accepted. Applications are alerted of state changes by the LINE_LINEDEVSTATE message. An application may also send user-to-user information when a call has been accepted.

Parameters

hCall: A handle to the call to be accepted or rejected. The application must be an owner of the call. The call state of *hCall* must be offering.

lpsUserUserInfo: A pointer (of type LPCSTR) 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 set to NIL if no user-to-user information is to be sent. User-to-user information is only sent if supported by the underlying network (see the LINEDEVCAPS structure). The protocol discriminator field for the user-to-user information, if required, should appear as the first byte of the buffer pointed to by *lpsUserUserInfo*. You must account for the added size in the *dwSize* parameter.

dwSize: A DWORD that holds the size (in bytes) of the user-to-user information in *lpsUserUserInfo*. If *lpsUserUserInfo* is set to NIL, no user-to-user information is sent to the calling party and *dwSize* will be ignored.

Return Value

This function returns a positive request ID if the function will be completed asynchronously or a negative error number if an error occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message will be set to zero if the function is successful or to a negative error number if an error occurred. Possible error return values are LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALCALLSTATE, LINEERR_OPERATIONUNAVAIL, LINEERR_NOTOWNER, LINEERR_UNINITIALIZED, LINEERR_INVALPOINTER, LINEERR_OPERATIONFAILED, LINEERR_NOMEM, and LINEERR_USERUSERINFOTOOBIG.

See Also

LINE_REPLY, lineAnswer, LINEDEVCAPS, lineDrop, lineRedirect (in TAPI Help file)

Example

See the use of this function in the ALineCallBack() procedure given in the "Line Callback" section of Chapter 9.

function lineAnswer TAPI.pas

Syntax

function lineAnswer(hCall: HCALL; lpsUserUserInfo: LPCSTR; dwSize: DWORD): Longint; stdcall;

Description

This function answers the specified offering call.

Parameters

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

lpsUserUserInfo: A pointer to a string (LPCSTR) that contains user-to-user information to be sent to the remote party at the time of answering the call. This pointer can be set to NIL if no user-to-user information is to be sent. User-to-user information is only sent if the functionality is supported by the underlying network (see LINEDEVCAPS). The protocol discriminator field for the user-to-user information, if required, should appear as the first byte of the buffer pointed to by *lpsUserUserInfo* and must be accounted for in the *dwSize* parameter.

dwSize: A DWORD indicating the size, in bytes, of the user-to-user information in *lpsUserUserInfo*. If *lpsUserUserInfo* is NIL, no user-to-user information is sent to the calling party and *dwSize* will be ignored.

Return Value

Returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INUSE, LINEERR_OPERATIONUNAVAIL, LINEERR_INVAL-CALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALL-STATE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, LINEERR_NOMEM, LINEERR_USERUSER-INFOTOOBIG, and LINEERR_NOTOWNER.

See Also

LINE_CALLSTATE, LINE_REPLY, LINEDEVCAPS

Example

The code fragment in Listing 11-1, responding to a LINECALLSTATE_ ACCEPTED message in the line TLineCallback() function (see Chapter 9), answers an incoming call.

Listing 11-1: Answering an incoming call

```
        LINECALLSTATE_ACCEPTED:
          begin
            TapiInterface.CallState := csAccepted;
            TapiInterface.OnSendTapiMessage(
              'The call was offering and has been accepted.');
            if TapiInterface.App.MessageBox('Do you want to accept this call?',
               'Incoming Phone Call', MB_OKCANCEL + MB_ICONQUESTION)=IDOK  then
               lineAnswer(TapiInterface.CurrentCall, Nil, 0);
          end;
```

function lineDeallocateCall   TAPI.pas

Syntax

function lineDeallocateCall(hCall: HCALL): Longint; stdcall;

Description

This function deallocates the specified call handle.

Parameter

hCall: The call handle (HCALL) 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 will no longer be valid after it has been deallocated.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSTATE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, and LINEERR_UNINITIALIZED.

See Also

LINE_REPLY, lineDrop, lineShutdown

Example

Listing 11-2 shows how to use lineDeallocateCall() in the process of completely hanging up a call and freeing resources.

Listing 11-2: Hanging up a call completely

```
function TTapiInterface.HangUp: Boolean;
begin
  TAPIResult := lineDrop(FCall, Nil, 0);
  result := TapiResult>0;
  If NOT Result then
    ReportError(TapiResult)
  else
    OnSendTapiMessage('line drop successful');
```

```
  if result then
  begin
    TAPIResult := lineDeallocateCall(FCall);
    result := TapiResult=0;
    If NOT Result then
      ReportError(TapiResult)
    else
      OnSendTapiMessage('line deallocation successful');
  end;
end;
```

function lineDrop TAPI.pas

Syntax

function lineDrop(hCall: HCALL; lpsUserUserInfo: LPCSTR; dwSize: DWORD): Longint; stdcall;

Description

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.

Parameters

hCall: A handle (HCALL) to the call to be dropped. The application must be an owner of the call. The call state of *hCall* can be any state except idle.

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

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

Return Value

This function returns a positive request ID if the function will be completed asynchronously or a negative error number if an error has occurred. The *dwParam2* parameter of the corresponding LINE_REPLY message is zero if the function is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONUNAVAIL, LINEERR_NOMEM, LINEERR_OPERATIONFAILED, LINEERR_NOTOWNER, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_USERUSERINFOTOOBIG, LINEERR_INVALCALLSTATE, and LINEERR_UNINITIALIZED.

See Also

LINE_CALLSTATE, LINE_REPLY, LINEDEVCAPS

Example

See Listing 11-2.

function lineGetCallInfo TAPI.pas

Syntax

function lineGetCallInfo(hCall: HCALL; lpCallInfo: PLineCallInfo): Longint; stdcall;

Description

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

Parameters

hCall: A handle (HCALL) to the call to be queried. The call state of *hCall* can be any state.

lpCallInfo: A pointer (PLineCallInfo) to a variably sized data structure of type LINECALLINFO. If the request is succesfully completed, this structure is filled with call-related information. Before you call lineGetCallInfo(), you should set the *dwTotalSize* field of the LINECALLINFO structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALCALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONFAILED, and LINEERR_OPERATIONUNAVAIL.

See Also

LINE_CALLINFO, LINE_CALLSTATE, LINECALLINFO

Example

Listing 11-3 shows how to retrieve call information.

Listing 11-3: Retrieving call information

```
function TTapiInterface.GetCallInfo: boolean;
begin
  if CallState <> csConnected then
  begin
    ShowMessage('Call must be connected to get address status');
    result := false;
    exit;
  end;
```

```
  if fLineCallInfo=Nil then
    fLineCallInfo := AllocMem(SizeOf(TLineCallInfo)+1000);
  fLineCallInfo.dwTotalSize := SizeOf(LineCallInfo)+1000;
  TapiResult := lineGetCallInfo(fCall, fLineCallInfo);
  result := TapiResult=0;
  if NOT result then ReportError(TapiResult);
end;
```

structure LINECALLINFO   TAPI.pas

The huge LINECALLINFO structure contains information about a call. This information remains relatively fixed for the duration of a particular call. A number of functions use LINECALLINFO. The structure is returned by the lineGetCallInfo() function and the TSPI_lineGetCallInfo() function. If a part of the structure does change, a LINE_CALLINFO message is sent to the application indicating which information item has changed. Dynamically changing information about a call, such as call progress status, is available in the LINECALLSTATUS structure, returned by a call to the lineGetCallStatus() function.

If your application uses device-specific extensions, you should use the DevSpecific (*dwDevSpecificSize* and *dwDevSpecificOffset*) variably sized area of this data structure. The members *dwCallTreatment* through *dwReceivingFlowspecOffset* are available only to applications that open the line device with an API version of 2.0 or later.

NOTE: The preferred format for the specification of the contents of the *dwCallID* field and the other five similar fields (*dwCallerIDFlag*, *dwCallerIDSize*, *dwCallerIDOffset*, *dwCallerIDNameSize*, and *dwCallerIDNameOffset*) is the TAPI canonical number format that we discussed in detail in Chapter 9.

For example, a ICLID of "4258828080" received from the switch should be converted to "+1 (425) 8828080" before being placed in the LINECALLINFO structure. This standardized format facilitates searching of databases and callback functions implemented in applications.

This structure is defined as follows in TAPI.pas:

```
PLineCallInfo = ^TLineCallInfo;
linecallinfo_tag = packed record
  dwTotalSize,
  dwNeededSize,
  dwUsedSize: DWORD;
  hLine: HLINE;
  dwLineDeviceID,
  dwAddressID,
  dwBearerMode,
  dwRate,
  dwMediaMode,
```

```
dwAppSpecific,
dwCallID,
dwRelatedCallID,
dwCallParamFlags,
dwCallStates,
dwMonitorDigitModes,
dwMonitorMediaModes: DWORD;
DialParams: TLineDialParams;
dwOrigin,
dwReason,
dwCompletionID,
dwNumOwners,
dwNumMonitors,
dwCountryCode,
dwTrunk,
dwCallerIDFlags,
dwCallerIDSize,
dwCallerIDOffset,
dwCallerIDNameSize,
dwCallerIDNameOffset,
dwCalledIDFlags,
dwCalledIDSize,
dwCalledIDOffset,
dwCalledIDNameSize,
dwCalledIDNameOffset,
dwConnectedIDFlags,
dwConnectedIDSize,
dwConnectedIDOffset,
dwConnectedIDNameSize,
dwConnectedIDNameOffset,
dwRedirectionIDFlags,
dwRedirectionIDSize,
dwRedirectionIDOffset,
dwRedirectionIDNameSize,
dwRedirectionIDNameOffset,
dwRedirectingIDFlags,
dwRedirectingIDSize,
dwRedirectingIDOffset,
dwRedirectingIDNameSize,
dwRedirectingIDNameOffset,
dwAppNameSize,
dwAppNameOffset,
dwDisplayableAddressSize,
dwDisplayableAddressOffset,
dwCalledPartySize,
dwCalledPartyOffset,
dwCommentSize,
dwCommentOffset,
dwDisplaySize,
dwDisplayOffset,
dwUserUserInfoSize,
dwUserUserInfoOffset,
dwHighLevelCompSize,
dwHighLevelCompOffset,
dwLowLevelCompSize,
dwLowLevelCompOffset,
dwChargingInfoSize,
dwChargingInfoOffset,
dwTerminalModesSize,
dwTerminalModesOffset,
```

```
    dwDevSpecificSize,
    dwDevSpecificOffset: DWORD;
{$IFDEF TAPI20}
    dwCallTreatment,                              // TAPI v2.0
    dwCallDataSize,                               // TAPI v2.0
    dwCallDataOffset,                             // TAPI v2.0
    dwSendingFlowspecSize,                        // TAPI v2.0
    dwSendingFlowspecOffset,                      // TAPI v2.0
    dwReceivingFlowspecSize,                      // TAPI v2.0
    dwReceivingFlowspecOffset: DWORD;             // TAPI v2.0
{$ENDIF}
{$IFDEF TAPI30}
    dwAddressType: DWORD;                         // TAPI v3.0
{$ENDIF}
  end;
  TLineCallInfo = linecallinfo_tag;
  LINECALLINFO = linecallinfo_tag;
```

The fields of this structure are defined in Table 11-3.

Table 11-3: Fields of the LINECALLINFO structure

Field	Meaning
dwTotalSize	This field indicates the total size, in bytes, allocated to this data structure.
dwNeededSize	This field indicates the size, in bytes, for this data structure that is needed to hold all the returned information.
dwUsedSize	This field indicates the size, in bytes, of the portion of this data structure that contains useful information.
hLine	This field indicates the handle for the line device with which this call is associated.
dwLineDeviceID	This field indicates the device identifier of the line device with which this call is associated.
dwAddressID	This field indicates the address identifier permanently associated with the address where the call exists.
dwBearerMode	This field indicates the current bearer mode of the call. This field uses one of the LINEBEARERMODE_ constants.
dwRate	This field indicates the rate of the call's data stream in bps (bits per second).
dwMediaMode	This field indicates the media type of the information stream currently on the call. It is determined by the owner of the call. It uses the LINEMEDIAMODE_ constants.
dwAppSpecific	This field is not interpreted by the API implementation and service provider. It can be set by any owner application of this call with the lineSetAppSpecific() function.
dwCallID	This field indicates a unique identifier to each call assigned by the switch or service provider.
dwRelatedCallID	This field indicates the related call ID. Telephony environments that use the call ID may often find it necessary to relate one call to another.
dwCallParamFlags	This field indicates a collection of call-related parameters when the call is outgoing. These are the same call parameters specified in lineMakeCall(), using one or more of the LINECALLPARAMFLAGS_ constants.
dwCallStates	This field indicates the call states, one or more of the LINECALLSTATE_ constants (see Table 9-11), for which the application can be notified on this call. The dwCallStates member is constant in LINECALLINFO and does not change depending on the call state.

Field	Meaning
dwMonitorDigitModes	This field indicates the various digit modes, one or more of the LINEDIGITMODE_ constants, for which monitoring is currently enabled.
dwMonitorMediaModes	This field indicates the various media types for which monitoring is currently enabled; one or more of the LINEMEDIAMODE_ constants.
DialParams	This field indicates the dialing parameters currently in effect on the call of type LINEDIALPARAMS. Unless these parameters are set by either lineMakeCall() or lineSetCallParams(), their values are the same as the defaults used in the LINEDEVCAPS structure.
dwOrigin	This field indicates the identifiers where the call originated; one of the LINECALLORIGIN_ constants.
dwReason	This field indicates the reason why the call occurred; one of the LINECALLREASON_ constants.
dwCompletionID	This field indicates the completion identifier for the incoming call if it is the result of a completion request that terminates. This identifier is meaningful only if dwReason is LINECALLREASON_CALLCOMPLETION.
dwNumOwners	This field indicates the number of application modules with different call handles with owner privilege for the call.
dwNumMonitors	This field indicates the number of application modules with different call handles with monitor privilege for the call.
dwCountryCode	This field indicates the country code of the destination party. It is zero if unknown.
dwTrunk	This field indicates the number of the trunk over which the call is routed. This member is used for both incoming and outgoing calls. The dwTrunk member should be set to $FFFFFFFF if it is unknown.
dwCallerIDFlags	This field includes flags indicating the validity and content of the caller, or originator, party identifier information. It uses one of the LINECALLPARTYID_ constants.
dwCallerIDSize	This field indicates the size, in bytes, of the variably sized field containing the caller party ID number information of this data structure.
dwCallerIDOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the caller party ID number information.
dwCallerIDNameSize	This field indicates the size, in bytes, of the variably sized field containing the caller party ID name information.
dwCallerIDNameOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the caller party ID name information.
dwCalledIDFlags	This field includes flags indicating the validity and content of the called party ID information. The called party corresponds to the originally addressed party. This member uses one of the LINECALLPARTYID_ constants.
dwCalledIDSize	This field indicates the size, in bytes, of the variably sized field containing the called party ID number information.
dwCalledIDOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the called party ID number information.
dwCalledIDNameSize	This field indicates the size, in bytes, of the variably sized field containing the called party ID name information.
dwCalledIDNameOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the called party ID information.

Field	Meaning
dwConnectedIDFlags	This field includes flags indicating the validity and content of the connected party ID information. The connected party is the party that was actually connected to. This may be different from the called party ID if the call was diverted. This member uses one of the LINECALLPARTYID_ constants.
dwConnectedIDSize	This field indicates the size, in bytes, of the variably sized field containing the connected party identifier number information.
dwConnectedIDOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the connected party identifier number information.
dwConnectedIDNameSize	This field indicates the size, in bytes, of the variably sized field containing the connected party identifier name information.
dwConnectedIDNameOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the connected party identifier name information.
dwRedirectionIDFlags	This field includes flags indicating the validity and content of the redirection party identifier information. The redirection party identifies the address to which the session was redirected. This member uses one of the LINECALLPARTYID_ constants.
dwRedirectionIDSize	This field indicates the size, in bytes, of the variably sized field containing the redirection party identifier number information, and the offset, in bytes, from the beginning of this data structure.
dwRedirectionIDOffset	This field indicates the size, in bytes, of the variably sized field containing the redirection party identifier number information, and the offset, in bytes, from the beginning of this data structure.
dwRedirectionIDNameSize	This field indicates the size, in bytes, of the variably sized field containing the redirection party identifier name information.
dwRedirectionIDNameOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the redirection party identifier name information.
dwRedirectingIDFlags	This field includes flags indicating the validity and content of the redirecting party identifier information. The redirecting party identifies the address that redirects the session. This member uses one of the LINECALLPARTYID_ constants.
dwRedirectingIDSize	This field indicates the size, in bytes, of the variably sized field containing the redirecting party identifier number information.
dwRedirectingIDOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the redirecting party identifier number information.
dwRedirectingIDNameSize	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the redirecting party identifier number information.
dwRedirectingIDNameOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field containing the redirecting party identifier name information.
dwAppNameSize	This field indicates the size, in bytes, of the variably sized field holding the user-friendly application name of the application that first originated, accepted, or answered the call. This is the name that an application can specify in lineInitializeEx(). If the application specifies no such name, then the application's module filename is used instead.
dwAppNameOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field holding the user-friendly application name of the application that first originated, accepted, or answered the call. This is the name that an application can specify in lineInitializeEx(). If the application specifies no such name, then the application's module filename is used instead.

Field	Meaning
dwDisplayableAddressSize	This field indicates the size of the displayable string that is used for logging purposes. The information is obtained from LINECALLPARAMS for functions that initiate calls. The lineTranslateAddress() function returns appropriate information to be placed in this field in the dwDisplayableAddressSize field of the LINETRANSLATEOUTPUT structure.
dwDisplayableAddressOffset	This field indicates the offset of the displayable string that is used for logging purposes. The information is obtained from LINECALLPARAMS for functions that initiate calls. The lineTranslateAddress() function returns appropriate information to be placed in this field in the dwDisplayableAddressOffset field of the LINETRANSLATE-OUTPUT structure.
dwCalledPartySize	This field indicates the size, in bytes, of the variably sized field holding a user-friendly description of the called party. This information can be specified with lineMakeCall() and can be optionally specified in the lpCallParams parameter whenever a new call is established. It is useful for call logging purposes.
dwCalledPartyOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field holding a user-friendly description of the called party. This information can be specified with lineMakeCall() and can be optionally specified in the lpCallParams parameter whenever a new call is established. It is useful for call logging purposes.
dwCommentSize	This field indicates the size, in bytes, of the variably sized field holding a comment about the call provided by the application that originated the call using lineMakeCall(). This information can be optionally specified in the lpCallParams parameter whenever a new call is established.
dwCommentOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field holding a comment about the call provided by the application that originated the call using lineMakeCall(). This information can be optionally specified in the lpCallParams parameter whenever a new call is established.
dwDisplaySize	This field indicates the size, in bytes, of the variably sized field holding raw display information. Depending on the telephony environment, a service provider may extract functional information from this member pair for formatting and presentation most appropriate for this telephony configuration.
dwDisplayOffset	This field indicates the offset, in bytes, from the beginning of this data structure. Depending on the telephony environment, a service provider may extract functional information from this member pair for formatting and presentation most appropriate for this telephony configuration.
dwUserUserInfoSize	This field indicates the size, in bytes, of the variably sized field holding user-user information. The protocol discriminator field for the user-user information, if used, appears as the first byte of the data pointed to by dwUserUserInfoOffset and is accounted for in dwUserUserInfoSize.
dwUserUserInfoOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field holding user-user information. The protocol discriminator field for the user-user information, if used, appears as the first byte of the data pointed to by dwUserUserInfoOffset and is accounted for in dwUserUserInfoSize.
dwHighLevelCompSize	This field indicates the size, in bytes, of the variably sized field holding high-level compatibility information. The format of this information is specified by other standards (ISDN Q.931).
dwHighLevelCompOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field holding high-level compatibility information. The format of this information is specified by other standards (ISDN Q.931).

Field	Meaning
dwLowLevelCompSize	This field indicates the size, in bytes, of the variably sized field holding low-level compatibility information. The format of this information is specified by other standards (ISDN Q.931).
dwLowLevelCompOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field holding low-level compatibility information. The format of this information is specified by other standards (ISDN Q.931).
dwChargingInfoSize	This field indicates the size, in bytes, of the variably sized field holding charging information. The format of this information is specified by other standards (ISDN Q.931).
dwChargingInfoOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field holding charging information. The format of this information is specified by other standards (ISDN Q.931).
dwTerminalModesSize	This field indicates the size, in bytes, of the variably sized device field containing an array with DWORD-sized entries. Array entries are indexed by terminal identifiers in the range from zero to one less than dwNumTerminals. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the lineSetTerminal() function for this call's media stream, as specified by one of the LINETERMMODE_ constants described in Table 11-4.
dwTerminalModesOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized device field containing an array with DWORD-sized entries. Array entries are indexed by terminal identifiers in the range from zero to one less than dwNumTerminals. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the lineSetTerminal() function for this call's media stream, as specified by one of the LINETERMMODE_ constants described in Table 11-4.
dwDevSpecificSize	This field indicates the size, in bytes, of the variably sized field holding device-specific information.
dwDevSpecificOffset	This field indicates the offset, in bytes, from the beginning of this data structure of the variably sized field holding device-specific information.
dwCallTreatment	This field indicates the call treatment currently being applied on the call or that is applied when the call enters the next applicable state. It can be zero if call treatments are not supported.
dwCallDataSize	This field indicates the size, in bytes, of the call data that can be set by the application.
dwCallDataOffset	This field indicates the offset, in bytes, from the beginning of LINECALLINFO of the call data that can be set by the application.
dwSendingFlowspecSize	This field indicates the total size, in bytes, of a WinSock2 FLOWSPEC structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in SendingFlowspec.len in a WinSock2 QOS structure. It specifies the quality of service currently in effect in the sending direction on the call. The provider-specific portion following the FLOWSPEC structure must not contain pointers to other blocks of memory because TAPI does not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.

Field	Meaning
dwSendingFlowspecOffset	This field indicates the offset from the beginning of LINECALLINFO of a WinSock2 FLOWSPEC structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in SendingFlowspec.len in a WinSock2 QOS structure. It specifies the quality of service currently in effect in the sending direction on the call. The provider-specific portion following the FLOWSPEC structure must not contain pointers to other blocks of memory because TAPI does not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application
dwReceivingFlowspecSize	This field indicates the total size, in bytes, of a WinSock2 FLOWSPEC structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in ReceivingFlowspec.len in a WinSock2 QOS structure. It specifies the quality of service currently in effect in the receiving direction on the call. The provider-specific portion following the FLOWSPEC structure must not contain pointers to other blocks of memory because TAPI does not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.
dwReceivingFlowspecOffset	This field indicates the offset from the beginning of LINECALLINFO of a WinSock2 FLOWSPEC structure followed by WinSock2 provider-specific data, equivalent to what would have been stored in ReceivingFlowspec.len in a WinSock2 QOS structure. It specifies the quality of service currently in effect in the receiving direction on the call. The provider-specific portion following the FLOWSPEC structure must not contain pointers to other blocks of memory because TAPI does not know how to marshal the data pointed to by the private pointer(s) and convey it through interprocess communication to the application.
dwAddressType	The address type used for the call. This member of the structure is available only if the negotiated TAPI version is 3.0 or higher. Possible line address types include LINEADDRESSTYPE_PHONENUMBER, indicating a standard phone number, LINEADDRESSTYPE_SDP, indicating a Session Description Protocol (SDP) conference, LINEADDRESSTYPE_EMAILNAME, indicating a domain name, and LINEADDRESSTYPE_IPADDRESS, indicating an IP address.

Table 11-4: LINETERMMODE_ constants

Constant	Meaning
LINETERMMODE_LAMPS	Indicates that these are lamp events sent from the line to the terminal
LINETERMMODE_ BUTTONS	Indicates that these are button-press events sent from the terminal to the line
LINETERMMODE_ DISPLAY	Indicates that this is display information sent from the line to the terminal
LINETERMMODE_ RINGER	Indicates that this is ringer-control information sent from the switch to the terminal
LINETERMMODE_ HOOKSWITCH	Indicates that these are hookswitch events sent between the terminal and the line
LINETERMMODE_ MEDIATOLINE	Indicates that this is the unidirectional media stream from the terminal to the line associated with a call on the line (use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently)
LINETERMMODE_ MEDIAFROMLINE	Indicates that this is the unidirectional media stream from the line to the terminal associated with a call on the line (use this value when the routing of both unidirectional channels of a call's media stream can be controlled independently)

Constant	Meaning
LINETERMMODE_ MEDIABIDIRECT	Indicates that this is the bidirectional media stream associated with a call on the line and the terminal (use this value when the routing of both unidirectional channels of a call's media stream cannot be controlled independently)

function lineGetCallStatus TAPI.pas

Syntax

function lineGetCallStatus(hCall: HCALL; lpCallStatus: PLineCallStatus): Longint; stdcall;

Description

This function returns the current status of the specified call.

Parameters

hCall: A handle (HCALL) to the call to be queried. The call state of *hCall* can be any state.

lpCallStatus: A pointer (PLineCallStatus) to a variably sized data structure of type LINECALLSTATUS. If the request is successfully completed, this structure is filled with call status information. Before you call lineGetCall-Status(), you should set the *dwTotalSize* field of the LINECALLSTATUS structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVAL-CALLHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVAL-POINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, LINEERR_OPERATIONFAILED, and LINEERR_OPERATIONUNAVAIL.

See Also

LINE_CALLSTATE, LINECALLSTATUS, lineGetCallInfo

Example

Listing 11-4 shows how to get a call's status.

Listing 11-4: Getting a call's status

```
function TTapiInterface.GetCallStatus: boolean;
begin
  if fLineCallStatus=Nil then
    fLineCallStatus := AllocMem(SizeOf(TLineCallStatus)+1000);
  fLineCallStatus.dwTotalSize := SizeOf(TLineCallStatus)+1000;
  TapiResult := lineGetCallStatus(fCall, fLineCallStatus);
  result := TapiResult=0;
  if NOT result then ReportError(TapiResult);
end;
```

structure LINECALLSTATUS TAPI.pas

The LINECALLSTATUS structure describes the current status of a call. The information in this structure, as returned with lineGetCallStatus(), depends on the device capabilities of the address, the ownership of the call by the invoking application, and the current state of the call being queried. Device-specific extensions should use the DevSpecific (*dwDevSpecificSize* and *dwDevSpecificOffset*) variably sized area of this data structure. The application is sent a LINE_CALLSTATE message whenever the call state of a call changes. This message only provides the new call state of the call. Additional status about a call is available with lineGetCallStatus(). The members *dwCallFeatures2* and *tStateEntryTime* are available only to applications that open the line device with a TAPI version of 2.0 or greater. This structure is defined as follows in TAPI.pas:

```
  PLineCallStatus = ^TLineCallStatus;
  linecallstatus_tag = packed record
    dwTotalSize,
    dwNeededSize,
    dwUsedSize,
    dwCallState,
    dwCallStateMode,
    dwCallPrivilege,
    dwCallFeatures,
    dwDevSpecificSize,
    dwDevSpecificOffset: DWORD;
{$IFDEF TAPI20}
    dwCallFeatures2: DWORD;                          // TAPI v2.0
    {$IFDEF WIN32}
    tStateEntryTime: TSystemTime;                    // TAPI v2.0
    {$ELSE}
    tStateEntryTime: array[0..7] of WORD;            // TAPI v2.0
    {$ENDIF}
{$ENDIF}
  end;
  TLineCallStatus = linecallstatus_tag;
  LINECALLSTATUS = linecallstatus_tag;
```

The various fields of this structure are described in Table 11-5.

Table 11-5: Fields of the LINECALLSTATUS structure

Field	Meaning
dwTotalSize	The total size in bytes allocated to this data structure
dwNeededSize	The size in bytes for this data structure that is needed to hold all the returned information
dwUsedSize	The size in bytes of the portion of this data structure that contains useful information
dwCallState	Specifies the current call state of the call. This field uses the LINECALLSTATE_ constants described in Table 8-5.
dwCallStateMode	The interpretation of the dwCallStateMode field is call-state-dependent. It specifies the current mode of the call while in its current state (if that state defines a mode). This field uses the LINECALLSTATE_ constants described in Table 9-11.

Field	Meaning
dwCallPrivilege	The application's privilege for this call. This field uses the following LINECALLPRIVILEGE_ constants: LINECALLPRIVILEGE_MONITOR indicates that the application has monitor privilege. LINECALLPRIVILEGE_OWNER indicates that the application has owner privilege.
dwCallFeatures	These flags indicate which Telephony API functions can be invoked on the call, given the availability of the feature in the device capabilities, the current call state, and call ownership of the invoking application. A zero indicates the corresponding feature cannot be invoked by the application on the call in its current state; a one indicates the feature can be invoked. This field uses the LINECALLFEATURE_ constants described in Table 8-7.
dwDevSpecificSize	The size in bytes of the variably sized device-specific field
dwDevSpecificOffset	The offset in bytes from the beginning of this data structure
dwCallFeatures2	Indicates additional functions can be invoked on the call, given the availability of the feature in the device capabilities, the current call state, and call ownership of the invoking application. It has an extension of the dwCallFeatures field. This field uses LINECALLFEATURE2_ constants.
tStateEntryTime	The Coordinated Universal Time at which the current call state was entered

See Also

LINE_CALLSTATE, LINEDIALPARAMS, lineGetCallStatus

function lineGetConfRelatedCalls   TAPI.pas

Syntax

function lineGetConfRelatedCalls(hCall: HCALL; lpCallList: PLineCallList): Longint; stdcall;

Description

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

Parameters

hCall: A handle (HCALL) to a call. This is either a conference call or a participant call in a conference call. For a conference parent call, the call state of *hCall* can be any state. For a conference participant call, it must be in the conferenced state.

lpCallList: A pointer (PLineCallList) to a variably sized data structure of type LINECALLLIST. If the request is succesfully completed, call handles to all calls in the conference call are returned in this structure. The first call in the list is the conference call, and the other calls are the participant calls. The application is granted monitor privilege to those calls for which it does not already have handles; the privileges to calls in the list for which the application already has handles is unchanged. Before you call lineGetConfRelatedCalls(), you should set the *dwTotalSize* field of the LINECALLLIST structure to indicate the amount of memory available to TAPI for returning information.

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_NOCONFERENCE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_STRUCTURETOOSMALL, LINEERR_NOMEM, and LINEERR_UNINITIALIZED.

See Also

LINE_CALLSTATE, lineCompleteTransfer, lineGetCallInfo, lineGetCallStatus, lineSetCallPrivilege

Example

Listing 11-5 shows how to get the list of conference-related calls.

Listing 11-5: Getting the list of conference-related calls

```
function TTapiInterface.GetConfRelatedCalls: boolean;
begin
  if fCallList=Nil then
    fCallList := AllocMem(SizeOf(fCallList)+1000);
  fCallList.dwTotalSize := SizeOf(fCallList)+1000;
  TapiResult := lineGetConfRelatedCalls(fCall, fCallList);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

function lineGetNewCalls TAPI.pas

Syntax

function lineGetNewCalls(hLine: HLINE; dwAddressID, dwSelect: DWORD; lpCallList: PLineCallList): Longint; stdcall;

Description

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

Parameters

hLine: A handle (HLINE) to an open line device

dwAddressID: A DWORD holding an address on the given open line device

dwSelect: A DWORD holding the selection of requested calls. This *dwSelect* parameter can only have one bit set. This parameter should be selected from one of the following LINECALLSELECT_ constants:

LINECALLSELECT_LINE, which selects calls on the specified line device. The *hLine* parameter must be a valid line handle; *dwAddressID* is ignored;

LINECALLSELECT_ADDRESS, which selects calls on the specified address on the specified line device. Both *hLine* and *dwAddressID* must be valid.

lpCallList: A pointer(PLineCallList) to a variably sized data structure of type LINECALLLIST. If the request is successfully completed, call handles to all selected calls are returned in this structure. Before you call lineGetNewCalls(), you should set the *dwTotalSize* field of the LINECALLLIST structure to indicate the amount of memory available to TAPI for returning information.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALADDRESSID, LINEERR_OPERATIONFAILED, LINEERR_INVALCALLSELECT, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALLINEHANDLE, LINEERR_STRUCTURETOOSMALL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, and LINEERR_NOMEM.

See Also

LINECALLLIST, lineGetCallInfo, lineGetCallStatus, lineSetCallPrivilege

Example

Listing 11-6 shows how to get the handles to monitor calls not presently owned.

Listing 11-6: Getting the handles to monitor calls not presently owned

```
function TTapiInterface.GetNewCalls: boolean;
begin
  if fCallList=Nil then
    fCallList := AllocMem(SizeOf(fCallList)+1000);
  fCallList.dwTotalSize := SizeOf(fCallList)+1000;
  TapiResult := lineGetNewCalls(fLine, 0,
    LINECALLSELECT_LINE, fCallList);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

structure LINECALLLIST TAPI.pas

The LINECALLLIST structure describes a list of call handles. A structure of this type is returned by the functions lineGetNewCalls() and lineGetConf-RelatedCalls(). No extensions are used with this structure. This structure is defined as follows in TAPI.pas:

```
PLineCallList = ^TLineCallList;
linecalllist_tag = packed record
  dwTotalSize,
  dwNeededSize,
  dwUsedSize,
  dwCallsNumEntries,
  dwCallsSize,
  dwCallsOffset: DWORD;
end;
TLineCallList = linecalllist_tag;
LINECALLLIST = linecalllist_tag;
```

The fields of this structure are defined in Table 11-6.

Table 11-6: Fields of the LINECALLLIST structure

Fields	Meaning
dwTotalSize	The total size in bytes allocated to this data structure
dwNeededSize	The size in bytes for this data structure that is needed to hold all the returned information
dwUsedSize	The size in bytes of the portion of this data structure that contains useful information
dwCallsNumEntries	The number of handles in the hCalls array
dwCallsSize	The size in bytes of the variably sized field (which is an array of HCALL-sized handles)
dwCallsOffset	The offset in bytes from the beginning of this data structure of the variably sized field (which is an array of HCALL-sized handles)

See Also

lineGetConfRelatedCalls, lineGetNewCalls

function lineGetNumRings TAPI.pas

Syntax

function lineGetNumRings(hLine: HLINE; dwAddressID: DWORD; var dwNumRings: DWORD): Longint; stdcall;

Description

This function determines the number of times an inbound call on the given address should ring prior to answering the call.

Parameters

hLine: A handle (HLINE) to the open line device

dwAddressID: A DWORD indicating the number of rings that is the minimum of all current lineSetNumRings() requests

var dwNumRings: A DWORD holding an address on the line device

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALADDRESSID, LINEERR_OPERATIONFAILED, LINEERR_INVALLINEHANDLE, LINEERR_RESOURCEUNAVAIL, LINEERR_INVALPOINTER, LINEERR_UNINITIALIZED, and LINEERR_NOMEM.

See Also

LINE_LINEDEVSTATE, lineSetNumRings

Example

Listing 11-7 shows how to determine the number of rings an inbound call will be given before it is answered.

Listing 11-7: Determining the number of rings an inbound call will be given before it is answered

```
function TTapiInterface.GetNumRings(var NumRings : DWord): boolean;
begin
  TapiResult := lineGetNumRings(fLine, fAddressID,
    NumRings);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

function lineGetRequest TAPI.pas

Syntax

function lineGetRequest(hLineApp: HLINEAPP; dwRequestMode: DWORD; lpRequestBuffer: Pointer): Longint; stdcall;

Description

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

Parameters

hLineApp: The application's usage handle (HLINEAPP) for the line portion of TAPI

dwRequestMode: A DWORD indicating the type of request that is to be obtained. Note that *dwRequestMode* can only have one bit set. This parameter uses the LINEREQUESTMODE_ constant LINEREQUESTMODE_MAKECALL.

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

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALAPPHANDLE, LINEERR_NOTREGISTERED, LINEERR_INVALPOINTER, LINEERR_OPERATIONFAILED, LINEERR_INVALREQUESTMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, LINEERR_UNINITIALIZED, and LINEERR_NOREQUEST.

See Also

LINE_REQUEST, LINEREQMAKECALL, tapiRequestMakeCall

Example

Listing 11-8 shows how to retrieve the next by-proxy request.

Listing 11-8: Retrieving the next by-proxy request

```
function TTapiInterface.GetLineRequest: boolean;
begin
  if fLineReqMakeCallRec=Nil then
    fLineReqMakeCallRec := AllocMem(SizeOf(TLineReqMakeCall));
  TapiResult := lineGetRequest(fLine,
    LINEREQUESTMODE_MAKECALL, fLineReqMakeCallRec);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```


structure LINEREQMAKECALL TAPI.pas

The LINEREQMAKECALL structure describes a tapiRequestMakeCall() request.

The *szDestAddress* field contains the address of the remote party; the other fields are useful for logging purposes. An application must use this structure to interpret the request buffer it received from lineGetRequest() with the LINEREQUESTMODE_MAKECALL request mode. It is defined as follows in TAPI.pas:

```
  PLineReqMakeCall = ^TLineReqMakeCall;
  linereqmakecall_tag = packed record
    szDestAddress: array[0..TAPIMAXDESTADDRESSSIZE - 1] of Char;
    szAppName: array[0..TAPIMAXAPPNAMESIZE - 1] of Char;
```

```
    szCalledParty: array[0..TAPIMAXCALLEDPARTYSIZE - 1] of Char;
    szComment: array[0..TAPIMAXCOMMENTSIZE - 1] of Char;
  end;
  TLineReqMakeCall = linereqmakecall_tag;
  LINEREQMAKECALL = linereqmakecall_tag;
```

The fields of the LINEREQMAKECALL structure are described in Table 11-7.

Table 11-7: Fields of the LINEREQMAKECALL structure

Field	Member
szDestAddress	The NULL-terminated destination address [size TAPIMAXADDRESSSIZE] of the make-call request. The address can use the canonical address format or the dialable address format. The maximum length of the address is TAPIMAXDESTADDRESSSIZE characters, which includes the NULL terminator. Longer strings are truncated.
szAppName	The ASCII NULL-terminated string [size TAPIMAXAPPNAMESIZE] holding the user-friendly application name or filename of the application that originated the request. The maximum length of the address is TAPIMAXAPPNAMESIZE characters, which includes the NULL terminator.
szCalledParty	The ASCII NULL-terminated string [size TAPIMAXCALLEDPARTYSIZE] holding the user-friendly called-party name. The maximum length of the called-party information is TAPIMAXCALLEDPARTYSIZE characters, which includes the NULL terminator.
szComment	The ASCII NULL-terminated string [size TAPIMAXCOMMENTSIZE] comment about the call request. The maximum length of the comment string is TAPIMAXCOMMENTSIZE characters, which includes the NULL terminator.

structure LINEREQMEDIACALL TAPI.pas

The LINEREQMEDIACALL structure describes a request initiated by a call to the lineGetRequest() function. It is defined as follows in TAPI.pas:

```
  PLineReqMediaCall = ^TLineReqMediaCall;
  linereqmediacall_tag = packed record
    hWnd: HWND;
    wRequestID: WPARAM;
    szDeviceClass: array[0..TAPIMAXDEVICECLASSSIZE - 1] of Char;
    ucDeviceID: array[0..TAPIMAXDEVICEIDSIZE - 1] of Byte;
    dwSize,
    dwSecure: DWORD;
    szDestAddress: array[0..TAPIMAXDESTADDRESSSIZE - 1] of Char;
    szAppName: array[0..TAPIMAXAPPNAMESIZE - 1] of Char;
    szCalledParty: array[0..TAPIMAXCALLEDPARTYSIZE - 1] of Char;
    szComment: array[0..TAPIMAXCOMMENTSIZE - 1] of Char;
  end;
  TLineReqMediaCall = linereqmediacall_tag;
  LINEREQMEDIACALL = linereqmediacall_tag;

 {$IFDEF TAPI20}
  PLineReqMediaCallW = ^TLineReqMediaCallW;
  linereqmediacallW_tag = packed record
    hWnd: HWND;
    wRequestID: WPARAM;
    szDeviceClass: array[0..TAPIMAXDEVICECLASSSIZE - 1] of WideChar;
    ucDeviceID: array[0..TAPIMAXDEVICEIDSIZE - 1] of Byte;
    dwSize,
```

```
    dwSecure: DWORD;
    szDestAddress: array[0..TAPIMAXDESTADDRESSSIZE - 1] of WideChar;
    szAppName: array[0..TAPIMAXAPPNAMESIZE - 1] of WideChar;
    szCalledParty: array[0..TAPIMAXCALLEDPARTYSIZE - 1] of WideChar;
    szComment: array[0..TAPIMAXCOMMENTSIZE - 1] of WideChar;
  end;
  TLineReqMediaCallW = linereqmediacallW_tag;
  LINEREQMEDIACALLW = linereqmediacallW_tag;
```

The fields of the LINEREQMEDIACALL structure are described in Table 11-8.

Table 11-8: Fields of the LINEREQMEDIACALL structure

Field	Member
hWnd	Handle to the window of the application which made the request
wRequestID	Identifier of the request used to match an asynchronous response
szDeviceClass	The device class [size TAPIMAXDEVICECLASSSIZE] required to fill the request
ucDeviceID	The device identifier of size TAPIMAXDEVICEIDSIZE
dwSize	Size in bytes of this structure
dwSecure	Undocumented DWORD field; may be for future use
szDestAddress	Destination address of size TAPIMAXDESTADDRESSSIZE
szAppName	Name of application which made the request of size TAPIMAXAPPNAMESIZE
szCalledParty	Called party name of size TAPIMAXCALLEDPARTYSIZE
szComment	Comment buffer of size TAPIMAXCOMMENTSIZE

See Also

lineGetRequest

function lineHandoff TAPI.pas

Syntax

function lineHandoff(hCall: HCALL; lpszFileName: LPCSTR; dwMediaMode: DWORD): Longint; stdcall;

Description

This function gives ownership of the specified call to another application. The application can be either specified directly by its filename or indirectly as the highest priority application that handles calls of the specified media mode.

Parameters

hCall: A handle (HCALL) to the call to be handed off. The application must be an owner of the call. The call state of *hCall* can be any state.

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

dwMediaMode: A DWORD indicating the media mode used to identify the target for the indirect handoff. The *dwMediaMode* parameter indirectly identifies the target application that is to receive ownership of the call. This parameter is ignored if *lpszFileName* is not NULL. Only a single flag may be set in the *dwMediaMode* parameter at any one time. This parameter uses the LINEMEDIAMODE_ constants shown in Table 11-9.

Return Value

Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALCALLHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALMEDIAMODE, LINEERR_TARGETNOTFOUND, LINEERR_INVALPOINTER, LINEERR_TARGETSELF, LINEERR_NOMEM, LINEERR_UNINITIALIZED, and LINEERR_NOTOWNER.

See Also

LINECALLINFO, lineGetCallStatus, lineOpen, lineSetCallPrivilege, lineSetMediaMode

Example

Listing 11-9 shows how to hand off a call to another application.

Listing 11-9: Handing off a call to another application

```
function TTapiInterface.HandoffLine(ACall : HCall; TargetApp : string;
   ModeDesired : DWord): boolean;
begin
  TapiResult := lineHandoff(ACall, PChar(TargetApp), ModeDesired);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

Table 11-9: LINEMEDIAMODE_ constants used in the lineHandoff() function's dwMediaMode parameter

Constant	Meaning
LINEMEDIAMODE_UNKNOWN	The target application is the one that handles calls of unknown media mode (unclassified calls).
LINEMEDIAMODE_INTERACTIVEVOICE	The target application is the one that handles calls with the interactive voice media mode (live conversations).
LINEMEDIAMODE_AUTOMATEDVOICE	Voice energy is present on the call and the voice is locally handled by an automated application.
LINEMEDIAMODE_DATAMODEM	The target application is the one that handles calls with the data modem media mode.
LINEMEDIAMODE_G3FAX	The target application is the one that handles calls with the group 3 fax media mode.

Constant	Meaning
LINEMEDIAMODE_TDD	The target application is the one that handles calls with the TDD (Telephony Devices for the Deaf) media mode.
LINEMEDIAMODE_G4FAX	The target application is the one that handles calls with the group 4 fax media mode.
LINEMEDIAMODE_DIGITALDATA	The target application is the one that handles calls that are digital data calls.
LINEMEDIAMODE_TELETEX	The target application is the one that handles calls with the teletex media mode.
LINEMEDIAMODE_VIDEOTEX	The target application is the one that handles calls with the video-tex media mode.
LINEMEDIAMODE_TELEX	The target application is the one that handles calls with the telex media mode.
LINEMEDIAMODE_MIXED	The target application is the one that handles calls with the ISDN mixed media mode.
LINEMEDIAMODE_ADSI	The target application is the one that handles calls with the ADSI (Analog Display Services Interface) media mode.
LINEMEDIAMODE_VOICEVIEW	The media mode of the call is VoiceView.

function lineRegisterRequestRecipient TAPI.pas

Syntax

function lineRegisterRequestRecipient(hLineApp: HLINEAPP; dwRegistrationInstance, dwRequestMode, bEnable: DWORD): Longint; stdcall;

Description

This function registers the invoking application as a recipient of requests for the specified request mode.

Parameters

hLineApp: The application's usage handle (HLINEAPP) for the line portion of TAPI

dwRegistrationInstance: An application-specific DWORD value 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: A DWORD indicating the type or types of request for which the application registers. One or both bits may be set. This parameter uses the following LINEREQUESTMODE_ constant:
LINEREQUESTMODE_ MAKECALL, which indicates a tapiRequestMakeCall() request.

bEnable: A DWORD that can be set to TRUE or FALSE. If TRUE, the application registers; if FALSE, the application deregisters for the specified request modes.

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALAPPHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALREQUESTMODE, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, and LINEERR_UNINITIALIZED.

See Also

LINE_REQUEST, lineGetRequest, lineShutdown, tapiRequestMakeCall

Example

Listing 11-10 shows how to call the lineRegisterRequestRecipient() function.

Listing 11-10: Calling the lineRegisterRequestRecipient() function

```
function TTapiInterface.RegisterRequestRecipient: boolean;
begin
  TapiResult := lineRegisterRequestRecipient(fLineApp,
    fRegistrationInstance, LINEREQUESTMODE_MAKECALL, 1);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

LINEREQUESTMODE_ Constants

The LINEREQUESTMODE_ bit-flag constants describe different types of telephony requests that can be made from one application to another. They are defined in Table 11-10.

Table 11-10: LINEREQUESTMODE_ constants

Constant	Meaning
LINEREQUESTMODE_DROP	A tapiRequestDrop request
LINEREQUESTMODE_MAKECALL	A tapiRequestMakeCall request
LINEREQUESTMODE_MEDIACALL	A tapiRequestMediaCall request

function lineSetNumRings TAPI.pas

Syntax

function lineSetNumRings(hLine: HLINE; dwAddressID, dwNumRings: DWORD): Longint; stdcall;

Description

This function sets the number of rings that must occur before an incoming call is answered. This function can be used to implement a toll-saver-style function. It allows multiple independent applications to each register the number of rings.

The function lineGetNumRings() returns the minimum number of all the 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.

Parameters

hLine: A handle (HLINE) to the open line device

dwAddressID: A DWORD holding an address on the line device

dwNumRings: A DWORD indicating the number of rings before a call should be answered in order to honor the toll-saver requests from all applications

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_INVALLINEHANDLE, LINEERR_OPERATIONFAILED, LINEERR_INVALADDRESSID, LINEERR_RESOURCEUNAVAIL, LINEERR_NOMEM, and LINEERR_UNINITIALIZED.

See Also

LINE_CALLSTATE, LINE_LINEDEVSTATE, lineGetNumRings

Example

Listing 11-11 shows how to call the lineSetNumRings() function.

Listing 11-11: Calling the lineSetNumRings() function

```
function TTapiInterface.SetNumRings(RequestedRings : Cardinal): boolean;
begin
  TapiResult := lineSetNumRings(fLine, fAddressID, RequestedRings);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

function lineSetTollList TAPI.pas

Syntax

function lineSetTollList(hLineApp: HLINEAPP; dwDeviceID: DWORD; lpszAddressIn: LPCSTR; dwTollListOption: DWORD): Longint; stdcall;

Description

This function manipulates the toll list.

Parameters

hLineApp: The application handle (HLINEAPP) returned by lineInitializeEx(). If an application has not yet called the lineInitializeEx() function, it can set the *hLineApp* parameter to NULL.

dwDeviceID: A DWORD holding the device ID for the line device upon which the call is intended to be dialed, so variations in dialing procedures on different lines can be applied to the translation process.

lpszAddressIn: A pointer (LPCSTR) 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: A DWORD indicating the toll list operation to be performed. Only a single flag can be set. This parameter uses the following LINETOLLLISTOPTION_ constants:
LINETOLLLISTOPTION_ADD causes the prefix contained within the string pointed to by *lpszAddressIn* to be added to the toll list for the current location.
LINETOLLLISTOPTION_REMOVE causes the prefix to be removed from the toll list of the current location (if toll lists are not used or are not relevant to the country indicated in the current location, the operation has no effect).

Return Value

This function returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are LINEERR_BADDEVICEID, LINEERR_NODRIVER, LINEERR_INVALAPPHANDLE, LINEERR_NOMEM, LINEERR_INVALADDRESS, LINEERR_OPERATIONFAILED, LINEERR_INVALPARAM, LINEERR_RESOURCEUNAVAIL, LINEERR_INIFILECORRUPT, LINEERR_UNINITIALIZED, and LINEERR_INVALLOCATION.

See Also

lineInitializeEx

Example

Listing 11-12 shows how to call the lineSetTollList() function. The *CallAddress* parameter in the SetTollList() method must be in the canonical format we discussed in Chapter 10. The sample application calls this function in this manner (note the canonical number in the string):

```
if NOT TapiInterface.SetTollList('+1 (502) 597-6000', LINETOLLLISTOPTION_ADD)
                                  then ShowMessage('Could not set toll list');.
```

Listing 11-12: Calling the lineSetTollList() function

```
function TTapiInterface.SetTollList(CallAddress : string;
           TollListAction : DWord): boolean;
begin
  TapiResult := lineSetTollList(fLineApp, 0{fLine}, PChar(CallAddress), TollListAction);
  result := TapiResult=0;
  if NOT result then ReportError(TAPIResult);
end;
```

In this, our final chapter, we have examined the issues, functions, and structures used in accepting incoming calls. In this book, we have examined all of the basic TAPI line functions. The remaining TAPI functions, including those supporting phone devices, are topics for another book.

Appendix A

Glossary of Important Communications Programming Terms

address	With TAPI, the actual telephone number, generally including the national or international code. Under Winsock, an integer value used to identify a particular computer that must appear in each packet sent to the computer.
address binding	The translation of a higher-layer address into an equivalent lower-layer address. For example, translation of a computer's IP address to the computer's Ethernet addresses.
address mask	A synonym for subnet mask.
address resolution	Conversion of a protocol address into a corresponding physical address (IP address => Ethernet address). Depending on the underlying network, resolution may require broadcasting on a local network. See ARP.
API	Application Program Interface. The specifications of the operations an application program must invoke to communicate over a network. The Windows Sockets API is the most popular for Internet communication on Windows platforms. The Telephony API (TAPI) is the basic interface for telephony programming.
ARP	The Address Resolution Protocol maps an IP address to the equivalent hardware address.
ARPANET	A pioneering network that developed into the Internet.
Assisted Telephony	The high-level telephony functions that provide easy access to call-placing functionality in non-telephony applications.
ATM	Asynchronous Mode Transfer, which is a connection-oriented network technology that uses small, fixed-sized cells at the lowest layer. ATM has the potential advantage of being able to support voice, video, and data with a single underlying technology.

Basic Telephony	Telephony line device services that are available under any TAPI implementation regardless of the service provider. An application can always assume these services will be available.
big endian	A format for storage or transmission of binary data in which the most significant byte (bit) comes first. The TCP/IP standard network byte order is big endian. See little endian.
BSD UNIX	Berkeley Software Distribution UNIX. The version of UNIX released by University of California, Berkeley or one of the commercial systems derived from it. BSD UNIX was the first to include TCP/IP protocols.
canonical phone number	An ASCII string intended to function as a universally constant phone number, consisting of country code, area code, phone number, possibly other data, and always beginning with a plus (+) character.
CENTREX	CENTRal EXchange. Provides centralized network services (such as conferencing) without the need to install special in-house equipment (as one might find with a PBX). With CENTREX, the user must pay for the use of those services.
checksum	This is a computation that uses one's complement of 16-bit words to provide a basic check for integrity of the data.
client-server	A system in which a client requests and receives data from a server.
Completion Port mechanism	During TAPI's initialization, this method sets up a mechanism for handling TAPI messages that uses a completion port, one that your application sets up and specifies in the *hCompletionPort* field in LINEINITIALIZEEXPARAMS. Whenever a telephony event needs to be sent to an application, TAPI will send it by calling the PostQueuedCompletionStatus() function.
CR-LF	Carriage return and line feed pair (#13#10) that is required to mark the end of the data stream in high-level protocols, such as FTP, HTTP, and many others.
DARPA	Defense Advanced Research Projects Agency; sponsor of ARPANET.
dialable phone number	An address or phone number that can be dialed on the particular line. A dialable address contains part addressing information and is part navigational in nature. A phone number string that does not begin with a "+" character is assumed to be dialable.
DNS	The Domain Name System is an online distributed database system for mapping human-readable machine names into IP addresses. DNS servers throughout the connected Internet implement a hierarchical name space that allows sites freedom in assigning machine names and addresses. DNS also support separate mappings between mail destinations and IP addresses.

dotted decimal notation	This is a notation to represent a 32-bit binary integer that consists of four 8-bit numbers written in base 10 with periods separating them. Many TCP/IP applications accept dotted decimal notation in place of destination machine names.
event handle mechanism	During its initialization using this mechanism, TAPI creates an event object for the application, returning a handle to the object in the *hEvent* field in LINEINITIALIZEEXPARAMS.
Extended Telephony	Telephony services that vary from one service provider to another. An application that provides any of these services must check for their availability before exposing them for the user.
firewall	A configuration of routers and networks placed between an organization's internal network and a connection to the Internet to provide security.
Hidden Window mechanism	During TAPI's initialization, this method creates a hidden window to which all messages will be sent; it is the only one available to TAPI 1.x applications.
ICMP	Internet Control Message Protocol is an integral part of IP that handles errors and control messages. Routers and hosts use ICMP to send reports of problems about datagrams back to the original source that sent the datagram. ICMP also includes an echo request/reply used to test whether a destination is reachable and responding.
ICMPv6	Version 6 of ICMP.
internet	A collection of networks interconnected by routers to function logically as a large virtual network.
Internet	The collection of networks and routers that spans the world. It uses TCP/IP protocols to form a large and virtual network.
IP	The Internet Protocol is the base protocol for all protocols. It handles hardware-independent addressing, routing, fragmentation, and reassembly of packets.
IP address	In IPv4, an IP address is a 32-bit number.
IP Multicast	This is a technology that permits replication of data from a single sender to many receivers. The technology relies on a special class of addresses, Class D.
IP router	An intelligent device that routes incoming IP datagrams to other routers or hosts according to the IP address in the destination part of the IP header. See router.
IPng	A synonym for IPv6, Internet Protocol's next generation. It is also known as IPv6.7

IPv4	Internet Protocol version 4 is the official name of the current version of IP.
ISDN	Integrated Services Digital Network is a technology that provides a minimum of three channels (two for voice or data and one strictly for data or signaling information) and as many as 32 channels for simultaneous, independently operated transmission of voice and data.
ISP	An Internet Service Provider provides access to the Internet for dial-up and connected users.
LAN	A Local Area Network is a physical network that spans short distances (up to a few thousand meters). See MAN and WAN.
line device	One of the two principle TAPI abstractions (the other being a phone device) representing an actual physical line, such as a modem, fax device, or ISDN card, that is connected to an actual telephone line.
little endian	A format for storage or transmission of binary data in which the least-significant byte (bit) comes first. See big endian.
MAC	This is an acronym for Media Access Control. Each network card has a MAC address to identify itself to the network.
MAN	Metropolitan Area Network is a physical network that operates at high speed over distances sufficient for a city. See LAN and WAN.
mask	See subnet mask.
MBone	A cooperative agreement among sites to forward multicast datagrams across the Internet by use of IP tunneling.
media modes	TAPI uses media modes to keep track of the media being transferred over a line. Media are the forms in which data can be transmitted over a line, the four main types being voice, speech, fax, and data. Specific media could include normal interactive voice, automated voice, a specific fax format, and quite a few others.
MTU	Maximum Transmission Unit defines the largest amount of data that can be transmitted in one segment. The MTU is determined by the network hardware. It is typically 1500.
multicast	A technique that allows copies of a single packet to be passed to a selected subset of all possible destinations. Some hardware (e.g., Ethernet) supports multicast by allowing a network interface to belong to one or more multicast groups. See IP Multicast.
multihomed	A computer is said to be multihomed if it has multiple network interfaces. These interfaces can be separate network interface cards (NICs) or multiple IP addresses on one NIC.

network byte order	The TCP/IP standard for transmission of integers that specifies the most significant byte appears first (big endian).
NIC	Network Interface Card is a device that plugs into the bus on a computer and connects the computer to a network.
OSI	Open Systems Interconnection is a collection of protocols developed by the International Organization for Standardization (ISO) as a competitor to TCP/IP.
PBX	Private Branch Exchange. An organization's internal telephone system, one that might include functionality exceeding that of the local telephone company itself.
phone device	An abstraction of a physical phone with some of the features of such a device including newer ones, such as buttons, data storage, and display.
PING	Packet InterNet Groper is the name of a program to test reachability of destinations by sending an ICMP echo request and waiting for a reply.
POTS	Plain Old Telephone Service.
PPP	The Point to Point Protocol is a protocol for framing IP when sending across a serial line.
promiscuous mode	In this mode, a network interface hardware allows the host computer to receive all packets on the network.
QOS	Quality of Service sets limits on the loss, delay, jitter, and minimum throughput that a network guarantees to deliver.
RFC	Request For Comments is a document that is either a series of notes that contain surveys, measurements, ideas, techniques, and observations, or proposed and accepted TCP/IP protocol standards.
router	A special-purpose, dedicated computer that attaches to two or more networks and forwards packets from one to the other. In particular, an IP router forwards IP datagrams among the networks to which it connects. A router uses the destination address on a datagram to choose the next router to which it forwards the datagram.
socket	A "plug" or an endpoint of the communication link.
subnet mask	A bit mask used to select the bits from an IP address that correspond to the subnet. Each mask is 32 bits long. Bits set to one identify a network and bits set to zero identify a host.
TAPI	Telephony Application Programming Interface. A series of functions, structures, and constants that provide support for the full range of telephony functionality in Windows.

TCP	Transmission Control Protocol is the protocol that defines a virtual circuit between two computers, thus enabling them to exchange data in byte streams.
TSPI	The Telephony Service Provider Interface is the programming means through which a service provider delivers different levels of the telephony support—basic, supplementary, or extended. TAPI must call TAPISRV.EXE to implement and manage its functions; TAPISRV.EXE then communicates with one or more telephony service providers (drivers) to fulfill the telephony request.
TTL	Time To Live is a technique used in best-effort delivery systems to avoid endlessly looping packets. For example, each IP datagram is assigned an integer time to live when it is created. Each router decrements the time to live field when the datagram arrives, and a router discards any datagrams if the time to live counter reaches zero.
UDP	User Datagram Protocol is the protocol that allows an application on one machine to send a datagram to an application on another. UDP uses IP to deliver datagrams. The difference between UDP and IP is that UDP uses a port number, allowing the sender to distinguish among multiple applications on a remote machine.
WAN	Wide Area Network is a physical network that spans large geographic distances, such as continents. See LAN and MAN.
Winsock	This is an abbreviation for Windows Sockets.
WOSA	Microsoft's Windows Open Systems Architecture, used with TAPI and the Winsock API, provides transparent support for communications hardware and just about every other type of hardware through a device-independent interface.

Appendix B

Error Codes, Their Descriptions, and Their Handling

In this appendix we will provide the names, descriptions, and numerical values of Winsock and TAPI line error codes. Additional information on the TAPI errors can be found in the TAPI Help file. Please be aware, however, that not every error is listed (we have indicated the ones omitted). As a bonus, we will show the code we use for handling TAPI errors in a single, centralized routine.

Winsock Errors

Name	Description	Code Number
WSAEINTR	Interrupted system call	10004
WSAEBADF	Bad file number	10009
WSAEACCES	Permission denied	10013
WSAEFAULT	Bad address	10014
WSAEINVAL	Invalid argument	10022
WSAEMFILE	Too many open files	10024
WSAEWOULDBLOCK	Operation would block	10035
WSAEINPROGRESS	Operation now in progress	10036
WSAEALREADY	Operation already in progress	10037
WSAENOTSOCK	Socket operation on nonsocket	10038
WSAEDESTADDRREQ	Destination address required	10039
WSAEMSGSIZE	Message too long	10040
WSAEPROTOTYPE	Protocol wrong type for socket	10041
WSAENOPROTOOPT	Protocol not available	10042
WSAEPROTONOSUPPORT	Protocol not supported	10043
WSAESOCKTNOSUPPORT	Socket not supported	10044
WSAEOPNOTSUPP	Operation not supported on socket	10045
WSAEPFNOSUPPORT	Protocol family not supported	10046
WSAEAFNOSUPPORT	Address family not supported	10047
WSAEADDRINUSE	Address already in use	10048
WSAEADDRNOTAVAIL	Can't assign requested address	10049

Name	Description	Code Number
WSAENETDOWN	Network is down	10050
WSAENETUNREACH	Network is unreachable	10051
WSAENETRESET	Network dropped connection on reset	10052
WSAECONNABORTED	Software caused connection abort	10053
WSAECONNRESET	Connection reset by peer	10054
WSAENOBUFS	No buffer space available	10055
WSAEISCONN	Socket is already connected	10056
WSAENOTCONN	Socket is not connected	10057
WSAESHUTDOWN	Can't send after socket shutdown	10058
WSAETOOMANYREFS	Too many references: can't splice	10059
WSAETIMEDOUT	Connection timed out	10060
WSAECONNREFUSED	Connection refused	10061
WSAELOOP	Too many levels of symbolic links	10062
WSAENAMETOOLONG	File name is too long	10063
WSAEHOSTDOWN	Host is down	10064
WSAEHOSTUNREACH	No route to host	10065
WSAENOTEMPTY	Directory is not empty	10066
WSAEPROCLIM	Too many processes	10067
WSAEUSERS	Too many users	10068
WSAEDQUOT	Disk quota exceeded	10069
WSAESTALE	Stale NFS file handle	10070
WSAEREMOTE	Too many levels of remote in path	10071
WSASYSNOTREADY	Network subsystem is unusable	10091
WSAVERNOTSUPPORTED	Winsock DLL cannot support this application	10092
WSANOTINITIALISED	Winsock not initialized	10093
WSAEDISCON	Graceful shutdown in progress	10101
WSAENOMORE	All results have been retrieved. Note that this error code will be removed in future versions. Use WSA_E_NO_MORE instead.	10102
WSAECANCELLED	A call to WSALookupServiceEnd was made while this call was still processing. The call has been canceled.	10103
WSAEINVALIDPROCTABLE	The procedure call table is invalid	10104
WSAEINVALIDPROVIDER	The requested service provider is invalid	10105
WSAEPROVIDERFAILEDINIT	Unable to initialize a service provider	10106
WSASYSCALLFAILURE	System call failure	10107
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.	10108
WSATYPE_NOT_FOUND	Specified class was not found	10109
WSA_E_NO_MORE	No more results can be returned by WSALookup-ServiceNext()	10110
WSA_E_CANCELLED	A call to WSALookupServiceEnd() was made while this call was still processing. The call has been canceled.	10111

Name	Description	Code Number
WSAEREFUSED	A database query failed because it was actively refused.	10112
WSAHOST_NOT_FOUND	Host not found	11001
WSATRY_AGAIN	Non authoritative—host not found	11002
WSANO_RECOVERY	Non-recoverable error	11003
WSANO_DATA	Valid name, no data record of requested type	11004
WSA_QOS_RECEIVERS	At least one Reserve has arrived	11005
WSA_QOS_SENDERS	At least one Path has arrived	11006
WSA_QOS_NO_SENDERS	There are no senders.	11007
WSA_QOS_NO_RECEIVERS	There are no receivers.	11008
WSA_QOS_REQUEST_CONFIRMED	Reserve has been confirmed	11009
WSA_QOS_ADMISSION_FAILURE	Error due to lack of resources	11010
WSA_QOS_POLICY_FAILURE	Rejected for administrative reasons—bad credentials	11011
WSA_QOS_BAD_STYLE	Unknown or conflicting style	11012
WSA_QOS_BAD_OBJECT	Problem with some part of the filterspec or provider-specific buffer in general	11013
WSA_QOS_TRAFFIC_CTRL_ERROR	Problem with some part of the flowspec	11014
WSA_QOS_GENERIC_ERROR	General error	11015
WSA_QOS_ESERVICETYPE	Invalid service type in flowspec	11016
WSA_QOS_EFLOWSPEC	Invalid flowspec	11017
WSA_QOS_EPROVSPECBUF	Invalid provider-specific buffer	11018
WSA_QOS_EFILTERSTYLE	Invalid filter style	11019
WSA_QOS_EFILTERTYPE	Invalid filter type	11020
WSA_QOS_EFILTERCOUNT	Incorrect number of filters	11021
WSA_QOS_EOBJLENGTH	Invalid object length	11022
WSA_QOS_EFLOWCOUNT	Incorrect number of flows	11023
WSA_QOS_EUNKOWNPSOBJ	Unknown object in provider-specific buffer	11024
WSA_QOS_EPOLICYOBJ	Invalid policy object in provider-specific buffer	11025
WSA_QOS_EFLOWDESC	Invalid flow descriptor in the list	11026
WSA_QOS_EPSFLOWSPEC	Inconsistent flow spec in provider-specific buffer	11027
WSA_QOS_EPSFILTERSPEC	Invalid filter spec in provider-specific buffer	11028
WSA_QOS_ESDMODEOBJ	Invalid shape discard mode object in provider-specific buffer	11029
WSA_QOS_ESHAPERATEOBJ	Invalid shaping rate object in provider-specific buffer	11030
WSA_QOS_RESERVED_PETYPE	Reserved policy element in provider-specific buffer	11031

The following list of TAPI errors is complete. As such, some of these errors apply to functions that are not discussed in this book.

TAPI Errors

Constant Name	Description	DWord Value
LINEERR_ALLOCATED	The line cannot be opened due to a persistent condition.	$80000001
LINEERR_BADDEVICEID	The specified device ID or line device ID is invalid or out of range.	$80000002
LINEERR_BEARERMODEUNAVAIL	The bearer mode of a call cannot be changed to the specified bearer mode.	$80000003
LINEERR_CALLUNAVAIL	All call appearances on the specified address are currently in use.	$80000005
LINEERR_COMPLETIONOVERRUN	The maximum number of outstanding call completions has been exceeded.	$80000006
LINEERR_CONFERENCEFULL	The maximum number of parties for a conference has been reached, or requested number of parties cannot be satisfied.	$80000007
LINEERR_DIALBILLING	The dialable address parameter of a function contains dialing control characters that were not processed by the service provider.	$80000008
LINEERR_DIALDIALTONE	The dialable address parameter contains dialing control characters that are not processed by the service provider.	$80000009
LINEERR_DIALPROMPT	The dialable address parameter contains dialing control characters that are not processed by the service provider.	$8000000A
LINEERR_DIALQUIET	The dialable address parameter contains dialing control characters that are not processed by the service provider.	$8000000B
LINEERR_INCOMPATIBLEAPI-VERSION	The application requested an API version or version range that is either incompatible or cannot be supported by the Telephony API implementation and/or corresponding service provider.	$8000000C
LINEERR_INCOMPATIBLEEXT-VERSION	The application requested an extension version range that is either invalid or cannot be supported by the corresponding service provider.	$8000000D
LINEERR_INIFILECORRUPT	The TELEPHON.INI file cannot be read or understood properly by TAPI because of internal inconsistencies or formatting problems. For example, the [Locations], [Cards], or [Countries] section of the TELEPHON.INI file may be corrupted or inconsistent.	$8000000E
LINEERR_INUSE	The line device is in use and cannot currently be configured, allow a party to be added, allow a call to be answered, allow a call to be placed, or allow a call to be transferred.	$8000000F
LINEERR_INVALADDRESS	The specified address is either invalid or not allowed.	$80000010

Constant Name	Description	DWord Value
LINEERR_INVALADDRESSID	A specified address is either invalid or not allowed. If invalid, the address contains invalid characters or digits, or the destination address contains dialing control characters (W, @, $, or ?) that are not supported by the service provider. If not allowed, the specified address is either not assigned to the specified line or is not valid for address redirection.	$80000011
LINEERR_INVALADDRESSMODE	The specified address ID is either invalid or out of range.	$80000012
LINEERR_INVALADDRESSSTATE	dwAddressStates contains one or more bits that are not LINEADDRESSSTATE_ constants.	$80000013
LINEERR_INVALAPPHANDLE	The application handle (such as specified by an hLineApp parameter) or the application registration handle is invalid.	$80000014
LINEERR_INVALAPPNAME	The specified application name is invalid. If an application name is specified by the application, it is assumed that the string does not contain any non-displayable characters, and is zero-terminated.	$80000015
LINEERR_INVALBEARERMODE	The specified bearer mode is invalid.	$80000016
LINEERR_INVALCALLCOMPLMODE	The specified completion is invalid.	$80000017
LINEERR_INVALCALLHANDLE	The specified call handle is not valid. For example, the handle is not NULL but does not belong to the given line. In some cases, the specified call device handle is invalid.	$80000018
LINEERR_INVALCALLPARAMS	The specified call parameters are invalid.	$80000019
LINEERR_INVALCALLPRIVILEGE	The specified call privilege parameter is invalid.	$8000001A
LINEERR_INVALCALLSELECT	The specified select parameter is invalid.	$8000001B
LINEERR_INVALCALLSTATE	The current state of a call is not in a valid state for the requested operation.	$8000001C
LINEERR_INVALCALLSTATELIST	The specified call state list is invalid.	$8000001D
LINEERR_INVALCARD	The permanent card ID specified in dwCard could not be found in any entry in the [Cards] section in the registry.	$8000001E
LINEERR_INVALCOMPLETIONID	The completion ID is invalid.	$8000001F
LINEERR_INVALCONFCALL-HANDLE	The specified call handle for the conference call is invalid or is not a handle for a conference call.	$80000020
LINEERR_INVALCONSULTCALL-HANDLE	The specified consultation call handle is invalid.	$80000021
LINEERR_INVALCOUNTRYCODE	The specified country code is invalid.	$80000022
LINEERR_INVALDEVICECLASS	The line device has no associated device for the given device class, or the specified line does not support the indicated device class.	$80000023
LINEERR_INVALDEVICEHANDLE	The specified device handle is invalid. (Not included in TAPI Help file.)	$80000024
LINEERR_INVALDIALPARAMS	The specified dialing parameters are invalid.	$80000025
LINEERR_INVALDIGITLIST	The specified digit list is invalid.	$80000026

Constant Name	Description	DWord Value
LINEERR_INVALDIGITMODE	The specified digit mode is invalid.	$80000027
LINEERR_INVALDIGITS	The specified termination digits are not valid.	$80000028
LINEERR_INVALEXTVERSION	The specified extension version is not valid.	$80000029
LINEERR_INVALGROUPID	The specified group ID is invalid.	$8000002A
LINEERR_INVALLINEHANDLE	The specified call, device, line device, or line handle is invalid.	$8000002B
LINEERR_INVALLINESTATE	The current line state does not permit changing the device configuration.	$8000002C
LINEERR_INVALLOCATION	The permanent location ID specified in dwLocation could not be found in any entry in the [Locations] section in the registry.	$8000002D
LINEERR_INVALMEDIALIST	The specified media list is invalid.	$8000002E
LINEERR_INVALMEDIAMODE	The list of media types to be monitored contains invalid information, the specified media mode parameter is invalid, or the service provider does not support the specified media mode.	$8000002F
LINEERR_INVALMESSAGEID	The number given in dwMessageID is outside the range specified by the dwNumCompletionMessages field in the LINEADDRESSCAPS structure.	$80000030
LINEERR_INVALPARAM	A parameter (such as dwTollListOption, dwTranslate-Options, dwNumDigits, or a structure pointed to by lpDeviceConfig) contains invalid values, a country code is invalid, a window handle is invalid, or the specified forward list parameter contains invalid information.	$80000032
LINEERR_INVALPARKID	The specified park ID is invalid. (Not included in TAPI Help file.)	$80000033
LINEERR_INVALPARKMODE	The specified park mode is invalid.	$80000034
LINEERR_INVALPOINTER	One or more of the specified pointer parameters (such as lpCallList, lpdwAPIVersion, lpExtensionID, lpdwExt-Version, lphIcon, lpLineDevCaps, and lpToneList) are invalid, or a required pointer to an output parameter is NULL.	$80000035
LINEERR_INVALPRIVSELECT	An invalid flag or combination of flags was set for the dwPrivileges parameter.	$80000036
LINEERR_INVALRATE	The specified bearer mode is invalid.	$80000037
LINEERR_INVALREQUESTMODE	The specified request mode is invalid.	$80000038
LINEERR_INVALTERMINALID	The specified terminal identifier parameter (dwTerminalID) is invalid.	$80000039
LINEERR_INVALTERMINALMODE	One or more of the terminal modes (LINETERM-MODE_ constants) specified in the dwTerminalModes parameter is invalid.	$8000003A
LINEERR_INVALTIMEOUT	Timeouts are not supported or the values of either or both of the parameters dwFirstDigitTimeout or dwInterDigitTimeout fall outside the valid range specified by the call's line-device capabilities.	$8000003B

Constant Name	Description	DWord Value
LINEERR_INVALTONE	The specified custom tone does not represent a valid tone or is made up of too many frequencies or the specified tone structure does not describe a valid tone.	$8000003C
LINEERR_INVALTONELIST	The specified tone list is invalid.	$8000003D
LINEERR_INVALTONEMODE	The specified tone mode parameter is invalid.	$8000003E
LINEERR_INVALTRANSFERMODE	The specified transfer mode parameter is invalid.	$8000003F
LINEERR_LINEMAPPERFAILED	LINEMAPPER was the value passed in the dwDeviceID parameter, but no lines were found that match the requirements specified in the lpCallParams parameter.	$80000040
LINEERR_NOCONFERENCE	The specified call is not a conference call handle or a participant call.	$80000041
LINEERR_NODEVICE	The specified device ID, which was previously valid, is no longer accepted because the associated device has been removed from the system since TAPI was last initialized. Alternately, the line device has no associated device for the given device class.	$80000042
LINEERR_NODRIVER	Either TAPIADDR.DLL could not be located or the telephone service provider for the specified device found that one of its components is missing or corrupt in a way that was not detected at initialization time. The user should be advised to use the telephony control panel to correct the problem.	$80000043
LINEERR_NOMEM	Insufficient memory to perform the operation or unable to lock memory.	$80000044
LINEERR_NOREQUEST	Either there is currently no request pending of the indicated mode, or the application is no longer the highest priority application for the specified request mode.	$80000045
LINEERR_NOTOWNER	The application does not have owner privilege to the specified call.	$80000046
LINEERR_NOTREGISTERED	The application is not registered as a request recipient for the indicated request mode.	$80000047
LINEERR_OPERATIONFAILED	The operation failed for an unspecified or unknown reason.	$80000048
LINEERR_OPERATIONUNAVAIL	The operation is not available, such as for the given device or specified line.	$80000049
LINEERR_RATEUNAVAIL	The service provider currently does not have enough bandwidth available for the specified rate.	$8000004A
LINEERR_RESOURCEUNAVAIL	Insufficient resources to complete the operation. For example, a line cannot be opened due to a dynamic resource over-commitment.	$8000004B
LINEERR_REQUESTOVERRUN	Request overrun. (Not defined in TAPI Help file.)	$8000004C
LINEERR_STRUCTURETOOSMALL	The dwTotalSize field indicates insufficient space to contain the fixed portion of the specified structure.	$8000004D
LINEERR_TARGETNOTFOUND	A target for the call handoff was not found.	$8000004E
LINEERR_TARGETSELF	The application invoking this operation is the target of the indirect handoff.	$8000004F

Constant Name	Description	DWord Value
LINEERR_UNINITIALIZED	The operation was invoked before any application called lineInitialize() or lineInitializeEx().	$80000050
LINEERR_USERUSERINFOTOOBIG	The string containing user-to-user information exceeds the maximum number of bytes specified.	$80000051
LINEERR_REINIT	Attempt to reinitialize TAPI not permitted.	$80000052
LINEERR_ADDRESSBLOCKED	The specified address is blocked from being dialed on the specified call.	$80000053
LINEERR_BILLINGREJECTED	Attempt to bill rejected. (Not included in TAPI Help file.)	$80000054
LINEERR_INVALFEATURE	The dwFeature parameter is invalid.	$80000055
LINEERR_NOMULTIPLEINSTANCE	A telephony service provider, which does not support multiple instances, is listed more than once in the [Providers] section in the registry. The application should advise the user to use the telephony control panel to remove the duplicated driver.	$80000056
LINEERR_INVALAGENTID	The specified agent identifier is not valid.	$80000057
LINEERR_INVALAGENTGROUP	The specified agent group information is not valid or contains errors.	$80000058
LINEERR_INVALPASSWORD	The specified password is not correct and the requested action has not been carried out.	$80000059
LINEERR_INVALAGENTSTATE	The specified agent state is not valid or contains errors.	$8000005A
LINEERR_INVALAGENTACTIVITY	The specified agent activity is not valid.	$8000005B
LINEERR_DIALVOICEDETECT	No description available. (Not included in TAPI Help file.)	$8000005C
LINEERR_USERCANCELLED	Operation canceled by user. (Not included in TAPI Help file.)	$8000005D
LINEERR_INVALADDRESSTYPE	Invalid address type. (Not included in TAPI Help file.)	$8000005E
LINEERR_INVALAGENTSESSION-STATE	Agent session is invalid. (Not included in TAPI Help file.)	$8000005F
LINEERR_DISCONNECTED	Line has been disconnected. (Not included in TAPI Help file).	$80000060

The following error-handling method responds to each of the errors listed above:

```
procedure TTapiInterface.ReportError(ErrorNumber : DWord);
begin
case ErrorNumber of
  LINEERR_ALLOCATED: ErrorStr := 'The line cannot be opened due ' +
    'to a persistent condition, such as  a serial port being opened ' +
    'exclusively by another process.';
  LINEERR_BADDEVICEID: ErrorStr := 'The specified device ID or ' +
    ' line device ID is invalid or out of range. ';
  LINEERR_BEARERMODEUNAVAIL: ErrorStr := 'The call's bearer mode ' +
    'cannot be changed to the specified bearer mode.';
  LINEERR_CALLUNAVAIL: ErrorStr := 'All call appearances on the' +
    ' specified address are currently in use.';
  LINEERR_COMPLETIONOVERRUN: ErrorStr := 'The maximum number of ' +
```

```
    'outstanding call completions has been exceeded. ';
  LINEERR_CONFERENCEFULL: ErrorStr := 'The maximum number of ' +
    'parties for a conference has been reached, or the requested number of ' +
    ' parties cannot be satisfied. ';
  LINEERR_DIALBILLING: ErrorStr := 'The dialable address ' +
    'parameter contains dialing control characters that are not processed by ' +
    'the service provider.';
  LINEERR_DIALDIALTONE: ErrorStr := 'The dialable address' +
    ' parameter contains dialing control characters that are not processed by' +
    ' the service provider. ';
  LINEERR_DIALPROMPT: ErrorStr := 'The dialable address ' +
    'parameter contains dialing control characters that are not processed by' +
    ' the service provider. ';
  LINEERR_DIALQUIET: ErrorStr := 'The dialable address' +
    ' parameter contains dialing control characters that are not processed' +
    ' by the service provider.';
  LINEERR_INCOMPATIBLEAPIVERSION: ErrorStr := 'The application requested' +
    ' an API version or version range that is either incompatible or cannot' +
    ' be supported by the TAPI implementation and/or service provider. ';
  LINEERR_INCOMPATIBLEEXTVERSION: ErrorStr := 'The application ' +
    'requested an extension version range that is either invalid or cannot ' +
    'be supported by the corresponding service provider. ';
  LINEERR_INIFILECORRUPT: ErrorStr := 'The TELEPHON.INI file cannot' +
    ' be read or understood properly by TAPI. ';
  LINEERR_INUSE: ErrorStr := 'The line device is in' +
    ' use and cannot currently be configured or otherwise manipulated.';
  LINEERR_INVALADDRESS: ErrorStr := 'A specified address is ' +
    'either invalid or not allowed.';
  LINEERR_INVALADDRESSID: ErrorStr := 'The specified address ID' +
    ' is either invalid or out of range. ';
  LINEERR_INVALADDRESSMODE: ErrorStr := 'The specified address mode' +
    ' is invalid. ';
  LINEERR_INVALADDRESSSTATE: ErrorStr := 'dwAddressStates contains' +
    ' one or more bits that are not LINEADDRESSSTATE_ constants. ';
  LINEERR_INVALAPPHANDLE: ErrorStr := 'The application handle ' +
    'or the application registration handle is invalid. ';
  LINEERR_INVALAPPNAME: ErrorStr := 'Invalid Application Name';
  LINEERR_INVALBEARERMODE: ErrorStr := 'The specified bearer mode ' +
    'is invalid. ';
  LINEERR_INVALCALLCOMPLMODE: ErrorStr := 'The specified completion '  +
    'is invalid.';
  LINEERR_INVALCALLHANDLE: ErrorStr := 'The specified call handle ' +
    'is not valid.';
  LINEERR_INVALCALLPARAMS: ErrorStr := 'The specified call ' +
    'parameters are invalid. ';
  LINEERR_INVALCALLPRIVILEGE: ErrorStr := 'The specified select ' +
    'parameter is invalid. ';
  LINEERR_INVALCALLSELECT: ErrorStr := 'The specified select ' +
    'parameter is invalid.';
  LINEERR_INVALCALLSTATE: ErrorStr := 'The current state of a ' +
    'call is not in a valid state for the requested operation. ';
  LINEERR_INVALCALLSTATELIST: ErrorStr := 'The specified call state' +
    ' list is invalid.';
  LINEERR_INVALCARD: ErrorStr := 'The permanent card ID ' +
    'specified in dwCard could not be found in any entry in the [Cards] ' +
    'section in the registry. ';
  LINEERR_INVALCOMPLETIONID: ErrorStr := 'The completion ID is invalid.';
  LINEERR_INVALCONFCALLHANDLE: ErrorStr := 'The specified conference ' +
    'call handle is invalid or is not a handle for a conference call. ';
  LINEERR_INVALCONSULTCALLHANDLE: ErrorStr := 'The specified consultation' +
```

B Appendix

```
    ' call handle is invalid. ';
  LINEERR_INVALCOUNTRYCODE: ErrorStr := 'The specified country code' +
    ' is invalid. ';
  LINEERR_INVALDEVICECLASS: ErrorStr := 'The line device has no ' +
    ' associated device for the given device class, or the specified line' +
    ' does not support the indicated device class. ';
  LINEERR_INVALDEVICEHANDLE: ErrorStr := 'Invalid device handle';
  LINEERR_INVALDIALPARAMS: ErrorStr := 'Invalid dial parameters';
  LINEERR_INVALDIGITLIST: ErrorStr := 'Invalid digit list';
  LINEERR_INVALDIGITMODE: ErrorStr := 'Invalid digit mode';
  LINEERR_INVALDIGITS: ErrorStr := 'Invalid digits';
  LINEERR_INVALEXTVERSION: ErrorStr := 'Invalid EXT version';
  LINEERR_INVALGROUPID: ErrorStr := 'Invalid group ID';
  LINEERR_INVALLINEHANDLE: ErrorStr := 'Invalid line handle';
  LINEERR_INVALLINESTATE: ErrorStr := 'Invalid line state';
  LINEERR_INVALLOCATION: ErrorStr := 'Invalid location';
  LINEERR_INVALMEDIALIST: ErrorStr := 'Invalid media list';
  LINEERR_INVALMEDIAMODE: ErrorStr := 'Invalid media mode';
  LINEERR_INVALMESSAGEID: ErrorStr := 'Invalid message ID';
  LINEERR_INVALPARAM: ErrorStr := 'A parameter contains an' +
    'invalid value';
  LINEERR_INVALPARKID: ErrorStr := 'Invalid Park ID';
  LINEERR_INVALPARKMODE: ErrorStr := 'Invalid Park mode';
  LINEERR_INVALPOINTER: ErrorStr := 'One or more of the specified' +
    ' pointer parameters is/are invalid';
  LINEERR_INVALPRIVSELECT: ErrorStr := 'An invalid flag or ' +
    'combination of flags was set for the dwPrivileges parameter.';
  LINEERR_INVALRATE: ErrorStr := 'The specified bearer mode ' +
    'is invalid. ';
  LINEERR_INVALREQUESTMODE: ErrorStr :=
    'The specified request mode is invalid. ';
  LINEERR_INVALTERMINALID: ErrorStr :=
    'The specified terminal mode parameter is invalid. ';
  LINEERR_INVALTERMINALMODE: ErrorStr :=
    'The specified terminal modes parameter is invalid. ';
  LINEERR_INVALTIMEOUT: ErrorStr := 'Timeouts are not supported ' +
    'or the values of one or both of the parameters dwFirstDigitTimeout or ' +
    'dwInterDigitTimeout are invalid.';
  LINEERR_INVALTONE: ErrorStr := 'The specified custom tone ' +
    'is invalid contains too many frequencies.';
  LINEERR_INVALTONELIST: ErrorStr :=
    'The specified tone list is invalid. ';
  LINEERR_INVALTONEMODE: ErrorStr := 'The specified tone mode ' +
    'parameter is invalid.';
  LINEERR_INVALTRANSFERMODE: ErrorStr := 'The specified transfer mode ' +
    ' parameter is invalid.';
  LINEERR_LINEMAPPERFAILED: ErrorStr := 'no lines were found that ' +
    'match the requirements specified when using the LINEMAPPER constant.';
  LINEERR_NOCONFERENCE: ErrorStr :=
    'The specified call is not a conference call handle or a participant call.';
  LINEERR_NODEVICE: ErrorStr :=
    'The specified previously valid device ID can no longer be accepted.';
  LINEERR_NODRIVER: ErrorStr :=
    'Driver problem for the specified device; use the Telephony Control Panel' +
    ' to correct the problem.';
  LINEERR_NOMEM: ErrorStr :=
    'Insufficient memory to perform the operation.';
  LINEERR_NOREQUEST: ErrorStr :=
    'No request pending of the indicated modeor application problem.';
  LINEERR_NOTOWNER: ErrorStr :=
```

```
    'The application does not have owner privilege to the specified call. ';
  LINEERR_NOTREGISTERED: ErrorStr :=
    'Application  not registered as  request recipient for the indicated  mode.';
  LINEERR_OPERATIONFAILED: ErrorStr :=
    'Operation failed for an unspecified or unknown reason.';
  LINEERR_OPERATIONUNAVAIL: ErrorStr :=
    'Operation  not available for the given device or specified line.';
  LINEERR_RATEUNAVAIL: ErrorStr :=
    'Insufficient bandwidth available for the specified rate.';
  LINEERR_RESOURCEUNAVAIL: ErrorStr :=
    'Insufficient resources to complete the operation.';
  LINEERR_REQUESTOVERRUN: ErrorStr :=
    'Line Request overrun.';
  LINEERR_STRUCTURETOOSMALL: ErrorStr :=
    'The dwTotalSize field indicates insufficient space to contain the ' +
    'fixed portion of the specified structure.';
  LINEERR_TARGETNOTFOUND: ErrorStr :=
    'A target for the call handoff was not found.';
  LINEERR_TARGETSELF: ErrorStr :=
    'The telephony application invoking this operation is the target ' +
    'of the indirect handoff.';
  LINEERR_UNINITIALIZED: ErrorStr :=
    'The operation was invoked before any application called lineInitialize,' +
    ' lineInitializeEx.';
  LINEERR_USERUSERINFOTOOBIG: ErrorStr :=
    'The string containing user-to-user information exceeds the maximum number' +
    ' of bytes specified in one of fields.';
  LINEERR_REINIT: ErrorStr :=
    'Improper attempt to reinitialize TAPI; must close TAPI down first.';
  LINEERR_ADDRESSBLOCKED: ErrorStr :=
    'This address is blocked.';
  LINEERR_BILLINGREJECTED: ErrorStr :=
    'Billing attempt rejected.';
  LINEERR_INVALFEATURE: ErrorStr :=
    'Requested feature not available.';
  LINEERR_NOMULTIPLEINSTANCE: ErrorStr :=
    'Multiple instances not permitted';
  {$IFDEF TAPI20}
  LINEERR_INVALAGENTID: ErrorStr :=
    'Invalid agent ID';
  LINEERR_INVALAGENTGROUP: ErrorStr :=
    'Invalid agent group.';
  LINEERR_INVALPASSWORD: ErrorStr :=
    'Invalid password.';
  LINEERR_INVALAGENTSTATE: ErrorStr :=
    'Invalid agent state.';
  LINEERR_INVALAGENTACTIVITY: ErrorStr :=
    'Invalid agent activity.';
  LINEERR_DIALVOICEDETECT: ErrorStr :=
    'Dial Voice Mode Detected.';
  {$ENDIF}
  {$IFDEF TAPI22}
  LINEERR_USERCANCELLED: ErrorStr :=
    'Line request canceled by user.';
  {$ENDIF}
  {$IFDEF TAPI30}
  LINEERR_INVALADDRESSTYPE: ErrorStr :=
    'Invalid address type.';
  {$ENDIF}
  {$IFDEF TAPI22}
```

```
  LINEERR_INVALAGENTSESSIONSTATE: ErrorStr :=
    'Invalid agent session state.';
  LINEERR_DISCONNECTED: ErrorStr :=
    'Line disconnected.';
  {$ENDIF}
  end; { case }
end;
```

This centralized error-handling routine makes it possible to create more succinct code. Compare an older version of one of the methods in the TAPI interface unit with a newer version that calls the above method:

```
(* Old Version that includes error handling within it *)
function TTapiInterface.GetAddressID: boolean;
begin
  TapiResult := lineGetAddressID(fLine, fAddressID,
    LINEADDRESSMODE_DIALABLEADDR, PChar(FPhoneNumber),
    SizeOf(FPhoneNumber));
  result := TapiResult=0;
  if result then exit;
  case TAPIResult of //
    LINEERR_UNINITIALIZED: TempStr := 'UNINITIALIZED';
    LINEERR_INVALPOINTER: TempStr := 'INVALPOINTER';
    LINEERR_INVALADDRESSMODE: TempStr := 'INVALADDRESSMODE';
    LINEERR_NOMEM: TempStr := 'NOMEM';
    LINEERR_INVALCALLHANDLE: TempStr := 'INVALCALLHANDLE';
    LINEERR_OPERATIONUNAVAIL: TempStr := 'OPERATIONUNAVAIL';
    LINEERR_OPERATIONFAILED: TempStr := 'OPERATIONFAILED';
    LINEERR_INVALLINEHANDLE: TempStr := 'INVALLINEHANDLE';
    LINEERR_RESOURCEUNAVAIL: TempStr := 'RESOURCEUNAVAIL';
  end; // case
  ShowMessage('Could not get Address ID because of error: ' + TempStr);
end;

(* New version of method that calls centralized error handling routine *)

function TTapiInterface.GetAddressID: boolean;
begin
  TapiResult := lineGetAddressID(fLine, fAddressID,
    LINEADDRESSMODE_DIALABLEADDR, PChar(FPhoneNumber),
    SizeOf(FPhoneNumber));
  result := TapiResult=0;
  if not result then ReportError(TapiResult);
end;
```

Appendix C

Bibliography of Printed and Online Communications Programming Resources

Delphi TAPI Articles

"Delphi and TAPI Part I: An Introduction to Telephony Programming"
by Major Ken Kyler and Alan C. Moore, Ph.D. *Delphi Informant Magazine*, July 1998, available online at http://www.delphizine.com/features/1998/07/di199807am_f/di199807am_f.asp.

"Delphi and TAPI Part II: Building a Telephony Application"
by Major Ken Kyler and Alan C. Moore, Ph.D. *Delphi Informant Magazine*, August 1998, available online at http://www.delphizine.com/features/1998/08/di199808am_f/di199808am_f.asp.

"Delphi and TAPI Part III: Wrapping Up Telephony"
by Major Ken Kyler and Alan C. Moore, Ph.D. *Delphi Informant Magazine*, September 1998, available online at http://www.delphizine.com/features/1998/09/di199809am_f/di199809am_f.asp.

"Extending TAPI Playing and Recording Sounds During Telephony Calls"
by Robert Keith Elias and Alan C. Moore, Ph.D. *Delphi Informant Magazine*, November 1999, available online at http://www.delphizine.com/features/1999/11/di199911re_f/di199911re_f.asp.

"Single-Tier Database Apps—Putting the ClientDataSet Component to Work"
by Bill Todd (some reference to TAPI, using a TAPI dialer in a database application), *Delphi Informant Magazine*, January 1998, available online at http://www.delphizine.com/features/1998/01/di199801bt_f/di199801bt_f.asp.

Microsoft White Papers on TAPI

"IP Telephony with TAPI 3.0"
(white paper on Microsoft site) http://www.microsoft.com/windows2000/techinfo/howitworks/communications/telephony/iptelephony.asp

"Other Microsoft White Papers on TAPI 3.0"
http://www.microsoft.com/windows2000/techinfo/howitworks/communications/telephony/

"Introductory Articles and Links to Other Microsoft White Papers on Communications and Networking"
http://www.microsoft.com/windows2000/technologies/communications/default.asp

Telephony Articles

Toward 2000 Part 8: Telephony
http://www.nss.co.uk/Windows2000/Telephony.htm

Programmer's Heaven – Delphi and Kylix Zone, TAPI Files
http://www.programmersheaven.com/zone2/cat70/index.htm

TAPI Programming Resources

There are a number of Delphi TAPI solutions on Torry's Pages at http://www.torry.net/tapi.htm.

TAPI Books (printed and online)

Windows Telephony Programming: A Developer's Guide to TAPI
by Chris Sells, [ISBN: 0-201-63450-3], Addison-Wesley. A classic work by one of the early TAPI gurus. Excellent advice and code examples.

Communications Programming for Windows 95
by Charles A. Mirho and Andre Terrisse, [ISBN: 1-55615-668-5], Microsoft Press. An early reference that contains excellent information on early TAPI versions along with information on basic communications programming with the Serial Port, Simple Messaging, and TAPI.

"MAPI, SAPI, and TAPI Developer's Guide"
by Michael C. Amundsen, available online at http://developer.grup.com.tr/misc/mapi/.

Winsock Books

Effective TCP/IP Programming
by Jon C. Snader, [ISBN: 0-201-61589-4], Addison-Wesley, 2000.

TCP/IP Illustrated Volume 1—The Protocols
by W. Richard Stevens, [ISBN: 0-201-63346-9], Addison-Wesley, 2001.

TCP/IP Illustrated Volume 2
by Gary R. Wright and W. Richard Stevens, [ISBN: 0-201-63354-X], Addison-Wesley, 2000.

Internetworking with TCP/IP—Principles, Protocols, and Architectures
by Douglas E. Comer, [ISBN: 0-13-018380-6], Prentice-Hall, 2000.

Unix Network Programming
by W. Richard Stevens, [ISBN:0-13-949876-1], Prentice-Hall, 1990.

Network Programming for Microsoft Windows
by Anthony Jones and Jim Ohlund, [ISBN: 0-7356-0560-2], Microsoft Press, 1999.

Windows Sockets Network Programming
by Bob Quinn and David Shute, [ISBN: 0-201-63372-8], Addison-Wesley, 1996.

Programming Winsock
by Arthur Dumas, [ISBN: 0-672-30594-1], SAMS, 1995.

Other Internet Programming and Winsock Sources

There are numerous sites on the Internet that cater to Winsock and TCP/IP issues. Below are just a few of the many:

http:///www.microsoft.com
http://www.sockets.com
http://www.winsock2.com
http://www.sockaddr.com
http://www.tangentsoft.net/wskfaq
http://www.google.com/search?q=winsock+tutorial

Internet Programming and Winsock Newsgroups

alt.winsock
alt.winsock.programming
comp.os.ms-windows.networking.tcp-ip
comp.os.ms-windows.networking.windows:
comp.os.ms-windows.programmer.networks
comp.os.ms-windows.programmer.tools.winsock

RFCs

For more information on the protocols, such as IP, ICMP, UDP, and TCP, take a look at http://www.ietf.org/rfc.html.

Index

G

H

I

L

S

T

About the CD

The CD-ROM that accompanies this book includes example programs demonstrating the use of Winsock and TAPI functions. The examples are organized into folders named for the chapters and are located in the Source Files folder.

Many of the Winsock examples are console programs that simply demonstrate Winsock functions and techniques. Each of these is simply a stand-alone project. There are also a few GUI projects (such as EX36), which are organized into separate folders.

The majority of the TAPI examples are functions in the file TAPIInft.pas, a unit that introduces a large class that wraps many TAPI functions. Some of the example programs make calls into this class to demonstrate various aspects of TAPI, while others emphasize initialization and configuration issues and demonstrate practical tasks like placing and receiving phone calls.

See the Readme file on the CD for more information about the examples.

Warning: By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement on the following page.

Opening the CD package makes this book nonreturnable.

CD/Source Code Usage License Agreement

Please read the following CD/Source Code usage license agreement before opening the CD and using the contents therein:

1. By opening the accompanying software package, you are indicating that you have read and agree to be bound by all terms and conditions of this CD/Source Code usage license agreement.
2. The compilation of code and utilities contained on the CD and in the book are copyrighted and protected by both U.S. copyright law and international copyright treaties, and is owned by Wordware Publishing, Inc. Individual source code, example programs, help files, freeware, shareware, utilities, and evaluation packages, including their copyrights, are owned by the respective authors.
3. No part of the enclosed CD or this book, including all source code, help files, shareware, freeware, utilities, example programs, or evaluation programs, may be made available on a public forum (such as a World Wide Web page, FTP site, bulletin board, or Internet news group) without the express written permission of Wordware Publishing, Inc. or the author of the respective source code, help files, shareware, freeware, utilities, example programs, or evaluation programs.
4. You may not decompile, reverse engineer, disassemble, create a derivative work, or otherwise use the enclosed programs, help files, freeware, shareware, utilities, or evaluation programs except as stated in this agreement.
5. The software, contained on the CD and/or as source code in this book, is sold without warranty of any kind. Wordware Publishing, Inc. and the authors specifically disclaim all other warranties, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the disk, the program, source code, sample files, help files, freeware, shareware, utilities, and evaluation programs contained therein, and/or the techniques described in the book and implemented in the example programs. In no event shall Wordware Publishing, Inc., its dealers, its distributors, or the authors be liable or held responsible for any loss of profit or any other alleged or actual private or commercial damage, including but not limited to special, incidental, consequential, or other damages.
6. One (1) copy of the CD or any source code therein may be created for backup purposes. The CD and all accompanying source code, sample files, help files, freeware, shareware, utilities, and evaluation programs may be copied to your hard drive. With the exception of freeware and shareware programs, at no time can any part of the contents of this CD reside on more than one computer at one time. The contents of the CD can be copied to another computer, as long as the contents of the CD contained on the original computer are deleted.
7. You may not include any part of the CD contents, including all source code, example programs, shareware, freeware, help files, utilities, or evaluation programs in any compilation of source code, utilities, help files, example programs, freeware, shareware, or evaluation programs on any media, including but not limited to CD, disk, or Internet distribution, without the express written permission of Wordware Publishing, Inc. or the owner of the individual source code, utilities, help files, example programs, freeware, shareware, or evaluation programs.
8. You may use the source code, techniques, and example programs in your own commercial or private applications unless otherwise noted by additional usage agreements as found on the CD.

Warning: By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement.

Opening the CD package makes this book nonreturnable.