Powered by Microsoft® Windows® CE

MICROSOFT PROFESSIONAL EDITIONS

Microsoft® Press

The ultimate reference and toolkit for Windows CE

Microsoft® Windows® CE Communications Guide

Microsoft®

Windows® CE Communications Guide

Microsoft® Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Microsoft Windows CE Developer's Kit / Microsoft Corporation.
p. cm.
ISBN 0-7356-0619-6
1. Microsoft Windows (Computer file) 2. Operating systems (Computers) I. Microsoft Corporation.
QA76.76.O63M74515 1999
005.4'469--dc21 99-24745
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 4 3 2 1 0 9

Distributed in Canada by ITP Nelson, a division of Thomson Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com.

Acquisitions Editor: Ben Ryan
Project Editor: Alice Turner

Part No. 097-0002196

Contents

Preface

The *Microsoft® Windows® CE Developer's Kit* provides all the information you need to write applications for devices based on the Microsoft® Windows® CE operating system. The kit includes the following four books:

- *Microsoft® Windows® CE Programmer's Guide*

  Introduces the architecture of the Windows CE operating system.

  Explains the low-level details of creating a Windows CE–based application, including handling processes and threads, managing memory and power, accessing the object store, and modifying the registry.

  Provides information on connecting a Windows CE–based device to a desktop computer, synchronizing data between a device and desktop, and transferring files.

  Provides information on using Unicode and localizing Windows CE–based applications.

- *Microsoft® Windows® CE User Interface Services Guide*

  Describes all tasks associated with creating a user interface (UI) for a Windows CE–based device, including how to create windows and dialog boxes, how to handle messages, and how to add menus, controls, and other resources to a UI.

  Discusses how to handle various user input methods (IMs) such as keyboards and touch screens.

- *Microsoft® Windows® CE Communications Guide*

  Provides basic instructions for implementing communications support on a Windows CE–based device, including how to handle infrared connections, develop telephony applications, implement Remote Access Service (RAS) features into an application, handle networking and security issues, work with Windows Sockets, and establish an Internet connection.

- *Microsoft® Windows® CE Device Driver Kit*

  Provides procedures for writing device drivers for Windows CE–based devices.

  Explains how to create native and stream interface drivers as well as how to implement Universal Serial Bus (USB) and Network Driver Interface Specification (NDIS) drivers.

The CD that accompanies the books includes online versions of the books plus the following content.

Content	Description
Windows CE API	Shows the interfaces, functions, structures, messages, and other application programming interface (API) elements for Windows CE.
Device Driver Kit API	Shows the interfaces, functions, structures, messages, and other API elements needed to create device drivers for Windows CE.
Microsoft Foundation Class (MFC) Library for Windows CE	Shows the classes, global functions, global variables, and macros needed to create full-featured Windows CE–based applications.
Active Template Library (ATL) for Windows CE	Shows the classes, macros, and global functions needed to develop small, fast Microsoft® ActiveX® controls for platforms that run Windows CE.
Mobile Channels	Demonstrates how to use Active Server Pages (ASP) and Channel Definition Format technology to enable offline Web site browsing on a Windows CE–based device.
Writing applications for a Palm-size PC	Demonstrates how to work with the Palm–size PC shell, handle memory and power, programmatically access Palm–size PC navigation controls, and design the UI for applications running on a Palm–size PC.
Writing applications for a Handheld PC	Demonstrates how to work with the Handheld PC (H/PC) shell, handle memory and power, and synchronize data between an H/PC and a desktop computer.
Writing applications for an Auto PC	Demonstrates how to implement speech, control the audio system, interact with a vehicle computer, communicate with a Global Positioning System (GPS) device, and design an effective UI for an Auto PC application.

This book, the *Microsoft Windows CE Communications Guide*, contains the following chapters:

Windows CE Communications Overview

This chapter provides an overview of Windows CE communications technologies and how they fit into the International Organization for Standardization Open Systems Interconnection (ISO/OSI) model for network communications.

Serial Communications

This chapter discusses serial communications functions, serial cables and connectors, as well as creating and implementing a serial communication application.

Telephony API

This chapter provides information about the Windows CE implementation of the Microsoft Telephony API (TAPI). Features discussed include outbound dialing, address translation services, installable service providers, the TAPI programming model, and use of TAPI in Windows CE–based applications.

Remote Access Service

This chapter explains accessing a network from a remote location, Remote Access Service (RAS) features and related functions.

Windows Sockets

This chapter provides an overview of Transmission Control Protocol/Internet Protocol (TCP/IP), upon which Windows Sockets (Winsock) is built, as well as creating applications with Winsock.

Windows Networking

This chapter explains the Windows Networking API (WNet) which includes functions used to manage network connections and retrieve current configuration information about the Microsoft Network.

Internet Connections

This chapter discusses the Windows Internet API (WinInet), its functions, and implementing WinInet as a browser or FTP application.

Security Support Provider Interface

This chapter describes the Security Support Provider Interface (SSPI), that enables applications to access DLLs, known as Security Support Providers (SSPs), containing common authentication and cryptography scripts. SSPs enable applications to use multiple security solutions for package management, credential management, context management, and message support.

Cryptography

This chapter discusses the Windows CE implementation of the Microsoft cryptographic system, the Microsoft Cryptographic API (CAPI), and implementing CAPI to enable application encryption and decryption.

Wireless Services

This chapter provides an overview of Wireless Services for Windows CE. It describes support for receiving e-mail messages, pager messages, testing custom DLLs, and other radio services on a Windows CE–based device.

About the Code Samples Included in this Guide

Most code samples included with the *Windows CE Communications Guide* were developed with Microsoft Visual C++® version 5.0 and the Microsoft Windows CE Toolkit for Visual C++ version 5.0. The Sspi and Crypt samples were developed with Microsoft Visual C++ version 6.0 and the Microsoft Windows CE Toolkit for Visual C++ 6.0. The code in sample applications is ported for a Handheld PC, but the programming concepts that are presented apply to all Windows CE–based platforms. Because of their large size, some code samples shown are incomplete. The CD that accompanies this book contains the entire code for all the samples.

Sample	Description
Tty	Shows how to open, configure, and close a serial communications port and perform read/write operations for a TTY terminal emulation application. The code is partially listed in the Serial Communications chapter.
CeDialer	Shows how to initialize an application's use of TAPI, open a line device, negotiate an API version to use, translate an address into another format, place a call on an opened line device, close an opened line device, shut down an application's usage of the line abstraction of the API. The code is partially listed in the Telephony API chapter.
RasConn	Demonstrates how to start a RAS connection, dial entries from the default phone book, and close an active connection. The code is partially listed in the Remote Access Service chapter.
Winsock	A Winsock server and a Winsock client sample. The server sample shows how to implement a Winsock TCP stream socket server. The client sample shows how to implement a Winsock TCP stream socket client. Both code samples are listed in their entirety in the Windows Sockets chapter.

Sample	Description
IRSock	An Infrared Sockets server and an Infrared Sockets client sample. The server sample shows how to implement an Infrared Sockets server. The client sample shows how to implement an Infrared Sockets client. Both code samples are listed in their entirety in the Windows Sockets chapter.
Multicast	A multicast receiver and a multicast sender sample. The Receive sample shows how to receive an IP multicast datagram. The Send sample shows how to send an IP multicast datagram. Both code samples are listed in their entirety in the Windows Sockets chapter.
CeHttp	Demonstrates how to create and submit a HTTP request. The sample requests a default HTML document from a server and displays it along with the HTTP transaction headers. The code is partially listed in the Internet Connections chapter.
Sspi	Shows how to use the Security Support Provider Interface to access common authentication and cryptographic data schemes. The code is listed in it's entirety in the Security Support Provider Interface chapter.
Crypto	A cryptography encryption and a decryption sample. The Encrypt sample shows how to encrypt data read from a text file. The Decrypt sample shows how to decrypt the file created by the Encrypt sample. Both code samples are listed in their entirety in the Cryptography chapter.

Document Conventions

The following table shows the typographical conventions used throughout this book.

Convention	Description
`monospace`	Indicates source code, structure syntax, examples, user input, and application output. For example, `ptbl->SortTable(pSort, TBL_BATCH);`
Bold	Indicates an interface, method, function, structure, macro, or other keyword in Windows CE, the Microsoft Windows operating system, C, or C++. For example, **CommandBar_Height** is a function. Within discussions of syntax, bold type indicates that text must be entered exactly as shown.
Italic	Indicates placeholders, most often method or function parameters; these placeholders stand for information that must be supplied by the implementation or the user. For example, *lpButtons* is a function parameter. Also indicates new terms defined in the glossary.
UPPERCASE	Indicates flags, return values, messages, and properties. For example, WSAEFAULT is a Windows Sockets error value, MF_CHECKED is a flag, and TB_ADDBUTTONS is a message. In addition, uppercase letters indicate segment names, registers, and terms used at the operating-system command level.
( )	Indicate one or more parameters that you pass to a function, in syntax.

CHAPTER 1

Windows CE Communications Overview

Windows CE supports multiple options for data communications. For example, a Windows CE–based device can use communications for the following operations:

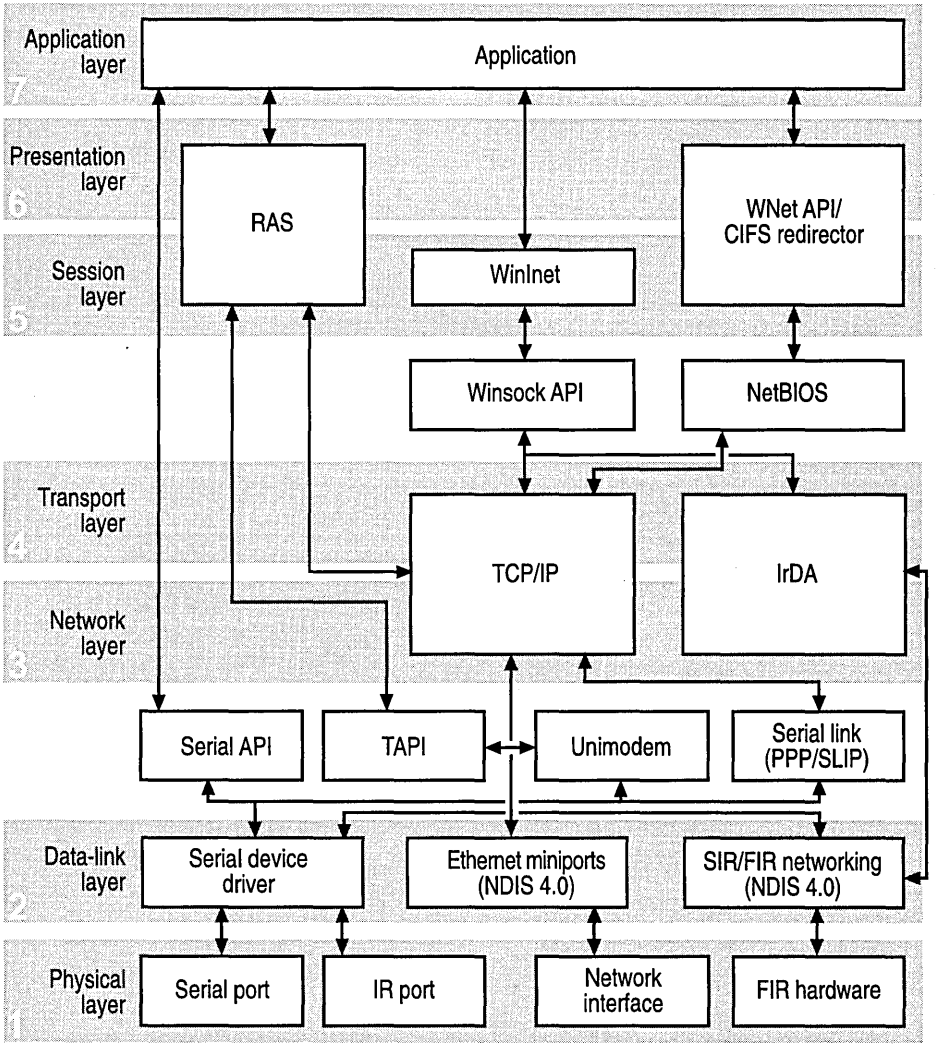
- Downloading files from a desktop computer or network server
- Exchanging data with another Windows CE–based device
- Sending and receiving electronic mail
- Sending data to a network server
- Browsing the Internet and the Web
- Scanning bar codes

To support different types of communication, Windows CE–based devices can include a variety of hardware configurations. For example, most Windows CE–based devices include a serial cable connector, and some devices include an infrared (IR) transceiver. If a hardware expansion slot is available, users can extend the capabilities of a Windows CE–based device with third-party communications hardware, such as a modem or bar code scanner.

Computer communications models are divided into layers. Software applications comprise the top layer of the communications model. Communications hardware comprises the bottom layer of this model. Windows CE includes API methods for moving data between the application layer and the physical hardware layer.

Open Systems Interconnection Model

The International Organization for Standardization (ISO) developed the Open Systems Interconnection (OSI) model. This seven-layer model is an industry standard reference for describing data I/O. The following illustration shows Windows CE communications in reference to the ISO/OSI model.



Physical Layer

The physical layer of the ISO/OSI model is the hardware. Hardware in the physical layer converts electrical signals into binary code, which is passed to the data-link layer. A Windows CE–based device can include the following hardware:

- A Serial port
- An IR transceiver
- A Wireless transceiver
- A Network interface card

For more information on communicating using serial ports, see Serial Communications. For more information on infrared communications, see Serial Communications and Windows Sockets.

Wireless Services for Windows CE supports multiple service inputs to a Windows CE–based device using wireless hardware—typically a PC Card radio device. The wireless hardware device may be limited to receiving signals or transmitting a simple acknowledgment, or it may be a fully bidirectional device. For more information, see Wireless Services.

Data-Link Layer

Windows CE provides data-link layer support for serial I/O and local area networks (LANs). Low-level software, called device drivers, operate at the data-link layer, managing communications with physical layer hardware. For example, the serial driver manages the serial port, and the Microsoft *network driver interface specification* (NDIS) drivers manage network interface connections.

The Windows CE NDIS is a subset of Microsoft NDIS version 4.0, which is used in Windows-based desktop operating systems. Windows CE supports NDIS Ethernet (802.3) miniport drivers. Windows CE also supports Serial Infrared (SIR) and Fast Infrared (FIR) IrDA miniport drivers. For more information on NDIS architecture and network connectivity issues, see the Microsoft Windows CE Device Driver Kit.

Operating at the data-link layer, or between the data-link layer and the network layer, are the Microsoft Telephony API (TAPI), Unimodem and the *point-to-point protocol* (PPP) and *Serial Line Internet Protocol* (SLIP) for direct serial and dial-up connections. For more information on TAPI and Unimodem, see Telephony API. For more information on PPP and SLIP, see Serial Communications and Remote Access Service.

Network and Transport Layers

Software that operates at the network layer fragments, routes, and reassembles data. The transport layer works with the network layer to package and transfer data that it receives from the session layer.

TCP/IP operates at the network and transport layers. TCP/IP is an industry standard communications protocol that defines methods for packaging data for transmission over a network. For more information on TCP/IP, see Windows Sockets.

For infrared communications Windows CE supports the Infrared Data Association (IrDA) standards using the IrDA protocols at the network and transport layers. For more information on using the IrDA protocols, see Windows Sockets.

Session, Presentation, and Application Layers

The session, presentation, and application layers comprise the upper layers of the ISO/OSI model. The upper layers depend on the lower layers: transport, network, data-link, and physical to handle low-level communications. The session layer manages high-level connections, called sessions. The presentation layer formats data received from the application layer before passing it to the layers. Applications that include a user interface operate at the application layer.

Windows Sockets (Winsock) operates at the session layer interface to the transport layer. Winsock, an interface between applications and the transport protocol, works as a conduit for data I/O. For more information on Winsock, see Windows Sockets.

The Microsoft Windows CE Internet API (WinInet), is an API used for Internet client application development. WinInet uses Winsock internally. The WinInet.dll module exports WinInet functions used to develop Internet applications, such as Web browsers and File Transfer Protocol (FTP) applications. For more information on WinInet, see Internet Connections.

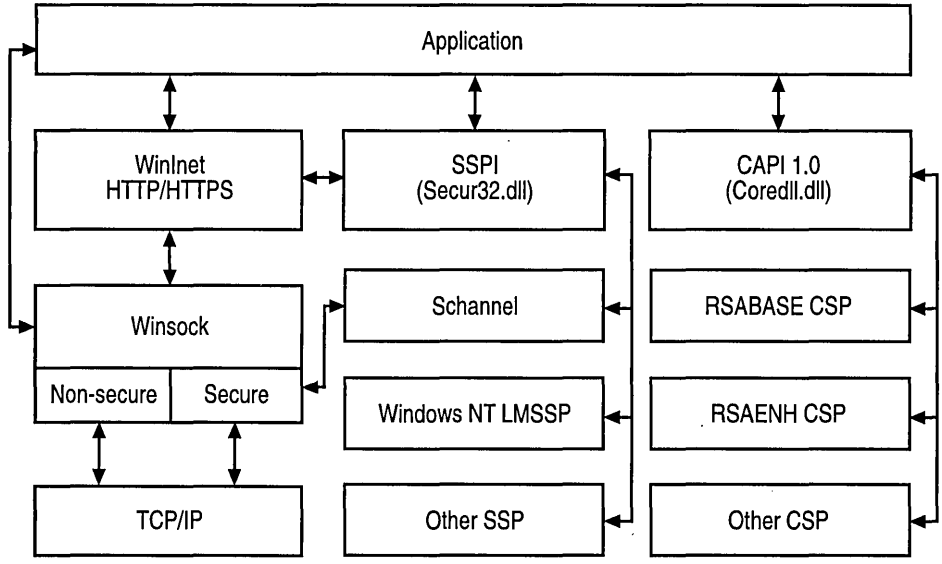
Remote Access Service (RAS) operates in the upper layers of the ISO/OSI model. RAS is an application used to access network resources from a remote location. For more information on RAS, see Remote Access Service.

A Windows CE–based application can use Windows Networking functions to establish and terminate network connections and to retrieve current configuration data for the Microsoft Network. Access to this data is made possible by way of the Windows CE Networking API (WNet). WNet communicates through the *Common Internet File System (CIFS)* redirector to the remote host. A CIFS redirector is a module through which one computer accesses another. For more information on WNet, see Windows Networking.

Providing Secure Communications

Windows CE supports secure socket connections through Winsock and WinInet. For more information on secure sockets, see Windows Sockets and Internet Connections.

Windows CE also supports the Microsoft Cryptographic API (CAPI) and Security Support Provider Interface (SSPI) for secure communications. The following illustration shows the relationship between these elements and your application.



The cryptographic functions supported in Windows CE exist as an integral part of CAPI. Services provided by these functions enable you to add encryption to your Windows CE–based application without requiring extensive knowledge of cryptography.

The algorithms and standards used by CAPI are implemented through cryptographic service providers (CSPs). CAPI functions are available through the Coredll.dll module.

SSPI provides a common interface between transport-level applications and security providers. It provides a mechanism by which a transport application can call one of several security providers and obtain an authentic connection without knowing the details of the security protocol. Security providers included with Windows CE: Windows NT® LAN Manager (NTLM), secure socket layer (SSL) version 2.0, SSL version 3.0, and Private Communication Technology (PCT) version 1.0 are provided through the Schannel Cryptographic Provider. The Schannel Cryptographic Provider is accessed through Winsock. Security Support Provider Interface (SSPI) functions are available through the Secur32.dll module.

For more information about CAPI and SSPI features available for Windows CE, see Security Support Provider Interface and Cryptography.

CHAPTER 2

Serial Communications

Some Windows CE–based devices can communicate with other computers, printers, modems, or Global Positioning System (GPS) satellites by way of a serial connection.

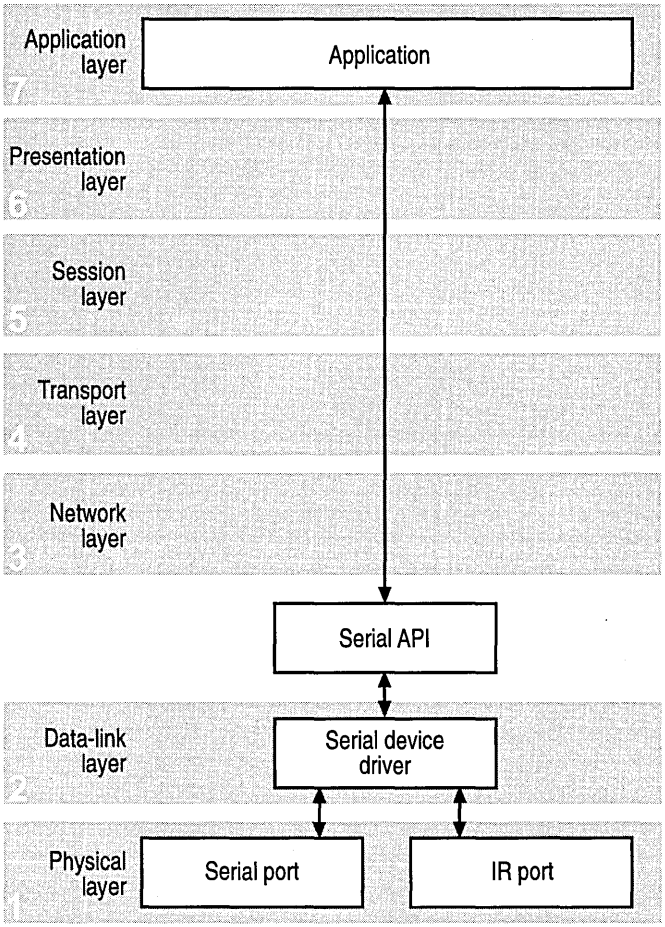
Serial I/O is the simplest form of communication supported by Windows CE. It is used when a direct, one-to-one connection exists between two devices. Serial I/O can occur by way of various hardware connections; however, most Windows CE–based devices use *serial cables* or a PC Card device such as a modem or *infrared* (IR) transceiver. Exchanging data by way of a serial cable is similar to reading from or writing to a file.

Windows CE supports standard Windows-based desktop functions for serial communication. These functions can be used to open, close, and manipulate *serial ports*, transmit and receive data, and manage the connection. Windows CE–based devices use *point-to-point protocol* (PPP) and *Serial Line Internet Protocol* (SLIP) for direct serial and dial-up connections.

Serial Protocols and the OSI Model

In the International Organization for Standardization Open Systems Interconnection (ISO/OSI) model for network communications, serial communications operates between the physical layer and the application layer. The RS–232–C standard describes the physical layer. Serial device drivers are stored in the next layer, the data-link layer. The Windows CE serial communications functions enable applications to exchange data by way of serial hardware.

The following illustration shows serial communications within the context of the ISO/OSI model.



Serial Communications Functions

Functions and structures used for serial communications are defined in the Winbase.h header file. Reading to and writing from a serial communications port on a Windows CE–based device is done by calling file input and output functions. The following table describes functions and structures defined in the Winbase.h header file.

Function	Description
CreateFile	Opens a serial port.
GetCommState	Fills in a device-control block—**DCB** structure—with the current control settings for a specified communications device.
SetCommState	Configures a communications device according to the specifications in a **DCB** structure. The function reinitializes all hardware and control settings, but does not empty I/O queues.
GetCommTimeouts	Retrieves the time-out parameters for all read/write operations on a specified communications device.
SetCommTimeouts	Sets the time-out parameters for all read/write operations on a specified communications device.
WriteFile	Writes data to a serial port, which transfers data to the device at the other end of a serial connection.
ReadFile	Reads data from a serial port, which receives data from a device at the other end of a serial connection.
SetCommMask	Specifies a set of events to monitor for a communications device.
GetCommMask	Retrieves the value of the event mask for a specified communications device.
WaitCommEvent	Waits for an event to occur for a specified communications device. The set of events monitored by **WaitCommEvent** is contained in the event mask associated with the device handle.
EscapeCommFunction	Directs a specified communications device to perform an extended function. Often used to set a serial port to IR mode.
ClearCommBreak	Restores character transmission for a specified communications device and places the transmission line in a non-break state.
ClearCommError	Retrieves communications error data and reports the current status of a specified communications device.

Serial Cables and Connectors

Windows CE–based devices use the RS-232-C standard to exchange data with serial devices and other computers by way of a serial connection.

When a serial connection is established between two devices, an application designates one device as the Data Communications Equipment (DCE) and the other device as the Data Terminal Equipment (DTE). The following table shows pin locations and describes common pin functions used on 9-pin and 25-pin connectors.

9-pin	25-pin	Purpose	Description
3	2	TD Transmit data	Sends data to another device
2	3	RD Receive data	Receives data from another device
7	4	RTS Request to send	Indicates the device is ready to send data
8	5	CTS Clear to send	Indicates the device is ready to accept data
6	6	DSR Data set ready	Indicates the receiving device is connected and ready to accept data
5	7	GND Signal ground	Verifies both devices are using the same voltage for transmitting data
4	20	DTR Data terminal ready	Indicates that the DTE device is ready

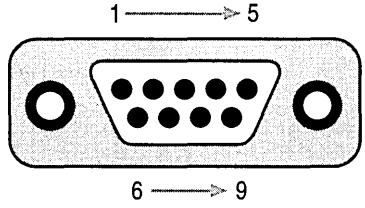
Windows CE–based devices use cables with a mini-connector, 9-pin connector, or 25-pin connector. Each connector is discussed in the following sections.

Mini-Connectors

Mini-connectors provide Windows CE–based devices with a compact serial port for 9-pin connectors. This port is a different shape than the 9-pin connector found on desktop computers; however, the pins have the same features as a standard 9-pin connector. The pin arrangement and shape can vary by manufacturer. The following illustration shows a mini-connector.

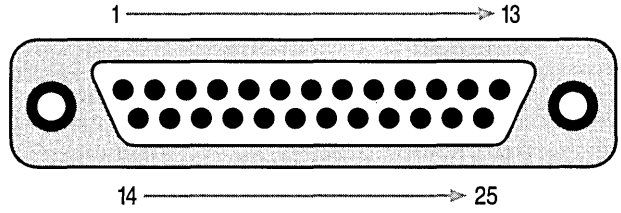
9-Pin Connectors

The standard 9-pin connector is commonly found on desktop computers for connecting devices such as keyboards. The following illustration shows a 9-pin connector. The top row, left-to-right pin arrangement is from 1 through 5. The bottom row, left-to-right pin arrangement is from 6 through 9.



25-Pin Connectors

Devices that require all pins defined by the RS-232-C standard, such as modems, require a 25-pin connector. The following illustration shows a 25-pin connector. The top row, left-to-right pin arrangement is from 1 through 13. The bottom row, left-to-right pin arrangement is from 14 through 25.



Programming Serial Connections

The following sections discuss steps carried out by an application to transfer data between devices, using a serial connection. The sections are discussed in the order of the programming task: opening a port, configuring the port, writing and reading data, and closing the port.

Opening a Port

Call the **CreateFile** function to open a serial port. Because hardware vendors and device driver developers can give any name to a port, an application should list the available ports and enable users to specify the port to open. If a port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND, and users should be notified the port is not available.

▶ **To open a serial port**

1. Insert a colon after the communications port pointed to with the first parameter; *lpzPortName*. For example, specify "COM1:" as the communications port.
2. Specify zero in the *dwShareMode* parameter. Communications ports cannot be shared in the same manner that files are shared.
3. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter. This flag is required.
4. Specify zero in the *dwFlagsAndAttributes* parameter. Windows CE supports only nonoverlapped I/O.

The following code example shows how to open a serial communications port.

```
// Open the serial port.
hPort = CreateFile (lpszPortName, // Pointer to the name of the port
                    GENERIC_READ | GENERIC_WRITE,
                                  // Access (read-write) mode
                    0,            // Share mode
                    NULL,         // Pointer to the security attribute
                    OPEN_EXISTING,// How to open the serial port
                    0,            // Port attributes
                    NULL);        // Handle to port with attribute
                                  // to copy
```

Before writing to or reading from a port, configure the port. When an application opens a port, it uses the default configuration settings, which may not be suitable for the device at the other end of the connection. The next section discusses serial port configuration settings.

Configuring a Serial Port

The most critical phase in serial communications programming is configuring the port settings with the **DCB** structure. Erroneously initializing the **DCB** structure is a common problem. When a serial communications function does not produce the expected results, the **DCB** structure may be in err.

A call to the **CreateFile** function opens a serial port with default port settings. Usually, the application needs to change the defaults. Use the **GetCommState** function to retrieve the default settings and use the **SetCommState** function to set new port settings.

Also, port configuration involves using the **COMMTIMEOUTS** structure to set the time-out values for read/write operations. When a time-out occurs, the **ReadFile** or **WriteFile** function returns the specific number of characters successfully transferred.

▶ To configure a serial port

1. Initialize the *DCBlength* member of the **DCB** structure to the size of the structure. This initialization is required before passing this member as a variable to any function.
2. Call the **GetCommState** function to retrieve the default settings for the port opened with the **CreateFile** function. To identify the port, specify in the *hPort* parameter the handle that **CreateFile** returns.
3. Modify **DCB** members as required. The following table shows the **DCB** structure members most frequently modified.

Member	Use
DCBlength	Before calling the **GetCommState** function, set this member to the length of the **DCB** structure. Neglecting to do this can cause a failure or return erroneous data.
BaudRate	Specifies the device communication rate. Assigns an actual baud rate or an index by specifying a CBR_ constant.
ByteSize	Specifies the bits per byte transmitted and received.
Parity	Specifies the parity scheme. Do not confuse this member with the *fParity* member, which turns parity on or off. Because parity is rarely used, this member can usually be NOPARITY.
StopBits	Specifies the number of stop bits. ONESTOPBIT is the most common setting.
fOutX and *fInX*	Turns software flow control on and off.
fRtsControl	Turns the RTS flow control on and off. RTS_CONTROL_ENABLE turns on the RTS line during the connection. RTS_CONTROL_HANDSHAKE turns on RTS handshaking. RTS_CONTROL_DISABLE turns off the RTS line.
fOutxCtsFlow	Turn the CTS flow control on and off. To use RTS/CTS flow control, specify TRUE for this member and RTS_CONTROL_HANDSHAKE for the *fRtsControl* member.
fOutxDsrFlow	Turns the DSR flow control on and off. DSR flow control is rarely used. A typical port configuration is to set this member to FALSE, while enabling the DTR line.
fDtrControl	Specifies the DTR flow control. DTR_CONTROL_ENABLE turns on the DTR line during the connection. DTR_CONTROL_HANDSHAKE turns on DTR handshaking. DTR_CONTROL_DISABLE turns off the DTR line.

4. Call the **SetCommState** function to set the new port settings.

The following code example shows how to use the **GetCommState** and **SetCommState** functions to configure a serial port.

```
DCB PortDCB;

// Initialize the DCBlength member.
PortDCB.DCBlength = sizeof (DCB);

// Get the default port setting information.
GetCommState (hPort, &PortDCB);

// Change the DCB structure settings.
PortDCB.BaudRate = 9600;                // Current baud
PortDCB.fBinary = TRUE;                 // Binary mode; no EOF check
PortDCB.fParity = TRUE;                 // Enable parity checking
PortDCB.fOutxCtsFlow = FALSE;           // No CTS output flow control
PortDCB.fOutxDsrFlow = FALSE;           // No DSR output flow control
PortDCB.fDtrControl = DTR_CONTROL_ENABLE;
                                        // DTR flow control type
PortDCB.fDsrSensitivity = FALSE;        // DSR sensitivity
PortDCB.fTXContinueOnXoff = TRUE;       // XOFF continues Tx
PortDCB.fOutX = FALSE;                  // No XON/XOFF out flow control
PortDCB.fInX = FALSE;                   // No XON/XOFF in flow control
PortDCB.fErrorChar = FALSE;             // Disable error replacement
PortDCB.fNull = FALSE;                  // Disable null stripping
PortDCB.fRtsControl = RTS_CONTROL_ENABLE;
                                        // RTS flow control
PortDCB.fAbortOnError = FALSE;          // Do not abort reads/writes on
                                        // error
PortDCB.ByteSize = 8;                   // Number of bits/bytes, 4-8
PortDCB.Parity = NOPARITY;              // 0-4=no,odd,even,mark,space
PortDCB.StopBits = ONESTOPBIT;          // 0,1,2 = 1, 1.5, 2

// Configure the port according to the specifications of the DCB
// structure.
if (!SetCommState (hPort, &PortDCB))
{
  // Could not create the read thread.
  MessageBox (hMainWnd, TEXT("Unable to configure the serial port"),
              TEXT("Error"), MB_OK);
  dwError = GetLastError ();
  return FALSE;
}
```

Configuring Time-Outs

An application must always set communications time-outs using the **COMMTIMEOUTS** structure each time it opens a communications port. If this structure is not configured, the port uses default time-outs supplied by the driver, or time-outs from a previous communications application. By assuming specific time-out settings when the settings are actually different, an application can have read/write operations that never complete or complete too often.

When read/write operations time out, the operations complete with no error values returned to the **ReadFile** and **WriteFile** functions. To determine if an operation has timed out, verify that the number of bytes actually transferred is fewer than the number of bytes requested. For example, if the **ReadFile** function returns TRUE, but fewer bytes were read than requested, the operation has timed out.

▶ **To configure time-outs for a serial port**

1. Initialize the **COMMITEMEOUTS** structure either by calling the **GetCommTimeouts** function or by setting the members manually.
2. Specify the maximum number of milliseconds that can elapse between two characters without a time-out occurring with the *ReadIntervalTimeout* member.
3. Specify the read time-out multiplier with the *ReadTotalTimeoutMultiplier* member. For each read operation, this number is multiplied by the number of bytes that the read operation expects to receive.
4. Specify the read time-out constant with the *ReadTotalTimeoutConstant* member. This member is the number of milliseconds added to the result of multiplying the total number of bytes to read by *ReadTotalTimeoutMultiplier.* The result is the number of milliseconds that must elapse before a time-out for the read operation occurs.
5. Specify the write time-out multiplier with the *WriteTotalTimeoutMultiplier* member. For each write operation, this number is multiplied by the number of bytes that the write operation expects to receive.
6. Specify the write time-out constant with the *WriteTotalTimeoutConstant* member. This member is the number of milliseconds added to the result of multiplying the total number of bytes to write by *WriteTotalTimeoutMultiplier.* The result is the number of milliseconds that must elapse before a time-out for the write operation occurs.
7. Call the **SetCommTimeouts** function to activate port time-out settings.

To assist with multitasking, it is common to configure **COMMTIMEOUT** so that **ReadFile** immediately returns with the characters to be read. To do this, set *ReadIntervalTimeout* to MAXWORD and set both *ReadTotalTimeoutMultiplier* and *ReadTotalTimeoutMultiplier* to zero.

The following code example shows how to configure time-outs for a serial port.

```
// Retrieve the time-out parameters for all read and write operations
// on the port.
COMMTIMEOUTS CommTimeouts;
GetCommTimeouts (hPort, &CommTimeouts);

// Change the COMMTIMEOUTS structure settings.
CommTimeouts.ReadIntervalTimeout = MAXDWORD;
CommTimeouts.ReadTotalTimeoutMultiplier = 0;
CommTimeouts.ReadTotalTimeoutConstant = 0;
CommTimeouts.WriteTotalTimeoutMultiplier = 10;
CommTimeouts.WriteTotalTimeoutConstant = 1000;

// Set the time-out parameters for all read and write operations
// on the port.
if (!SetCommTimeouts (hPort, &CommTimeouts))
{
  // Could not create the read thread.
  MessageBox (hMainWnd, TEXT("Unable to set the time-out parameters"),
              TEXT("Error"), MB_OK);
  dwError = GetLastError ();
  return FALSE;
}
```

Writing to a Serial Port

The **WriteFile** function transfers data through the serial connection to another device. Before calling this function, an application must open and configure a serial port.

Because Windows CE does not support overlapped I/O—also called asynchronous I/O—the primary thread or any thread that creates a window should not try to write a large amount of data to a serial port. Such threads are blocked and cannot manage message queues. An application can simulate overlapped I/O by creating multiple threads to handle read/write operations. To coordinate threads, an application calls the **WaitCommEvent** function to block threads until specific communications events occur. For more information about communications events, see Using Communications Events.

▶ **To write to a serial port**

1. Pass the port handle to the **WriteFile** function in the *hFile* parameter. The **CreateFile** function returns this handle when an application opens a port.
2. Specify a pointer to the data to be written in *lpBuffer*. Often this data is binary data or a character array.

3. Specify the number of characters to write in *nNumberOfBytesToWrite*. For Windows CE–based devices, usually one character is written because an application must convert Unicode characters to ASCII characters to enable text transfer to a device at the opposite end of a serial connection.
4. Specify in *lpNumberOfBytesWritten* a pointer to the number of bytes actually written. **WriteFile** fills this variable so that an application can determine if the data transferred.
5. Be sure that *lpOverlapped* is NULL.

The following code example shows how to transfer data using the **WriteFile** function.

```
DWORD dwError,
      dwNumBytesWritten;

WriteFile (hPort,              // Port handle
           &Byte,              // Pointer to the data to write
           1,                  // Number of bytes to write
           &dwNumBytesWritten, // Pointer to the number of bytes
                               // written
           NULL                // Must be NULL for Windows CE
);
```

Reading from a Serial Port

An application calls the **ReadFile** function to receive data from a device at the other end of a serial connection. **ReadFile** takes the same parameters as the **WriteFile** function.

Typically, a read operation is a separate thread that is always ready to process data arriving at a serial port. A communications event signals the read thread that there is data to read at a serial port. The thread usually reads one byte at a time—one **ReadFile** call for each byte—until all of the data is read. Then the read thread waits for another communications event.

For more information about communications events, see Using Communications Events.

▶ **To read from a serial port**

1. Pass the port handle to **ReadFile** in the *hFile* parameter. The **CreateFile** function returns this handle when an application opens a port.
2. Specify a pointer to receive the data that is read in *lpBuffer*.
3. Specify the number of characters to read in *nNumberOfBytesToRead*.

4. Specify a pointer to the number of bytes actually read in *lpNumberOfBytesRead.*
5. Be sure that *lpOverlapped* is NULL. Windows CE does not support overlapped I/O.

The following code example shows how to receive data using the **ReadFile** function.

```
BYTE Byte;
DWORD dwBytesTransferred;

ReadFile (hPort,                // Port handle
          &Byte,                // Pointer to data to read
          1,                    // Number of bytes to read
          &dwBytesTransferred,  // Pointer to number of bytes
                                // read
          NULL                  // Must be NULL for Windows CE
);
```

Using Communications Events

A *communications event* is a notification sent by Windows CE to an application when a significant incident occurs. Using the **WaitCommEvent** function, an application can block a thread until a specific event occurs. A call to the **SetCommMask** function specifies which event or events must occur before processing can continue. When more than one event is specified, any single specified event that occurs causes **WaitCommEvent** to return.

For example, this mechanism enables an application to find out when data arrives at the serial port. By waiting for a communications event that indicates data is present, an application avoids forestalling the serial port with a call to the **ReadFile** function that waits for data to arrive. **ReadFile** is called only when there is data to read.

The following table lists the communications events that an application can use with the **WaitCommEvent** function.

Event	Description
EV_BREAK	A break occurred on input.
EV_CTS	The CTS signal changed state.
EV_DSR	The DSR signal changed state.
EV_ERR	A line-status error occurred. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
EV_RING	A ring indicator was detected.
EV_RLSD	The receive-line-signal-detect signal changed state.
EV_RXCHAR	A character was received and placed in the input buffer.
EV_RXFLAG	The event character was received and placed in the input buffer.
EV_TXEMPTY	The last character in the output buffer was sent.

▶ To use communications events

1. Specify events to look for with a call to the **SetCommMask** function.
2. Call the **WaitCommEvent** function and specify which events cause this function to return. When an application specifies more than one event, the *lpEvtMask* parameter points to a variable that identifies the event that caused **WaitCommEvent** to return.
3. After **WaitCommEvent** returns, use a loop that calls **ReadFile** until all received data has been read.
4. Call **SetCommMask** again to specify which events to look for.

SetCommMask is the first call in a loop that applications generally use to monitor a serial port and read data. The following code example shows how to use communications events for this purpose.

```
BYTE Byte;
DWORD dwBytesTransferred;

// Specify a set of events to be monitored for the port.
SetCommMask (hPort, EV_RXCHAR | EV_CTS | EV_DSR | EV_RLSD | EV_RING);

while (hPort != INVALID_HANDLE_VALUE)
{
  // Wait for an event to occur for the port.
  WaitCommEvent (hPort, &dwCommModemStatus, 0);
```

```
    // Respecify the set of events to be monitored for the port.
    SetCommMask (hPort, EV_RXCHAR | EV_CTS | EV_DSR | EV_RING);

    if (dwCommModemStatus & EV_RXCHAR)
    {
      // Loop while waiting for data.
      do
      {
        // Read the data from the serial port.
        ReadFile (hPort, &Byte, 1, &dwBytesTransferred, 0);

        // Display the data read.
        if (dwBytesTransferred == 1)
          ProcessChar (Byte);

      } while (dwBytesTransferred == 1);
    }
}
```

Closing a Serial Port

Call the **CloseHandle** function to close a serial port when an application is done using it. **CloseHandle** has one parameter, which is the handle returned by the **CreateFile** function call that opened the port.

Using Infrared Communications

Many Windows CE–based devices have an infrared port compliant with the *Infrared Data Association* (IrDA). The IrDA specifies standards for hardware specifications and software protocols.

Windows CE–based devices have three options for implementing IR communications: raw infrared (raw IR), IrCOMM, and Infrared Sockets (IrSock).

For more information about IrSock, see Windows Sockets.

Raw IR

Using raw IR involves accessing the IR port and handling the connection as a serial port with IR hardware attached. An application has the most control of IR communications using this non-IrDA-compliant method. Without this standard, it is possible for signal collisions to occur between devices during a data exchange. Also, it is possible to lose data when the infrared beam is broken, such as when someone walks between the two devices. The application must detect these error conditions and correct them.

Before using the IR port, an application must identify the COM port attached to the IR transceiver. The **HKEY_LOCAL_MACHINE\Comm\IrDA** key specifies the port. Use the **RegOpenKeyEx** function to open this key. Then use the **RegQueryValueEx** function to get the value of the **Port** subkey, which is the port number.

If a Windows CE–based device shares serial hardware with the IR port and the serial port, an application must instruct the COM driver to route data through the IR port. To do this, use the **EscapeCommFunction** function with the *dwFunc* parameter set to CLRIR. Now use the standard serial communications functions to transmit and receive data.

IrCOMM

An easy way to use the standard communications functions for transferring data over the IR port is to use the IrCOMM mode.

An IrCOMM port is a simulated port and not a real device, which causes some differences with a raw IR port. For instance, an application cannot configure an IrCOMM port. Windows CE transparently uses IrSock to configure an IrCOMM port to meet the IrDA standard. When the **GetCommState** function is called to retrieve the communications settings for an IrCOMM port, the **DCB** structure returned contains zeros. But, with the IR link managed by IrSock, it tackles important issues such as signal collision, signal interruption, and remote device detection. Relieving an application of these tasks greatly simplifies programming IR communications.

IrCOMM only supports two devices connected simultaneously.

To determine the IrCOMM port, open the **HKEY_LOCAL_MACHINE\Drivers\Builtin\IrCOMM** key and query the **Index** subkey.

Serial Communications Sample Application

The following code example opens, configures, and closes the serial communications port and performs read/write operations for a TTY terminal emulation application. The complete TTY application is contained on the compact disc that accompanies this book.

```
/*****************************************************************************
Module Name:
  port.c

*****************************************************************************/

#include <windows.h>
#include "tty.h"

/*****************************************************************************

  PortInitialize (LPTSTR lpszPortName)

*****************************************************************************/
BOOL PortInitialize (LPTSTR lpszPortName)
{
  DWORD dwError,
        dwThreadID;
  DCB PortDCB;
  COMMTIMEOUTS CommTimeouts;

  // Open the serial port.
  hPort = CreateFile (lpszPortName, // Pointer to the name of the port
                      GENERIC_READ | GENERIC_WRITE,
                                    // Access (read/write) mode
                      0,            // Share mode
                      NULL,         // Pointer to the security attribute
                      OPEN_EXISTING,// How to open the serial port
                      0,            // Port attributes
                      NULL);        // Handle to port with attribute
                                    // to copy

  // If it fails to open the port, return FALSE.
  if ( hPort == INVALID_HANDLE_VALUE )
```

```
  // Could not open the port.
  MessageBox (hMainWnd, TEXT("Unable to open the port"),
              TEXT("Error"), MB_OK);
  dwError = GetLastError ();
  return FALSE;
}

PortDCB.DCBlength = sizeof (DCB);

// Get the default port setting information.
GetCommState (hPort, &PortDCB);

// Change the DCB structure settings.
PortDCB.BaudRate = 9600;                // Current baud
PortDCB.fBinary = TRUE;                 // Binary mode; no EOF check
PortDCB.fParity = TRUE;                 // Enable parity checking.
PortDCB.fOutxCtsFlow = FALSE;           // No CTS output flow control
PortDCB.fOutxDsrFlow = FALSE;           // No DSR output flow control
PortDCB.fDtrControl = DTR_CONTROL_ENABLE;
                                        // DTR flow control type
PortDCB.fDsrSensitivity = FALSE;        // DSR sensitivity
PortDCB.fTXContinueOnXoff = TRUE;       // XOFF continues Tx
PortDCB.fOutX = FALSE;                  // No XON/XOFF out flow control
PortDCB.fInX = FALSE;                   // No XON/XOFF in flow control
PortDCB.fErrorChar = FALSE;             // Disable error replacement.
PortDCB.fNull = FALSE;                  // Disable null stripping.
PortDCB.fRtsControl = RTS_CONTROL_ENABLE;
                                        // RTS flow control
PortDCB.fAbortOnError = FALSE;          // Do not abort reads/writes on
                                        // error.
PortDCB.ByteSize = 8;                   // Number of bits/byte, 4-8
PortDCB.Parity = NOPARITY;              // 0-4=no,odd,even,mark,space
PortDCB.StopBits = ONESTOPBIT;          // 0,1,2 = 1, 1.5, 2

// Configure the port according to the specifications of the DCB
// structure.
if (!SetCommState (hPort, &PortDCB))
{
  // Could not create the read thread.
  MessageBox (hMainWnd, TEXT("Unable to configure the serial port"),
              TEXT("Error"), MB_OK);
  dwError = GetLastError ();
  return FALSE;
}

// Retrieve the time-out parameters for all read and write operations
// on the port.
GetCommTimeouts (hPort, &CommTimeouts);
```

```
// Change the COMMTIMEOUTS structure settings.
  CommTimeouts.ReadIntervalTimeout = MAXDWORD;
  CommTimeouts.ReadTotalTimeoutMultiplier = 0;
  CommTimeouts.ReadTotalTimeoutConstant = 0;
  CommTimeouts.WriteTotalTimeoutMultiplier = 10;
  CommTimeouts.WriteTotalTimeoutConstant = 1000;

  // Set the time-out parameters for all read and write operations
  // on the port.
  if (!SetCommTimeouts (hPort, &CommTimeouts))
  {
    // Could not create the read thread.
    MessageBox (hMainWnd, TEXT("Unable to set the time-out parameters"),
                TEXT("Error"), MB_OK);
    dwError = GetLastError ();
    return FALSE;
  }

  // Direct the port to perform extended functions SETDTR and SETRTS
  // SETDTR: Sends the DTR (data-terminal-ready) signal.
  // SETRTS: Sends the RTS (request-to-send) signal.
  EscapeCommFunction (hPort, SETDTR);
  EscapeCommFunction (hPort, SETRTS);

  // Create a read thread for reading data from the communication port.
  if (hReadThread = CreateThread (NULL, 0, PortReadThread, 0, 0,
                                  &dwThreadID))
  {
    CloseHandle (hReadThread);
  }
  else
  {
    // Could not create the read thread.
    MessageBox (hMainWnd, TEXT("Unable to create the read thread"),
                TEXT("Error"), MB_OK);
    dwError = GetLastError ();
    return FALSE;
  }

  return TRUE;
}


/*****************************************************************************

  PortWrite (BYTE Byte)

*****************************************************************************/
```

```
void PortWrite (BYTE Byte)
{
  DWORD dwError,
        dwNumBytesWritten;

  if (!WriteFile (hPort,              // Port handle
                  &Byte,              // Pointer to the data to write
                  1,                  // Number of bytes to write
                  &dwNumBytesWritten, // Pointer to the number of bytes
                                      // written
                  NULL))              // Must be NULL for Windows CE
  {
    // WriteFile failed. Report error.
    dwError = GetLastError ();
  }
}



/*****************************************************************************

  PortReadThread (LPVOID lpvoid)

*****************************************************************************/
DWORD PortReadThread (LPVOID lpvoid)
{
  BYTE Byte;
  DWORD dwCommModemStatus,
        dwBytesTransferred;

  // Specify a set of events to be monitored for the port.
  SetCommMask (hPort, EV_RXCHAR | EV_CTS | EV_DSR | EV_RLSD | EV_RING);

  while (hPort != INVALID_HANDLE_VALUE)
  {
    // Wait for an event to occur for the port.
    WaitCommEvent (hPort, &dwCommModemStatus, 0);

    // Respecify the set of events to be monitored for the port.
    SetCommMask (hPort, EV_RXCHAR | EV_CTS | EV_DSR | EV_RING);

    if (dwCommModemStatus & EV_RXCHAR)
    {
      // Loop while waiting for data.
      do
      {
        // Read the data from the serial port.
        ReadFile (hPort, &Byte, 1, &dwBytesTransferred, 0);
```

```
        // Display the data read.
        if (dwBytesTransferred == 1)
          ProcessChar (Byte);

      } while (dwBytesTransferred == 1);
    }

    // Retrieve modem control-register values.
    GetCommModemStatus (hPort, &dwCommModemStatus);

    // Set the indicator lights.
    SetLightIndicators (dwCommModemStatus);
  }

  return 0;
}


/*****************************************************************************

  PortClose (HANDLE hCommPort)

*****************************************************************************/
BOOL PortClose (HANDLE hCommPort)
{
  DWORD dwError;

  if (hCommPort != INVALID_HANDLE_VALUE)
  {
    // Close the communication port.
    if (!CloseHandle (hCommPort))
    {
      dwError = GetLastError ();
      return FALSE;
    }
    else
    {
      hCommPort = INVALID_HANDLE_VALUE;
      return TRUE;
    }
  }

  return FALSE;
}
```

CHAPTER 3

Telephony API

To simplify the process of using a modem, Windows CE provides an implementation of the Microsoft Telephony API (TAPI).

Using TAPI functions included with Windows CE, you can create an application that enables a user to make a phone call by choosing a phone number or choosing an image. A user can set up a conference call, or connect using a modem, to a remote host computer to download data at predetermined times.

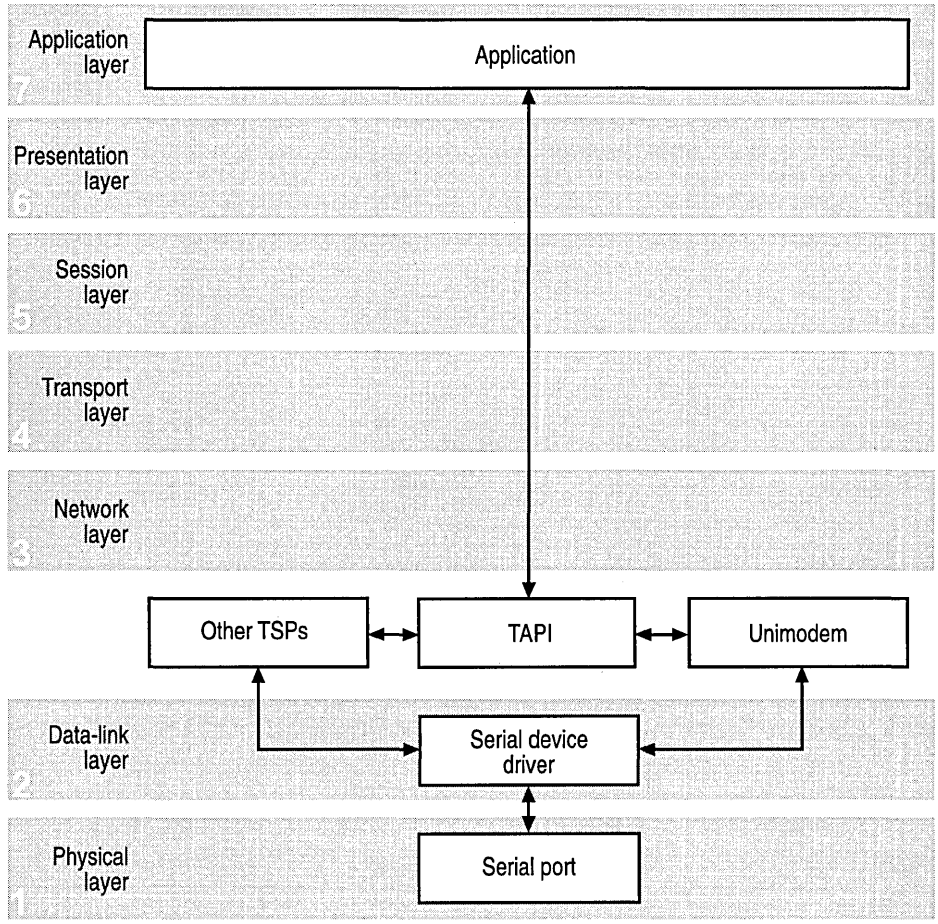
TAPI supports outbound calls and address translation services. Windows CE does not support inbound calls. TAPI also supports installable service providers.

The following sections discuss TAPI for Windows CE, the telephony programming model, and TAPI code examples. Also discussed is use of the TAPI in Windows CE–based applications used with a modem.

TAPI and the OSI Model

In the International Organization for Standardization Open Systems Interconnection (ISO/OSI) model for network communications, TAPI operates at the data-link layer. TAPI functions provide basic support for outbound dialing and controlling a modem.

The following illustration shows TAPI within the context of the ISO/OSI model.



TAPI is an open industry standard and is independent of switches, so applications can run on a variety of computers, telephony hardware, and support network services. TAPI is part of the Windows Open System Architecture that creates a hardware-independent work environment. Telephony Service Provider Interface (TSPI) enables developers to create telephony service applications that handle function calls from remote applications to carry out and control communications over the telephone network. In Windows CE, TAPI links to and calls TSPI functions using standard dynamic-link library (DLL) functions.

Telephony Service Provider Interface

Windows CE supports TSPI, a DLL that provides communication services over a telephone network through a set of exported functions called by TAPI. The Tspi.h header file should only be used in conjunction with the Tapi.h header file. Most parameters are passed through from corresponding procedures in Tapi.h.

The TSPI is responsible for managing the data from TAPI to control line and phone devices.

TAPI supports the **lineAddProvider** function, which installs a new TSPI into the telephony system. When an application calls this function, TAPI verifies that it can access the service provider. If the function fails, or the DLL or the service provider cannot be found, the provider is not added to the telephony system. If the function succeeds, and the system is active, TAPI starts the new service provider DLL.

When a Windows CE–based application calls a TAPI function, the TAPI DLL validates and arranges function parameters and forwards them to the appropriate service provider. A service provider can provide different levels of the service provider interface: basic, supplementary, or extended. For example, a simple service provider might provide basic telephony service, such as support for outbound calls, through a Hayes-compatible modem. A custom service provider, written by a third-party vendor, might provide a full range of outbound call management.

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

Applications use the TAPI functions to determine which services are available on the device. TAPI identifies available service providers and the services supported, and provides this data to applications. In this way, any number of applications can request services from the same service provider; TAPI manages all access to the service provider.

For more information about TSPI and service providers, see the *Microsoft TSPI Programmer's Reference* available on the Microsoft Developer Network (MSDN).

TAPI Functions

The Tapi.dll module exports the TAPI functions used to develop TAPI client applications, such as outbound dialing telephony applications. TAPI functions and structures are defined in the Tapi.h header file.

TAPI functions are identified as asynchronous if they can return before making a call to the application callback function; otherwise, they are considered synchronous.

The following table shows the TAPI functions supported by Windows CE.

Function	Description
lineAddProvider	Installs a new service provider
lineClose	Closes a specified line device
lineConfigDialogEdit	Displays a dialog box for a user to change configuration data for a line device
lineDeallocateCall	Frees system-allocated memory related to the call after the call has been dropped
lineDrop	Drops or disconnects a call
lineGetDevCaps	Returns the capabilities of a specified line device
lineGetDevConfig	Returns the default configuration of a specified line device
lineGetID	Retrieves a device identifier associated with the specified open line, address, or call
lineGetTranslateCaps	Returns the address of translation capabilities
lineInitialize	Initializes the TAPI line for use by invoking applications
lineMakeCall	Initiates outbound dialing, makes a call, and returns a handle
lineNegotiateAPIVersion	Negotiates the API version
lineOpen	Opens a specified line device for providing subsequent monitoring and control of the line
lineSetCurrentLocation	Sets the location used as the context for address translation
lineSetDevConfig	Sets the configuration of the specified medial stream device

Function	Description
lineSetStatusMessages	Specifies the status change on the line device or its addresses for which the application is notified
lineShutdown	Shuts down TAPI
lineTranslateAddress	Translates a specified address into a string format capable of being dialed
lineTranslateDialog	Displays a dialog box enabling a user to change the current location of a phone number to be dialed

Note All TAPI functions return zero to signal success.

Callback Function

To receive status notices, the application must implement a TAPI **lineCallbackFunc** function to establish a way to communicate with TAPI. The callback function is used for notifying applications of changes in the status of calls, lines, and phone devices. TAPI uses the callback function to send messages to an application.

For example, the LINE_REPLY message, sent to the Windows CE–based application, carries the request identifier and an error indicator. Only the application that issued the request receives the reply message, but when the request causes changes in the device state or call, other related applications might also receive event-related messages through the callback function.

Modem Support

Windows CE includes a telephony service provider (TSP) for an AT command-based modem. A Unimodem driver is included with Windows CE. Other TSPs can be written by an independent software vendor.

It is important to remember that a driver is a separate DLL and not part of TAPI. The modem driver translates TAPI function calls into AT commands to configure and dial modems. TAPI manages the application's use of devices, ports, and call traffic.

If you decide to write an application that enables a user to select and dial a phone number through a dialog box, a modem must be attached or built into the device. A user also needs a telephone set connected to the modem. If the Windows CE–based application dials for a user, a user needs to use the phone and talk through the microphone.

The Unimodem file included with Windows CE is called Unimodem.dll. This file is included in the device ROM.

Creating a Plug and Play Device Identifier

A modem needs to have a Plug and Play identifier to enable the OS to recognize the device so that it can load the appropriate software. Each device manufacturer is responsible for assigning the Plug and Play identifier for each product and storing it in the hardware. Refer to the relevant Plug and Play specification to determine how to include the identifier in the hardware.

For modem manufacturers, it is important to follow the Plug and Play specifications and implement a unique device identifier to ensure that it works with Windows CE. A Plug and Play identifer usually consists of:

- A three-character vendor identifier.
- A four-digit product identifier; for example, XYZ1234.

For a modem, the Plug and Play identifier syntax is CompanyName-ModelName-Checksum.

Modem Registry Keys

The following section describes data stored in specific modem registry keys that help advanced users correct problems with the commands Windows CE uses to control a modem.

Modem registry keys are stored under the following key.

HKEY_LOCAL_MACHINE\Drivers\PCMCIA*CompanyX-GenericModem-1234*

Each installed modem uses one registry key; additional subkeys, which contain AT commands that Windows CE uses to initialize and dial the modem, plus other entries that communications and modem drivers use.

Some of the more important entries you can use to correct or optimize modem operation are described in the following sections. The full set of modem registry keys are documented in the Windows CE Device Driver Development Kit.

GenericModem Key

The following table shows the registry settings for **HKEY_LOCAL_MACHINE\Drivers\PCMCIA*CompanyX-GenericModem-1234***.

Registry key	Description	Example
Tsp	Provides the TSP name that services this port	Usually Unimodem.dll.
DeviceArrayIndex	Driver-specific	Always 1 for version 1 PC Card devices.
Prefix	Device prefix	Normally "COM" for PC Card devices.
Dll	Serial.dll	n/a
FriendlyName	Name displayed to a user by the TAPI applications	n/a
DeviceType	Device type	Always 3 for version 1 PC Card devices.
ResetDelay	Optional value	Specifies a number of milliseconds of delay used during the PC Card reset sequence. Most modems do not require this value; some modems need more than the PC–Card specified reset delay. For such modems, specify an appropriate value here.
DevConfig	Unimodem-specific data	This is a binary structure indicating the serial device capabilities: baud, parity, and so on. Not published. Use the default case.

Init Key

The **\Init** sequence in **HKEY_LOCAL_MACHINE\Drivers\PCMCIA** ***CompanyX-GenericModem-1234*****\Init** is an enumerated sequence of strings used to initialize the modem. There can be any number of **Init** strings. Unimodem steps through them in sequence, sending the **Init** string to the modem and waiting for an "OK" response before continuing to the next command in the sequence. The name of each entry is its sequence number, starting with the number 1, and its data is the command that is sent to the modem. Usually, the **Init** key entry 1 is AT<cr>, which is sent to the modem to start it. **Init** key entry 2 usually contains &F or a similar command to restore the modem to its default settings. Subsequent **Init** key entries contain miscellaneous commands to configure the modem to maintain compatibility with Windows CE.

Settings Key

The **Settings** key contains commands for configuring various modem settings. The following table shows the registry settings for **HKEY_LOCAL_MACHINE\Drivers\PCMCIA*****CompanyX-GenericModem-1234*****\settings**.

Registry key	Description	Example
MaxCmd	Maximum command length.	40
Prefix	Modem command prefix.	AT
Terminator	Configuration command suffix; added to the end of any command sequences sent to the modem by Unimodem.	<cr>
DialPrefix	Prefix for any dial commands.	D
DialSuffix	Extend a dialing string across multiple commands. Unimodem breaks long dial commands into approximately 40 char strings because many modems cannot handle longer commands. This suffix char is used to indicate to the modem that the dial sequence is continued in the next command.	<;>

Registry key	Description	Example
Pulse	Dial prefix used for pulse dialing, such as ATDP.	P
Tone	Dial prefix used for tone dialing, such as ATDT.	T
Blind_Off	Detect dial tone before dialing.	X4
Blind_On	Detect dial tone before dialing.	X3, See Blind_Off.
CallSetupFailTimeout	Call time-out register.	S7=<#>
Reset	Reset modem.	ATZ<cr>
FlowHard	Enable hardware flow control.	AT\Q3<cr>
FlowSoft	Enable software flow control.	AT\Q1<cr>
FlowOff	Disable all flow control.	AT\Q<cr>

Example of Registry Key Settings

The following is an example that uses a fictitious modem with the ***CompanyX-GenericModem-1234*** identifier. For an actual modem, this string would have to be replaced with the Plug and Play identifier syntax: CompanyName-ModelName-Checksum.

For efficiency, Unimodem supports a default for many of these values. If no specified modem value exists, Unimodem attempts to read that value from a default set of registry values.

The following code example is an example of registry key settings.

```
[HKEY_LOCAL_MACHINE\Drivers\PCMCIA\CompanyX-GenericModem-1234]
   "Tsp"="Unimodem.dll"
   "DeviceArrayIndex"=dword:1
   "Prefix"="COM"
   "Dll"="Serial.dll"
   "FriendlyName"="Motorola Montana 28.8"
   "DeviceType"=dword:3
   "ResetDelay"=dword:800
   "DevConfig"=hex: 10,00, 00,00, 78,00,00,00, 10,01,00,00, 00,4B,00,00,
00,00, 08, 00, 00, 00,00,00,00

[HKEY_LOCAL_MACHINE\Drivers\PCMCIA\CompanyX-GenericModem-1234\Init]
   "1"="AT<cr>"
   "2"="ATE0V1&C1&D2<cr>"
   "3"="ATS7=60M1<cr>"
```

```
[HKEY_LOCAL_MACHINE\Drivers\PCMCIA\CompanyX-GenericModem-1234\Settings]
    "MaxCmd"=dword:28
   "Prefix"="AT"
   "Terminator"="<cr>"
   "DialPrefix"="D"
   "DialSuffix"=";"
   "Pulse"="P"
   "Tone"="T"
   "Blind_Off"="X4"
   "Blind_On"="X3"
   "CallSetupFailTimeout"="S7=<#>"
   "Reset"="ATZ<cr>"
   "FlowHard"="AT\Q3<cr>"
   "FlowSoft"="AT\Q1<cr>"
   "FlowOff"="AT\Q<cr>"
```

Creating a TAPI Application

TAPI consists of functions that provide access to the telephone network. The application can consist of dialog boxes and have an interactive interface so that a user can easily dial a phone number.

Using a Modem

A Windows CE–based application that uses a modem must be able to handle tasks such as dialing a phone number, initializing the modem, opening the line, and disconnecting when the session is complete.

▶ **To make a modem connection using TAPI**

1. Call **lineInitialize** to initialize TAPI.
2. Call **lineOpen** to open the line.
3. Call **lineMakeCall**.
4. Call **lineDeallocateCall**.
5. Call **lineClose** to close the line connection.
6. Call **lineShutdown** to end the session.

The **lineInitialize** function returns the number of line devices available. The pointer to the application callback function must be provided so that TAPI can return data.

When the call is set up, TAPI returns a LINE_REPLY message through the callback function. This message indicates only that the call has been established at the local end, which is perhaps indicated by a dial tone. The parameters for the **lineMakeCall** function are the phone number to dial, the handle to a line device, and other parameters. LINE_CALLSTATE messages are sent to indicate the status of the call, such as the following states: dialing, proceeding, ring-back, and connected.

As the connection process proceeds, TAPI returns a series of LINE_CALLSTATE messages through the callback function to indicate the progress of the connection; for example, dial tone and ringing. When the connection is completed, TAPI returns a LINECALLSTATE_CONNECTED message.

During data transfer, TAPI continues to manage the connection, but the application handles data transmission and reception. When the transmission is complete, TAPI returns a LINE_CALLSTATE message, such as one indicating that a remote disconnect has occurred.

Telephony System

Telephony system capabilities help people get the most from telecommunications systems, enabling them to efficiently manage voice calls and control data-transfer operations. You can use TAPI to bring this efficiency to any Windows CE–based application, such as a database manager, spreadsheet, word processor, or personal information manager—in fact, any application that can benefit by sending and receiving data through the telephone network.

TAPI provides a set of tools for incorporating these features into your application:

- Connect directly to the telephone network rather than rely on a separate communications application
- Dial phone numbers automatically
- Transmit documents as files or electronic mail
- Access data from news retrieval and other information services

Line Devices

A line device is a physical device such as a modem, or an ISDN card connected to a telephone line. Line devices support telephone functions by supporting applications to send or receive data to or from a telephone network. A line device may consist of a set of one or more similar lines used to establish telephone network connections. In TAPI applications, a line device is the logical representation of a physical line device.

TAPI requires that every TAPI-capable line device support Basic Telephony. If an application needs to use capabilities beyond those of Basic Telephony—namely Supplementary Telephony or Extended Telephony—it must first determine the line device capabilities, which can vary according to network configuration (client versus client/server), hardware, service-provider software, and the telephone network. The **lineNegotiateAPIVersion** function enables the application to identify the line device capabilities. The **lineGetDevCaps** function returns the telephone device capabilities implemented through the use of the TAPI functions of a specific line device in a data structure of the type **LINEDEVCAPS**.

Media Stream

The media stream consists of the data exchanged during a call. TAPI only provides control for line and telephone devices and does not give access to media stream content. To manage the media stream, a Windows CE–based application can use other APIs, Wave Audio, or Media Control Interface (MCI).

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

Initializing TAPI

When initializing TAPI, the application must establish a way to communicate between a Windows CE–based application and TAPI. An application tells TAPI the address of the callback function in the call to the **lineInitialize** function.

The **lineInitialize** function initializes the application's use of Tapi.dll for subsequent use. It registers the application-specific notification mechanism and returns the number of line devices available to the application.

The following code example shows how to initialize Tapi.dll.

```
DWORD InitializeTAPI ()
{
  DWORD dwLineID,
        dwReturn,
        dwTimeCount = GetTickCount ();

  TCHAR szWarning[] = TEXT("Cannot initialize the application's use\n")
                      TEXT("of tapi.dll. Quit all other telephony\n")
                      TEXT("programs, and then try again.");
```

```
    // Initialize Tapi.dll. Keep trying until the user cancels or
    // the application stops returning LINEERR_REINIT.
    while ( (dwReturn = lineInitialize (&g_hLineApp,
                                        g_hInst,
                                        (LINECALLBACK) lineCallbackFunc,
                                        g_szAppName,
                                        &g_dwNumDevs)) == LINEERR_REINIT)
    {
      // Display the message box if five seconds have passed.
      if (GetTickCount () > 5000 + dwTimeCount)
      {
        if (MessageBox (g_hwndMain, szWarning, TEXT("Warning"),
                        MB_OKCANCEL) == IDOK)
          break;

        // Reset the time counter.
        dwTimeCount = GetTickCount ();
      }
    }

    // If the lineInitialize function fails, then return.
    if (dwReturn)
      return dwReturn;

    // If there is no device, then return.
    if (g_dwNumDevs == 0)
    {
      ErrorBox (TEXT("There are no line devices available."));
      return LINEERR_NODEVICE;
    }

    // Allocate a buffer for storing LINEINFO for all the available lines.
    if (! (g_lpLineInfo = (LPLINEINFO) LocalAlloc (
                                        LPTR,
                                        sizeof (LINEINFO) * g_dwNumDevs)))
    {
      return LINEERR_NOMEM;
    }

    // Get the line data such as line name, permanent identifier,
    // and number of available addresses on the line by calling
    // the GetLineInfo function.
    for (dwLineID = 0; dwLineID < g_dwNumDevs; ++dwLineID)
    {
      GetLineInfo (dwLineID, &g_lpLineInfo [dwLineID]);
    }

    return ERR_NONE;
}
```

The following are the variables used in the previous function, and in code examples that follow.

```
HINSTANCE g_hInst = NULL;        // hInstance of the application
HWND g_hwndMain = NULL;          // Handle to the main window
HWND g_hwndDial = NULL;          // Handle to the dialing window

TCHAR g_szTitle[] = TEXT("CeDialer");
                                 // CeDialer application window name
TCHAR g_szAppName[] = TEXT("CeDialer Application");
                                 // Main window class name
HLINEAPP g_hLineApp = NULL;      // Applications use handle for TAPI
                                 // (lineInitialize)
HCALL g_hCall = NULL;            // Handle to the open line device on
                                 // which the call is originated
                                 // (lineMakeCall)
LONG g_MakeCallRequestID = 0;    // Request identifier returned by
                                 // lineMakeCall.
LONG g_DropCallRequestID = 0;    // Request identifier returned by
                                 // lineDrop.
BOOL g_bCurrentLineAvail = TRUE;// Indicates line availability

TCHAR g_szCurrentNum[TAPIMAXDESTADDRESSSIZE + 1];
                                 // Current phone number
TCHAR g_szLastNum[TAPIMAXDESTADDRESSSIZE + 1];
                                 // Last called phone number
DWORD g_dwNumDevs = 0;           // Number of line devices available
DWORD g_dwCurrentLineID = -1;    // Current line device identifier
DWORD g_dwCurrentLineAddr = -1; // Current line address

LINEINFO g_CurrentLineInfo;      // Contains the current line data
LINEINFO *g_lpLineInfo = NULL;   // Array that contains data for all
                                 // lines

#define InfoBox(_s)  MessageBox (g_hwndMain, _s, TEXT("Info"), MB_OK)
#define ErrorBox(_s) MessageBox (g_hwndMain, _s, TEXT("Error"), MB_OK)

#define ERR_NONE              0
#define TAPI_VERSION_1_0      0x00010003
#define TAPI_VERSION_1_4      0x00010004
#define TAPI_VERSION_2_0      0x00020000
#define TAPI_CURRENT_VERSION  TAPI_VERSION_2_0
```

```
typedef struct tagLINEINFO
{
  HLINE hLine;              // Line handle returned by lineOpen
  BOOL  bVoiceLine;         // Indicates if the line is a voice line
  DWORD dwAPIVersion;       // API version that the line supports
  DWORD dwNumOfAddress;     // Number of available addresses on the line
  DWORD dwPermanentLineID;  // Permanent line identifier
  TCHAR szLineName[256];    // Name of the line
} LINEINFO, *LPLINEINFO;
```

Getting and Opening a Line

When a Windows CE–based application is initialized using the **lineInitialize** function, a user needs to call a phone number and get an available line using the **lineOpen** function. The application can provide a dialog box for a user to enter the phone number of their choice.

Each call from a Windows CE device is identified by a *call handle*. A call handle provides a pointer to a value to identify a specific call. TAPI assigns call handles as required. One call handle exists for every call owned by an application. Certain TAPI functions create new calls. As they do so, they return new call handles to the application.

A line may have different capabilities; and to determine these capabilities, the application should call the **lineGetDevCaps** function. The function fills in the **LINEDEVCAPS** structure, which the telephone service provider defines. The size of the structure might be different for different service providers; therefore, the application must check to see if the buffer size is adequate. To check if the amount of space supplied for the structure is sufficient for the size of the structure of the provider, compare the *dwNeededSize* and *dwTotalSize* fields. If the total size is too small, the application needs to pass a larger buffer to the function.

The **lineNegotiateAPIVersion** function indicates which version of TAPI the application supports, and negotiates which API version number TAPI should use. The reason for negotiating the TAPI version is to be sure that the correct protocol is used. New versions might define new features, new fields to data structures, and so on. Version numbers therefore indicate how to interpret various data, structures, and messages. If version ranges do not match, the application and API or service provider versions are incompatible and an error is returned.

When the function succeeds, the line data such as permanent identifier, number of addresses, and line name, can be obtained from the **LINEDEVCAPS** structure.

The following code example shows how to open a line.

```
DWORD GetLineInfo (DWORD dwLineID, LPLINEINFO lpLineInfo)
{
  DWORD dwSize,
        dwReturn;
  LPTSTR lpszLineName = NULL;
  LPLINEDEVCAPS lpLineDevCaps = NULL;

  // Negotiate the API version number. If it fails, return to dwReturn.
  if (dwReturn = lineNegotiateAPIVersion (
        g_hLineApp,                    // TAPI registration handle
        dwLineID,                      // Line device to be queried
        TAPI_VERSION_1_0,              // Least recent API version
        TAPI_CURRENT_VERSION,          // Most recent API version
        &(lpLineInfo->dwAPIVersion),   // Negotiated API version
        NULL))                         // Must be NULL; the provider-
                                       // specific extension is not
                                       // supported by Windows CE.
  {
    goto exit;
  }

  dwSize = sizeof (LINEDEVCAPS);

  // Allocate memory for lpLineDevCaps.
  do
  {
    if (!(lpLineDevCaps = (LPLINEDEVCAPS) LocalAlloc (LPTR, dwSize)))
    {
      dwReturn = LINEERR_NOMEM;
      goto exit;
    }

    lpLineDevCaps->dwTotalSize = dwSize;

    if (dwReturn = lineGetDevCaps (g_hLineApp,
                                   dwLineID,
                                   lpLineInfo->dwAPIVersion,
                                   0,
                                   lpLineDevCaps))
    {
      goto exit;
    }
```

```
    // Stop if the allocated memory is equal to or greater than the
    // needed memory.
    if (lpLineDevCaps->dwNeededSize <= lpLineDevCaps->dwTotalSize)
      break;

    dwSize = lpLineDevCaps->dwNeededSize;
    LocalFree (lpLineDevCaps);
    lpLineDevCaps = NULL;

  } while (TRUE);

  // Store the line data in *lpLineInfo.
  lpLineInfo->dwPermanentLineID = lpLineDevCaps->dwPermanentLineID;
  lpLineInfo->dwNumOfAddress = lpLineDevCaps->dwNumAddresses;
  lpLineInfo->bVoiceLine =
    (lpLineDevCaps->dwMediaModes & LINEMEDIAMODE_INTERACTIVEVOICE);

  // Allocate memory for lpszLineName.
  if (!(lpszLineName = (LPTSTR) LocalAlloc (LPTR, 512)))
  {
    dwReturn = LINEERR_NOMEM;
    goto exit;
  }

  // Store the line name in *lpszLineName.
  if (lpLineDevCaps->dwLineNameSize >= 512)
  {
    wcsncpy (
      lpszLineName,
      (LPTSTR)((LPSTR)lpLineDevCaps + lpLineDevCaps->dwLineNameOffset),
      512);
  }
  else if (lpLineDevCaps->dwLineNameSize > 0)
  {
    wcsncpy (
      lpszLineName,
      (LPTSTR)((LPSTR)lpLineDevCaps + lpLineDevCaps->dwLineNameOffset),
      lpLineDevCaps->dwLineNameSize);
  }
  else
    wsprintf (lpszLineName, TEXT("Line %d"), dwLineID);

  // Copy lpszLineName to lpLineInfo->lpszLineName.
  lstrcpy (lpLineInfo->szLineName, lpszLineName);

  dwReturn = ERR_NONE;
```

```
exit:

  if (lpLineDevCaps)
    LocalFree (lpLineDevCaps);

  if (lpszLineName)
    LocalFree (lpszLineName);

  return dwReturn;
}
```

Opening a Line and Making a Phone Call

When an application has initiated and negotiated the API, the application needs to verify if the line is usable and ready for dialing out. The application can do this by checking the values filled into the **LINEDEVCAPS** structure by calling the **lineGetDevCaps** function.

To open a line device for any purpose, the application calls the **lineOpen** function. The **lineOpen** function opens the specified line device and returns a line handle to the opened line device. This line handle is used in subsequent operations on the line device. Later, when the application is finished using the line device, it can close it with the **lineClose** function.

Before it makes a telephone call, the application needs to open the line. The **lineOpen** function opens and the **lineClose** function closes a specific TAPI device.

The **lineOpen** function specifies:

- A handle to the application registration with TAPI.
- A value that identifies the line device to be opened. Windows CE does not support the LINEMAPPER value for the *dwDeviceID* parameter.
- A pointer to a line handle loaded with the handle representing the opened line device.
- The API version number under which the application and TAPI operate compatibly. This number is obtained by calling the **lineNegotiateAPIVersion** function.
- The extension version number under which the application and the service provider operate compatibly. Windows CE does not support provider-specific extensions. The *dwExtVersion* parameter should be set to zero prior to calling **lineOpen**.

- User-instance data passed back to the application with each message associated with this line or with addresses or calls on this line. This parameter is not interpreted by the Telephony API.
- The privilege the application wants for the calls it is notified for. This parameter can be a combination of the LINECALLPRIVILEGE constants.

To place a call, the application must call the **lineMakeCall** function using the **LINECALLPARAMS** structure. TAPI sends LINE_CALLSTATE messages to indicate the progress of the call. For example, LINE_CALLSTATE indicates states of connection, dialing, proceeding, and so on. The messages vary depending on the type of call and the service provided. The application should not be designed to one type or one special sequence of call states.

The **lineMakeCall** function has the following parameters:

- A handle to the open line device on which a call is originated
- A pointer to the handle to the call. Use this call handle to identify the call when invoking other telephony operations on the call.
- A pointer to the destination address. This follows the standard area code and telephone number format.
- The country code of the called party
- A pointer to a **LINECALLPARAMS** structure. This structure enables the application to specify how to set up the call. If NULL is specified, a default 3.1 kHz channel voice call is established and an arbitrary origination address on the line is selected. This structure enables the application to select elements such as the call bearer mode, data rate, expected media, and dialing parameters.

After the **lineMakeCall** function successfully sets up the call, the application receives a LINE_REPLY message. The callback function receives the message. The LINE_REPLY message also informs the application that the call handle returned by **lineMakeCall** is valid.

When the application has opened the line successfully, it receives a handle to the line. The application can then use that line to make outbound calls.

The following code example shows how to open a line and make an outbound call.

```
VOID MakePhoneCall (LPCTSTR lpszPhoneNum)
{
  DWORD dwReturn,
        dwSizeOfTransOut = sizeof (LINETRANSLATEOUTPUT),
        dwSizeOfCallParams = sizeof (LINECALLPARAMS);

  LPLINECALLPARAMS lpCallParams = NULL;
  LPLINETRANSLATEOUTPUT lpTransOutput = NULL;

  TCHAR szDialablePhoneNum[TAPIMAXDESTADDRESSSIZE + 1] = {'\0'};

  // Initialize g_MakeCallRequestID.
  g_MakeCallRequestID = 0;

  // Open the current line.
  if (dwReturn = lineOpen (
          g_hLineApp,                  // Usage handle for TAPI
          g_dwCurrentLineID,           // Cannot use the LINEMAPPER value
          &g_CurrentLineInfo.hLine,    // Line handle
          g_CurrentLineInfo.dwAPIVersion,
                                       // API version number
          0,                           // Set to zero for Windows CE
          0,                           // No data passed back
          LINECALLPRIVILEGE_NONE,      // Can only make an outgoing call
          0,                           // Media mode
          NULL))                       // Set to NULL for Windows CE
  {
    goto exit;
  }

  // Call translate address before dialing.
  do
  {
    // Allocate memory for lpTransOutput.
    if (!(lpTransOutput = (LPLINETRANSLATEOUTPUT) LocalAlloc (
                                                    LPTR,
                                                    dwSizeOfTransOut)))
    {
      goto exit;
    }

    lpTransOutput->dwTotalSize = dwSizeOfTransOut;
```

```
    if (dwReturn = lineTranslateAddress (
          g_hLineApp,                  // Usage handle for TAPI
          g_dwCurrentLineID,           // Line device identifier
          g_CurrentLineInfo.dwAPIVersion,
                                       // TAPI version supported
          lpszPhoneNum,                // Address to be translated
          0,                           // Must be 0 for Windows CE
          0,                           // No associated operations
          lpTransOutput))              // Result of the address translation
    {
      goto exit;
    }

    if (lpTransOutput->dwNeededSize <= lpTransOutput->dwTotalSize)
      break;
    else
    {
      dwSizeOfTransOut = lpTransOutput->dwNeededSize;
      LocalFree (lpTransOutput);
      lpTransOutput = NULL;
    }

  } while (TRUE);

  dwSizeOfCallParams += lpTransOutput->dwDisplayableStringSize;

  if (!(lpCallParams = (LPLINECALLPARAMS) LocalAlloc (
                                                LPTR,
                                                dwSizeOfCallParams)))
  {
    goto exit;
  }

  // Set the call parameters.
  lpCallParams->dwTotalSize = dwSizeOfCallParams;
  lpCallParams->dwBearerMode = LINEBEARERMODE_VOICE;
  lpCallParams->dwMediaMode = LINEMEDIAMODE_DATAMODEM;
  lpCallParams->dwCallParamFlags = LINECALLPARAMFLAGS_IDLE;
  lpCallParams->dwAddressMode = LINEADDRESSMODE_ADDRESSID;
  lpCallParams->dwAddressID = g_dwCurrentLineAddr;
  lpCallParams->dwDisplayableAddressSize =
                              lpTransOutput->dwDisplayableStringSize;
  lpCallParams->dwDisplayableAddressOffset = sizeof (LINECALLPARAMS);

  // Save the translated phone number for dialing.
  lstrcpy (szDialablePhoneNum, (LPTSTR)((LPSTR)lpTransOutput + \
           lpTransOutput->dwDisplayableStringOffset));
```

```
    // Set the cursor as the wait cursor.
    SetCursor (LoadCursor (NULL, IDC_WAIT));

    // Make the phone call. lpCallParams should be NULL if the default
    // call setup parameters are requested.
    g_MakeCallRequestID = lineMakeCall (g_CurrentLineInfo.hLine,
                                        &g_hCall,
                                        szDialablePhoneNum,
                                        0,
                                        lpCallParams);

    // Set the cursor back to an arrow.
    SetCursor (0);

    if (g_MakeCallRequestID > 0)
    {
      g_bCurrentLineAvail = FALSE;

      DialogBoxParam (g_hInst,
                      MAKEINTRESOURCE(IDD_DIALING),

                      g_hwndMain,
                      (DLGPROC) DialingProc, 0);
    }
    else
    {
      ErrorBox (TEXT("Failed in making the phone call, function")
                TEXT("\nlineMakeCall failed."));
      CurrentLineClose ();
    }

  exit :

    if (lpCallParams)
      LocalFree (lpCallParams);

    if (lpTransOutput)
      LocalFree (lpTransOutput);

    // If the make call did not succeed, but the line was opened,
    // then close it.
    if ((g_MakeCallRequestID <= 0) && (g_CurrentLineInfo.hLine))
      CurrentLineClose ();

    return;
```

Opening One or More Lines

If several lines are available, a Windows CE–based application can open one or more telephone lines for outbound calls. The **lineGetDevCaps** function queries a specified line device to determine its capabilities. The data returned is valid for all line device addresses. This function determines if the line supports functions that are required in order for calls to be made.

An application can open several instances of a line; in other words, an application can obtain more than one handle to the same line. After a line is selected, the application uses the **lineOpen**, function to open the specified line.

Using the Callback Function

The callback function processes messages or notifications that TAPI sends to the application. Various information is available from the LINE_CALLSTATE message, such as that the call is receiving a dial tone from the switch or the call is receiving a busy signal, as well as other data sent by the network. A phone call passes through different stages during a session, and the application can display the result from the LINE_CALLSTATE message in a dialog box.

The following code example shows how to process LINE_CALLSTATE messages in the **lineCallbackFunc** function.

```
// This code sample shows only part of the callback function.

LPTSTR lpszStatus;

case LINE_CALLSTATE:

  // If the CALLSTATE does not apply to the call in progress, return.
  if (g_hCall != (HCALL) hDevice)
    return;

  // dwParam1 is the specific CALLSTATE change occurring
  switch (dwParam1)
  {
    case LINECALLSTATE_DIALTONE:
      lpszStatus = TEXT("Dial tone");
      break;

    case LINECALLSTATE_DIALING:
      lpszStatus = TEXT("Dialing");
      break;
```

```
    case LINECALLSTATE_PROCEEDING:
      lpszStatus = TEXT("Dialing has completed and the call ")
                   TEXT("is in proceeding");
      break;

    case LINECALLSTATE_RINGBACK:
      lpszStatus = TEXT("Ring back");
      break;

    case LINECALLSTATE_CONNECTED:
      lpszStatus = TEXT("Connected");
      break;
    .
    .
    .

    case LINECALLSTATE_DISCONNECTED:
    {
      LPTSTR lpszDisconnected;

      switch (dwParam2)
      {
        case LINEDISCONNECTMODE_NORMAL:
          lpszDisconnected = TEXT("Remote party disconnected");
          break;

        case LINEDISCONNECTMODE_UNKNOWN:
          lpszDisconnected = TEXT("Disconnected: Unknown reason");
          break;

        case LINEDISCONNECTMODE_REJECT:
          lpszDisconnected = TEXT("Remote Party rejected call");
          break;
        .
        .
        .

        default:
          lpszDisconnected = TEXT("Disconnected: Unknown reason");
          break;
      }

      ErrorBox (lpszDisconnected);
      // Insert code here to close the current open line device.
      // ...
      break;
    }
  }
  break;
```

Address Translation

A telephone number might have to be converted or translated into a dialable address format. For example, an application can help a user to avoid dialing the wrong number. The application can call a function and display a dialog box for a user to change or confirm the number. Another example might be when a call placed to a different area code is not a long-distance phone call. Some calls placed to another exchange prefix within the same area code are long distance and require that a 1 be added before dialing. The **lineTranslateDialog** function displays a dialog box where a user can change the current phone number and adjust the location and see the effect on a phone number to be dialed.

Following the call to **lineTranslateDialog**, the application must call the **lineGetTranslateCaps** function to save any changes a user made to the telephony address translation parameters, and then call **lineTranslateAddress** to obtain a dialable string based on the new user selections.

The following code example shows how to use the **lineTranslateDialog** and **lineGetTranslateCaps** functions.

```
DWORD dwCounter,
                dwSizeTranslateCaps;
LPLINELOCATIONENTRY lpLocationEntry;
LPLINETRANSLATECAPS lpLineTranslateCaps;

            // Display the modal dialog box to adjust the location.
            lineTranslateDialog (g_hLineApp,
                                 g_dwCurrentLineID,
                                 g_CurrentLineInfo.dwAPIVersion,
                                 hwnd,
                                 NULL);

            dwSizeTranslateCaps = sizeof (LINETRANSLATECAPS);

            // Allocate memory for lpLineTranslateCaps.
            do
            {
            if (!(lpLineTranslateCaps = (LINETRANSLATECAPS*) LocalAlloc(
                                         LPTR, dwSizeTranslateCaps)))
            return FALSE;

            lpLineTranslateCaps->dwTotalSize = dwSizeTranslateCaps;

            if (lineGetTranslateCaps (g_hLineApp,
                                      g_CurrentLineInfo.dwAPIVersion,
                                      lpLineTranslateCaps))
            {
```

```
    LocalFree (lpLineTranslateCaps);
    return FALSE;
    }

    if (lpLineTranslateCaps->dwNeededSize <=
                                lpLineTranslateCaps->dwTotalSize)
        break;
    else
        {
        dwSizeTranslateCaps = lpLineTranslateCaps->dwNeededSize;
        LocalFree (lpLineTranslateCaps);
        lpLineTranslateCaps = NULL;
        }
    } while (TRUE);

    lpLocationEntry = (LPLINELOCATIONENTRY)
                      ((LPBYTE)lpLineTranslateCaps +
                      lpLineTranslateCaps->dwLocationListOffset);

    // Find the selected location.
    for (dwCounter = 0;
        dwCounter < lpLineTranslateCaps->dwNumLocations;
        dwCounter++)
    {
    if (lpLocationEntry[dwCounter].dwPermanentLocationID ==
                    lpLineTranslateCaps->dwCurrentLocationID)
    break;
}

// Error occurred in finding the selected location.
if (dwCounter == lpLineTranslateCaps->dwNumLocations)
{
    LocalFree (lpLineTranslateCaps);
    return FALSE;
}

// Save the location name, country, and area code data.
wsprintf (g_szLocationName,
          (LPTSTR)((LPSTR)lpLineTranslateCaps +
          lpLocationEntry[dwCounter].dwLocationNameOffset));

wsprintf (g_szCountryCode, TEXT("%ld"),
          lpLocationEntry[dwCounter].dwCountryCode);
```

```
wsprintf (g_szAreaCode,
          (LPTSTR)((LPSTR)lpLineTranslateCaps +
          lpLocationEntry[dwCounter].dwCityCodeOffset));

LocalFree (lpLineTranslateCaps);
return TRUE;
```

The **lineTranslateAddress** function examines the registry settings to find the user location, including the country and area code. It then produces a valid dialing sequence by removing unnecessary portions of the number—such as the country code or area code—and adding other digits such as a long-distance prefix or a digit used to dial out of a local private branch exchange.

Ending a Call and Shutting Down TAPI

When a user ends the call, the application should disconnect and terminate the call with **lineDrop**. The **lineDrop** function can also be used to drop a call in progress.

When the application receives the LINE_CALLSTATE message indicating that the call has ended, the handle should be released before finishing the **lineDeallocateCall** function.

▶ **To drop a line**

1. Call the **lineDrop** function.
2. Free memory with the **lineDeallocateCall** function.

The following code example shows how to use the **lineDrop** and **lineDeallocateCall** functions.

```
g_DropCallRequestID = lineDrop (g_hCall, NULL, 0);
lineDeallocateCall (g_hCall);
```

Deallocating a call handle means that the system must free system-allocated memory related to the call after the call has been dropped. The application must call **lineDeallocatedCall** to free allocated memory.

To finally close the line, the application should call **lineClose**. After the line has closed successfully, the handle is no longer valid.

▶ **To close a line**

1. **Cancel** the call using **lineDrop**.
2. **Close** the open line using **lineClose**.
3. Reinitialize the variables.

The following code example shows how to use **lineClose** to close the current line.

```
VOID CurrentLineClose ()
{
  // Close the current line.
  if (g_CurrentLineInfo.hLine)
    lineClose (g_CurrentLineInfo.hLine);

  // Reinitialize the variables.
  g_CurrentLineInfo.hLine = NULL;
  g_bCurrentLineAvail = TRUE;
  g_hCall = NULL;
}
```

The **lineShutdown** function completely disconnects the application from TAPI. If **lineShutdown** is called when the TAPI application has lines open or calls active, the calls are closed and the line is shut down.

▶ **To disconnect TAPI from the application**

- Call **lineShutdown** to disconnect TAPI from the application.

The following code example shows how to use **lineShutdown** to disconnect.

```
lineShutdown (g_hLineApp);
```

CHAPTER 4

Remote Access Service

To access network resources from a remote location, a Windows CE–based device requires the ability to connect and communicate with a remote host computer. After the connection is established, the client can upload and download files.

The Windows CE OS includes a router, Remote Access Service (RAS), a Windows-based application which connects a remote device, known as a client, to a host computer, here known as a remote access server. Applications using RAS are usually executed on the client and connect to the remote access server by way of the telephone network, using two standard remote access protocols, point-to-point protocol (PPP) and Serial Line Internet Protocol (SLIP).

A Windows CE–based device running RAS uses PPP to connect to a remote access server. PPP is a set of industry standard framing and authentication protocols that enable remote access. Windows CE uses dial-up networking for connecting to a remote access server.

Windows CE–based applications using RAS can link to Coredll.lib to resolve RAS API entry points, the proper method for a device build, or link to the NT RAS API set, Rasapi32.lib.

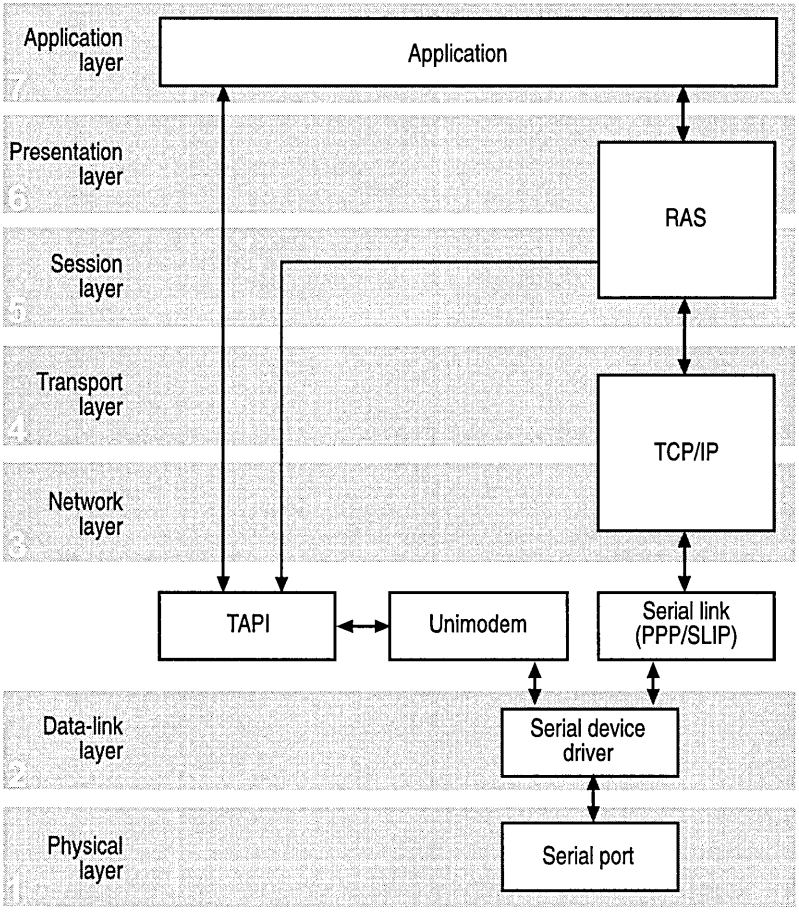
Overview

While most standard Windows-based desktop platform RAS functions are supported by Windows CE, only one point-to-point connection at a time is possible. Two connection types are direct serial and dial-up.

RAS and the OSI Model

RAS operates in the upper layers of the International Organization for Standardization Open Systems Interconnection (ISO/OSI) model for network communications. In Windows CE, RAS supports only IP-based protocols.

The following illustration shows RAS in relation to other Windows CE communications protocols within the context of the ISO/OSI model.



When creating an application requiring a connection to a remote access server, use Windows CE RAS functions to manipulate the transport layer of the selected Windows CE–based device.

Remote Access Service Functions and Structures

RAS functions are used to establish a connection with a remote access server. The functions and structures enable developers to create custom remote Windows CE–based applications that can establish a remote connection, use network resources, and reconnect in the event of a communications link failure.

Functions

The following table shows the RAS functions supported by Windows CE.

Function	Description
RasDeleteEntry	Deletes a phone-book entry in the registry.
RasDial	Establishes a RAS connection.
RasEnumConnections	Lists active RAS connections and returns each connection handle and phone-book entry name.
RasEnumEntries	Lists all registry phone-book entry names.
RasGetConnectStatus	Retrieves the current specified remote access connection status. An application can use this call to determine when a **RasDial** call has completed.
RasGetEntryDialParams	Retrieves connection data saved by the last successful call to the **RasDial** or **RasGetEntryDialParams** function for a specified phone-book entry.
RasGetEntryProperties	Retrieves phone-book entry properties.
RasHangUp	Terminates a RAS connection. The connection is specified with a RAS connection handle. **RasHangUp** releases all resources associated with the handle.
RasRenameEntry	Changes a phone-book entry name.
RasSetEntryDialParams	Changes connection data saved by the last successful call to the **RasDial** function or the **RasGetEntryDialParams** function for a specified phone-book entry.
RasSetEntryProperties	Changes the connection data for a phone-book entry or creates a new phone-book entry.
RasValidateEntryName	Validates the format of a connection entry name. The entry name must contain at least one non-white-space alphanumeric character.

Structures

The following table shows the RAS structures supported by Windows CE.

Structure	Description
RASCONN	This structure provides remote access connection data. The **RasEnumConnections** function returns an array of **RASCONN** structures.
RASCONNSTATE	This enumeration type contains values that specify connection states that may occur during a RAS connection operation.
RASCONNSTATUS	This structure describes the current status of a remote access connection. It is returned by the **RasGetConnectStatus** function.
RASDIALPARAMS	This structure contains parameters used by the **RasDial** function to establish a connection to a remote access server.
RASENTRY	This structure describes a phone-book entry. The **RasSetEntryProperties** function and the **RasGetEntryProperties** function use this structure to set and retrieve phone-book entry properties.
RASENTRYNAME	This structure contains an entry name from a remote access phone book. The **RasEnumEntries** function returns an array of **RASENTRYNAME** structures.

Synchronous Operations

Synchronous mode provides a simple way for a client to establish a connection. When **RasDial** is invoked as a synchronous operation, the function does not return until the connection is established or an error occurs. The client application can call **RasDial**, wait for the function to return, and then call the **RasGetConnectStatus** function to determine if the connection operation was successful. When the connection is established, the client application can terminate without closing the connection. If an error occurs, it must close the connection before terminating.

The disadvantage of synchronous mode is that the client does not receive progress notifications of the current connection operation state. As an alternative, a synchronous-mode client application can use a separate thread that calls **RasGetConnectStatus** to poll for and display the current state. However, for client applications that require progress data, the preferred technique is to invoke **RasDial** asynchronously.

Asynchronous Operations

When **RasDial** is invoked as an asynchronous operation, the function returns immediately. In asynchronous mode, the **RasDial** call must specify a notification handler that RAS uses to inform the client application when the connection operation state changes or an error occurs.

RAS makes its asynchronous notifications in the context of the thread that made the **RasDial** call. Therefore, the calling thread must not terminate until the connection is established or an error occurs. As in synchronous mode, the client application can safely terminate when the connection is established, and it must close the connection if an error occurs.

Phone-Book Files and Connection Data

A **RasDial** call must specify the data RAS requires to establish a connection. Typically, the **RasDial** call provides the connection data by specifying a phone-book entry. The connection data in a phone-book entry includes phone numbers, bits-per-second rate, user authentication data, and other connection data.

A client application uses **RasDial** function parameters to specify a phone-book file and an entry in that file. The *lpszPhonebookPath* parameter can specify the name of a phone-book file, or it can be NULL to indicate that the default phone-book file should be used. The *lpRasDialParams* parameter points to the **RASDIALPARAMS** structure that specifies the name of the phone-book entry to use.

To display a list of phone-book entries from which a user can select, a client application can call the **RasEnumEntries** function to enumerate the phone-book file entries.

User Authentication Data

Before authentication, the client application sends a user name and password to the remote access server. The server uses this data to authenticate a user. It can use the **RASDIALPARAMS** structure specified in the **RasDial** call to specify a user name and password.

If the server cannot authenticate with the specified data, it can enable the connection operation to enter a paused state so that the client application can query the user for correct authentication data.

The **RASDIALPARAMS** structure can also specify the network domain name on which authentication occurs.

Handling Errors

When an error occurs, the RAS invokes the client notification handler. The notification includes the connection state when the error occurred and a code that identifies the error. In these cases, the notification handler should call the **RasHangUp** function to end the RAS connection.

Informational Notifications

For running states, no action is required of the notification handler unless an error occurs. Running states occur during the connection operations that RAS handles automatically, such as connecting to necessary devices, user authentication, and waiting for a callback from the remote access server. The notification serves as a progress report to the client.

The client can pass notifications to a user. In some running states, the client might display additional information to a user. For example, a notification handler that receives a RASCS_ConnectDevice state can call the **RasGetConnectStatus** function to get the name and type of the device connected to. Another example is when the client receives a RASCS_Projected state. This occurs when the RAS projection phase of the connection operation is complete.

Completion Notifications

RAS continues to process notifications until connection operations are complete. This occurs when:

- The connection is established. The handler receives a RASCS_Connected notification. The client application can exit without closing the connection.
- An error occurs. The handler receives a notification indicating the error and the connection state when the error occurred.
- The connection operation is interrupted by a **RasHangUp** function call.

Disconnecting a RAS Connection

When a client application starts a connection operation, the **RasDial** function call receives an **HRASCONN** connection handle to identify the connection. If the returned handle is not NULL, the client must eventually call the **RasHangUp** function to end the connection. If an error occurs during the connection operation, the client must call **RasHangUp** even though the connection was never established.

A client application might require a connection to end even though it does not have the handle returned by **RasDial**. For example, the application that called **RasDial** might have exited when the connection was established. In this case, the disconnecting application can use the **RasEnumConnections** function to get data on all current connections. For each connection, **RasEnumConnections** returns a **RASCONN** structure containing the **HRASCONN** connection handle and the phone-book entry name or phone number specified when the connection operation started. This data can be used to display a list of connections from which a user can select the connection to close.

Phone-Book Entries

Phone-book entries, stored in the registry, contain data necessary to establish a RAS connection.

Windows CE supports a limited set of the Windows-based desktop platform functions for working with phone-book entries. You can use the **RasGetEntryDialParams** or the **RasSetEntryDialParams** function to set or retrieve the connection parameters for a phone-book entry. The **RasEnumEntries** function retrieves an array of **RASENTRYNAME** structures that contain the phone-book entry names.

You can use the **RasRenameEntry** function to rename a phone-book entry, or the **RasDeleteEntry** function to delete an entry. The **RasValidateEntryName** function determines if a specified string has the correct format to be used as an entry name.

You can use the **RasGetEntryProperties** and **RasSetEntryProperties** functions to get and set additional phone-book entry data. These functions use a **RASENTRY** structure.

Accessing the Internet Using a Modem

With a modem installed, a Windows CE–based device can be used to establish a RAS connection. If your device is not equipped with a modem, a PC Card modem or an external modem is required to use RAS. See the modem manufacturer's installation guide.

With a device, such as the Handheld PC running Windows CE, a user can access the Internet. Using Windows CE and RAS, a user can make an Internet Protocol (IP) over a PPP connection to an Internet host computer.

A corporation can establish a remote access server using Windows NT with connections, through a router, to the Internet. The server can operate isolated from the corporate network for added security. Users can dial one number for both Internet access and access to the corporate local area network (LAN).

Sample Application

This section describes and demonstrates the RAS API set. The sample application starts a RAS connection, dials entries from the default phone book, and closes an active connection.

Starting a RAS Connection

The files, Ppp.dll, Afd.dll, Tcpstkl.dll, and Cxport.dll, are necessary to start a RAS connection. They are stored in the Windows directory on the Windows CE–based device. When RAS starts, it reads the registry phone-book entries. Registry entries contain RAS entry settings, including the device used to dial, the phone number, and other data.

▶ **To start a RAS connection**

1. Open a RAS application, such as Remote Networking.
2. Select an entry name from the list or create a new entry.
3. Enter a user name, domain, and password.

The client logs onto the remote access server and starts the connection.

In some companies, a dial-up remote access server is provided for remote access to the corporate LAN. On a Handheld PC, connecting to a LAN by way of RAS is similar to connecting to an Internet service provider (ISP). You dial in and log on with a user name and password.

Connection Operation

The following section describes how to establish and terminate a connection as well as how to retrieve connection data.

Establishing a Connection

The Windows CE **RasDial** function indicates a successful connection in two ways. If RasDial makes a successful connection:

- It returns a zero, a non-zero value indicates failure.
- In addition, the function stores a handle to the RAS connection into the variable point to by the function parameter *pRasConn*.

The **RasDial** function must specify data so that RAS can establish the connection from a Windows CE–based device to a remote access server. The client uses the **RasDial** function parameters to specify a phone-book file and a phone-book entry. The *lpszPhonebookPath* parameter can specify the name of a phone-book file, or it can be NULL to indicate that the default phone-book file should be used. The *lpRasDialParams* parameter points to a **RASDIALPARAMS** structure that specifies the phone-book entry to use. To connect, the **RasDial** function must specify the data necessary to establish a connection. Typically, the **RasDial** call provides the connection data by specifying a phone-book entry using the **RASDIALPARAMS** structure to provide data such as phone number, user name, domain, and password.

The connection data includes callback and user authentication data. To make a connection, the **RasDial** function can specify an empty string for the *szEntryName* member of the **RASDIALPARAMS** structure. The *szPhoneNumber* member must contain the phone number to dial.

▶ **To use RasDial to establish a connection**

1. Set the *dialExtensions* parameter to NULL.
2. Set the *lpszPhonebook* parameter to NULL. Phone-book entries are stored in the registry rather than in a phone-book file.
3. Set the *dwNotiferType* parameter to 0xFFFFFFFF, specifying the *lpvNotifier* parameter as a handle to the window receiving progress notification messages. If the application requires messages from RAS, the messages must be sent to a window handle. There is no support for callback functions.
4. Set the *szEntryName, szUserName, szPassword, and szDomain* members of the **RASDIALPARAM** structure. Pass a pointer to this structure into *lpRasDialParam.*

Note **RasDial** does not automatically display the logon dialog box. This is currently done through the Remote Networking application. Your application is responsible for getting the data from a user.

The following code example shows how to establish a RAS connection.

```
BOOL MakeRasDial (HWND hDlgWnd)
{
  BOOL bPassword;
  TCHAR szBuffer[100];

  if (bUseCurrent)
  {
    // Get the last configuration parameters used for this connection.
    // If the password was saved, then the logon dialog box will not be
    // displayed.
    if (RasGetEntryDialParams (NULL, &RasDialParams, &bPassword) != 0)
    {
      MessageBox (hDlgWnd,
                  TEXT("Could not get parameter details"),
                  szTitle,
                  MB_OK);
      return FALSE;
    }
  }
  else
  {
    // Display the Authentication dialog box.
    DialogBox (hInst, MAKEINTRESOURCE(IDD_AUTHDLG), hDlgWnd,
               AuthDlgProc);

    // Set hRasConn to NULL before attempting to connect.
    hRasConn = NULL;

    // Initialize the structure.
    memset (&RasDialParams, 0, sizeof (RASDIALPARAMS));

    // Configure the RASDIALPARAMS structure.
    RasDialParams.dwSize = sizeof (RASDIALPARAMS);
    RasDialParams.szPhoneNumber[0] = TEXT('\0');
    RasDialParams.szCallbackNumber[0] = TEXT('\0');
    wcscpy (RasDialParams.szEntryName, szRasEntryName);
    wcscpy (RasDialParams.szUserName, szUserName);
    wcscpy (RasDialParams.szPassword, szPassword);
    wcscpy (RasDialParams.szDomain, szDomain);
  }
```

```
  // Try to establish RAS connection.
  if (RasDial (NULL,            // Extension not supported
               NULL,            // Phone book is in registry
               &RasDialParams,  // RAS configuration for connection
               0xFFFFFFFF,      // Use this value
               hDlgWnd,         // Window receives notification message
               &hRasConn) != 0) // Connection handle
  {
    MessageBox (hDlgWnd,
                TEXT("Could not connect using RAS"),
                szTitle,
                MB_OK);
    return FALSE;
  }

  wsprintf (szBuffer, TEXT("Dialing %s..."), szRasEntryName);

  // Set the Dialing dialog box window name to szBuffer.
  SetWindowText (hDlgWnd, szBuffer);

  return TRUE;
}
```

Connection Data

While connected to a remote access server, the application can use the **RasEnumConnections** function to receive data about existing device connections. The data for each connection includes a connection handle and the name of the phone-book entry used to establish the connection. You can use the connection handle in a call to the **RasGetConnectStatus** function get the current connection status. **RasGetConnectStatus** retrieves data on the current status of the specified remote access connection. An application can use **RasGetConnectStatus** to determine the status of the **RasDial** function. **RASCONNSTATE** specifies an enumerator value that indicates the current state of the **RasDial** connection process and determines which part of the **RasDial** function is currently executing. The application can call the **RasGetConnectStatus** function to determine the name and device type, or if the device has successfully connected.

The following list is an example of status values available to the application:

- A port is about to be opened.
- A port has been opened successfully.
- A device is about to be connected.
- A device has connected successfully.
- The authentication process is starting.

▶ **To get current status of an existing connection**

- Call the **RasGetConnectStatus** function.

The following code example shows how to get current status of an existing connection.

```
// Get the connection status.
RasStatus.dwSize = sizeof (RASCONNSTATUS);
dwReturn = RasGetConnectStatus (hRasConn, &RasStatus);

// If there is an error in getting the connection status
if (dwReturn)
{
  wsprintf (szBuffer,
            TEXT("Failed getting connect status.\r\n")
            TEXT("Error (%ld)."),
            dwReturn);
  MessageBox (hMainWnd, szBuffer, TEXT("Warning"), MB_OK);
  break;
}
```

Connection Sequence

Understanding the RAS connection sequence helps you understand PPP.

1. PPP operations begin upon connecting to a remote access server.
2. Framing rules are established between the client and server. This enables communication frame transfer to occur.
3. The remote access server authenticates the client using the PPP authentication protocols in serial-link communication: Password Authentication Protocol (PAP), Challenge Handshake Authentication Protocol (CHAP), and Microsoft CHAP. The protocols invoked depend on the security configurations of the client and server.
4. When the client is authenticated, Network Control Protocol (NCP) is used to enable and configure the server for the LAN protocol used for the client.

When the PPP connection sequence has successfully completed, the client and server can begin to transfer data using any supported protocol, such as Windows Sockets, remote procedure call (RPC), or NetBIOS.

Connection States

During the process of a client connecting to a remote access server, the client performs several actions to establish the connection. Each step is identified by a connection state. The **RASCONNSTATE** structure is a set of values that correspond to these connection states. The connection states can be divided into three groups:

- Running states
- Paused states
- Terminal states

Running states are connection operations RAS handles automatically such as connecting to a device and authentication. Unless an error occurs, no action is required of the client other than to pass notification to a user.

Paused states occur when the remote access server pauses the connection operation to get additional user input. During a paused state, a user can type a number, a different user name and password if the user authentication fails, or a new password if the old one has expired.

Terminal states occur when the connection is successfully established, the connection operation fails, or the connection is closed by a call to the **RasHangUp** function.

There are several mechanisms a client can use to determine the current state of a connection operation. When a client application calls the **RasDial** function asynchronously, RAS sends progress notifications to the client notification handler when the connection state changes. In addition, the client can use the **RasGetConnectStatus** function to get the current state of any RAS connection operation.

Terminating a Connection

When a RAS application starts a connection operation, the **RasDial** function receives a **RASCONN** connection handle to identify the connection. If the returned handle is not NULL, the client must eventually call the **RasHangUp** function to end the connection. If an error occurs during the connection operation, the client must call the **RasHangUp** function even though the connection was never established.

A client application might require a connection to end even though it does not have the handle returned by **RasDial**. For example, the application that called **RasDial** might have exited when the connection was successfully established. In this case, the disconnecting application can use **RasEnumConnections** to get all the current connections. For each connection, **RasEnumConnections** returns a **RASCONN** structure containing the **RASCONN** connection handle and the phone-book entry name or phone number specified when the connection operation was started. This data can be used to display a list of connections from which a user can select the connection to end.

If the application exits after calling **RasHangUp**, there may not be enough time for the modem to reset. To provide the modem time to reset, pause for a few seconds while the application exits.

Before an application terminates one or more connections, it is useful to enumerate the existing connections and hang up. The **RasEnumConnections** retrieves the current connections. The **RasHangUp** function terminates all remote connections.

▶ **To terminate a connection**

1. Call **RasEnumConnections** to find all RAS connections.
2. Call **RasHangUp** to terminate the connection.

The following code example shows how to terminate a connection.

```
DWORD CloseRasConnections ()
{
  int index;                  // An integer index
  TCHAR szError[100];         // Buffer for error codes
  DWORD dwError,              // Error code from a function call
        dwRasConnSize,        // Size of RasConn in bytes
        dwNumConnections;     // Number of connections found
  RASCONN RasConn[20];        // Buffer for connection state data
                              // Assume the maximum number of entries is
                              // 20.

  // Assume no more than 20 connections.
  RasConn[0].dwSize = sizeof (RASCONN);
  dwRasConnSize = 20 * sizeof (RASCONN);
```

```
    // Find all connections.
    if (dwError = RasEnumConnections (RasConn, &dwRasConnSize,
                                      &dwNumConnections))
    {
      wsprintf (szError, TEXT("RasEnumConnections Error: %ld"), dwError);
      return dwError;
    }

    // If there are no connections, return zero.
    if (!dwNumConnections)
    {
      wsprintf (szError, TEXT("No open RAS connections"));
      return 0;
    }

    // Terminate all of the remote access connections.
    for (index = 0; index < (int)dwNumConnections; ++index)
    {
      if (dwError = RasHangUp (RasConn[index].hrasconn))
      {
        wsprintf (szError, TEXT("RasHangUp Error: %ld"), dwError);
        return dwError;
      }
    }

    return 0;
}
```

Phone-Book Operation

Entries in the Remote Access Service phone-book contain the data necessary to establish a RAS connection. Unlike Windows-based desktop platforms, which keep phone-book entries in a file, Windows CE stores these entries in the registry. Phone-book entries contain the data necessary to establish a RAS connection.

To make a connection without using a phone-book entry, **RasDial** can specify an empty string for the *szEntryName* member of the **RASDIALPARAMS** structure. The *szPhoneNumber* member must contain the phone number to dial.

RAS phone-book data includes:

- The phone number to dial, including country code and area code.
- The IP addresses to use while the connection is active.
- The network protocols.
- The type of device used to make the connection.

Windows CE supports a limited set of Window-based desktop platform functions for working with phone-book entries. The application can use the **RasGetEntryProperties** or the **RasSetEntryProperties** function to create or edit a phone-book entry. The application can use the **RasGetEntryDialParams** and **RasSetEntryDialParams** functions to set or retrieve the connection parameters for a phone-book entry. The **RasEnumEntries** function lists all the phone-book entry names using the **RASENTRY** structure.

The configuration file, which stores data for the RAS phone book, is kept in the registry under the **HKEY_CURRENT_USER\Comm\RasBook** key. Entry names for Windows CE must be legal registry keys and cannot exceed 20 characters.

Creating or Changing a Phone-Book Entry

RasSetEntryProperties creates or changes connection data for a phone-book entry.

▶ **To use the RasSetEntryProperties to create or change registry phone-book entries**

1. Call **RasValidateEntryName** function to be sure that the name does not exist.
2. If you are editing an existing entry, call **RasGetEntryProperties** to retrieve the existing configuration.
3. Call **RasSetEntryProperties** to set the new configuration. Parameters should be set as follows:

   The *lpszName* parameter should point to the null-terminated string containing the entry name in the *lpszEntry* parameter. If the existing name matches an existing entry, **RasSetEntryProperties** modifies the entry properties. If the entry does not exist, the function creates a new entry.

   The *lpszEntry* parameter should point to a **RASENTRY** structure, which should be filled with data for the connection, including the telephone number, device type, and name.

   The *lpbDeviceInfo* and *dwDeviceInfoSize* parameters can be used to set a Telephony API (TAPI) device configuration data.

The following code example shows how to create a phone-book entry.

```
int CreateRasEntry (LPTSTR lpszName)
{
  DWORD dwSize,
        dwError;
  TCHAR szError[100];
  RASENTRY RasEntry;
  RASDIALPARAMS  RasDialParams;

  // Validate the format of a connection entry name.
  if (dwError = RasValidateEntryName (NULL, lpszName))
  {
    wsprintf (szError, TEXT("Unable to validate entry name.")
              TEXT(" Error %ld"), dwError);

    return FALSE;
  }

  // Initialize the RASENTRY structure.
  memset (&RasEntry, 0, sizeof (RASENTRY));

  dwSize = sizeof (RASENTRY);
  RasEntry.dwSize = dwSize;

  // Retrieve the entry properties.
  if (dwError = RasGetEntryProperties (NULL, TEXT(""),
                    (LPBYTE)&RasEntry, &dwSize, NULL, NULL))
  {
    wsprintf (szError, TEXT("Unable to read default entry properties.")
              TEXT(" Error %ld"), dwError);
    return FALSE;
  }

  // Insert code here to fill the RASENTRY structure.
  // ...

  // Create a new phone-book entry.
  if (dwError = RasSetEntryProperties (NULL, lpszName,
                    (LPBYTE)&RasEntry, sizeof (RASENTRY), NULL, 0))
  {
    wsprintf (szError, TEXT("Unable to create the phonebook entry.")
              TEXT(" Error %ld"), dwError);
    return FALSE;
  }
```

```
  // Initialize the RASDIALPARAMS structure.
  memset (&RasDialParams, 0, sizeof (RASDIALPARAMS));
  RasDialParams.dwSize = sizeof (RASDIALPARAMS);
  _tcscpy (RasDialParams.szEntryName, lpszName);

  // Insert code here to fill up the RASDIALPARAMS structure.
  // ...

  // Change the connection data.
  if (dwError = RasSetEntryDialParams (NULL, &RasDialParams, FALSE))
  {
    wsprintf (szError, TEXT("Unable to set the connection information.")
              TEXT(" Error %ld"), dwError);
    return FALSE;
  }

  return TRUE;
}
```

Changing an Existing Phone-Book Entry

The **RasRenameEntry** function renames a phone-book entry in the registry, while the **RasDeleteEntry** function deletes an entry. To verify if the string is in the correct format, the application can use the **RasValidateEntryName** function.

▶ **To change a registry phone-book entry**

1. Call **RasValidateEntryName** to verify the new name.
2. Call the **RasRenameEntry** function to change the name.
3. The return value is zero if successful, and a RAS error value if not.

The following code example shows how to change an existing phone-book entry.

```
BOOL RenameRasEntry (LPTSTR lpszOldName, LPTSTR lpszNewName)

{
  DWORD dwError;            // Return code from functions
  TCHAR szError[120];       // Buffer for error message

  if (dwError = RasValidateEntryName (NULL, lpszNewName))
  {
    wsprintf (szError, TEXT("Entry name validation failed: %ld"),
              dwError);
    return FALSE;
  }
```

```
  if (dwError = RasRenameEntry (NULL, lpszOldName, lpszNewName))
  {
    wsprintf (szError, TEXT("Unable to rename entry: %ld"), dwError);
    return FALSE;
  }

  return TRUE;
}
```

Enumerating Phone-Book Entries

The **RasEnumEntries** function is used to enumerate a list of phone-book entries. Using this list, a user can select a RAS connection.

▶ **To enumerate phone-book entries**

1. Allocate an array for the **RASENTRYNAME** structure.
2. Call **RasEnumEntries** to list the names in the phone book.
3. Display the list for a user.

The following code example shows how to dial and connect to a phone-book entry in the registry.

```
BOOL GetPhonebookEntries (HWND hDlgWnd)
{
  int index;
  DWORD dwSize,
        dwEntries;
  HWND hWndListBox;

  // Allocate an array of RASENTRYNAME structures. Assume
  // no more than 20 entries to be configured on the
  // Windows CE-based device.

  if (!(lpRasEntryName = new RASENTRYNAME [20]))
  {
    MessageBox (hDlgWnd, TEXT("Not enough memory"), szTitle, MB_OK);
    return FALSE;
  }

  // Initialize the dwSize member of the first RASENTRYNAME structure
  // in the array to the size of the structure to identify
  // the version of the structure being passed.
  lpRasEntryName[0].dwSize = sizeof (RASENTRYNAME);
```

```
    // Size of the array, in bytes
    dwSize = sizeof (RASENTRYNAME) * 20;

    // List all entry names in a remote access phone book.
    if ((RasEnumEntries (
            NULL,                 // Reserved, must be NULL
            NULL,                 // Phone book is stored in the Windows CE
                                  // registry.
            lpRasEntryName,       // Pointer to structure to receive entries
            &dwSize,              // Size of lpRasEntryName, in bytes
            &dwEntries)) != 0)    // Number of entries placed in array
    {
      MessageBox (hDlgWnd, TEXT("Could not obtain RAS entries"), szTitle,
                  MB_OK);
      return FALSE;
    }

    // Get the HWND of the list box control.
    hWndListBox = GetDlgItem (hDlgWnd, IDC_RASNAMES);

    // Remove all items from a list box.
    SendMessage (hWndListBox, LB_RESETCONTENT, 0, 0);

    // Add the names of each RAS connection to the list box.
    for (index = 0; index < (int)dwEntries; ++index)
    {
      SendMessage (hWndListBox, LB_INSERTSTRING, index,
                   (LPARAM)lpRasEntryName[index].szEntryName);
    }

    return TRUE;
}
```

Copying a Phone-Book Entry

An application can use the **RasGetEntryProperties** function or the **RasSetEntryProperties** function to copy the configuration data to another entry. The *lpszEntryName* parameter can be used to change the entry.

▶ **To copy a phone-book entry to the dialing list**

1. Call **RasGetEntryProperties** to get current phone-book properties.
2. Call **RasSetEntryProperties** to set new phone-book properties.
3. Call **RasGetEntryDialParams** to retrieve current user data.
4. Call **RasSetEntryDialParams** to set user password data.
5. The return value is TRUE if successful, and FALSE if not.

The following code example shows how to copy a RAS entry.

```
BOOL CopyRasEntry (LPTSTR lpszEntryName)
{
  BOOL bPasswordSaved;
  TCHAR szNewEntryName[100];        // Name for the copied entry
  TCHAR szError[100];               // Buffer for the error message
  DWORD dwError,                    // Return value from the functions
        dwRasEntrySize,             // Size of the RASENTRY structure
        dwDevConfigSize,            // Size of DevConfigBuf
        dwDeviceNum = 0xFFFFFFFF;   // Telephony API device number
  BYTE DevConfigBuf[128];           // Buffer for device configuration
                                    // data
  RASENTRY RasEntry;                // RASENTRY structure
  RASDIALPARAMS RasDialParams;      // RASDIALPARAMS structure
  LPVARSTRING lpDevConfig = (LPVARSTRING)&DevConfigBuf;
                                    // Pointer to the memory location of
                                    // the device configuration structure

  // Assign the name for the copied phone-book entry.
  wsprintf (szNewEntryName, TEXT("%s1"), lpszEntryName);

  // Validate the format of a connection entry name.
  if (dwError = RasValidateEntryName (NULL, szNewEntryName))
  {
    wsprintf (szError, TEXT("Unable to validate entry name.")
              TEXT(" Error %ld"), dwError);

    return FALSE;
  }

  dwDevConfigSize = sizeof (DevConfigBuf);
  dwRasEntrySize = sizeof (RASENTRY);
  RasEntry.dwSize = dwRasEntrySize;

  // Retrieve the entry properties.
  if (dwError = RasGetEntryProperties (NULL,
                                       lpszEntryName,
                                       (LPBYTE)&RasEntry,
                                       &dwRasEntrySize,
                                       DevConfigBuf,
                                       &dwDevConfigSize))
  {
    wsprintf (szError, TEXT("Unable to read entry properties.")
              TEXT(" Error %ld"), dwError);
    return FALSE;
  }
```

```
    memset (&RasDialParams, 0, sizeof (RasDialParams));
    RasDialParams.dwSize = sizeof (RASDIALPARAMS);
    _tcscpy (RasDialParams.szEntryName, lpszEntryName);

    // Retrieve the connection data.
    if (dwError = RasGetEntryDialParams (NULL, &RasDialParams,
                        &bPasswordSaved))
    {
      wsprintf (szError, TEXT("Unable to get the connection information.")
                TEXT(" Error %ld"), dwError);
      return FALSE;
    }

  // Create a new phone-book entry.
    if (dwError = RasSetEntryProperties (NULL,
                                         szNewEntryName,
                                         (LPBYTE)&RasEntry,
                                         dwRasEntrySize,
                                         DevConfigBuf,
                                         dwDevConfigSize))
    {
      wsprintf (szError, TEXT("Unable to copy the phonebook entry.")
                TEXT(" Error %ld"), dwError);
      return FALSE;
    }

    _tcscpy (RasDialParams.szEntryName, szNewEntryName);

    // Change the connection data.
    if (dwError = RasSetEntryDialParams (NULL, &RasDialParams, FALSE))
    {
      wsprintf (szError, TEXT("Unable to set the connection information.")
                TEXT(" Error %ld"), dwError);
      return FALSE;
    }

    return TRUE;
  }
```

Deleting a Phone-Book Entry

To delete a phone-book entry the application can use the **RasDeleteEntry** function.

▶ **To delete a phone-book entry from the dialing list**

1. Call **RasDeleteEntry** using the *lpszName* parameter to supply the entry to be deleted.
2. The return value is TRUE if successful, and FALSE if not.

The following code example shows how to delete a RAS phone-book entry from the dialing list.

```
BOOL DeleteRasEntry (LPTSTR lpszName)
{
  DWORD dwError;            // Return code from RasDeleteEntry
  TCHAR szError[100];       // Buffer for error message

  if (dwError = RasDeleteEntry (NULL, lpszName))
  {
    wsprintf (szError, TEXT("Unable to delete entry: %ld"), dwError);
    return FALSE;
  }

  return TRUE;
}
```

CHAPTER 5

Windows Sockets

Most Microsoft Windows CE network communications pass through the *Windows Sockets* (Winsock) interface. Sockets is a general-purpose networking API. The Microsoft Windows implementation of sockets, Winsock, is designed to run efficiently on Windows operating systems while maintaining compatibility with the Berkeley Software Distribution (BSD) standard, known as Berkeley Sockets.

The Windows Internet API (WinInet) uses Winsock internally to handle network connections. However, Winsock can be used directly in applications. The following sections discuss how to use Winsock in Windows CE–based applications to directly control the creation and management of socket connections. TCP/IP, upon which Winsock is built, is also discussed.

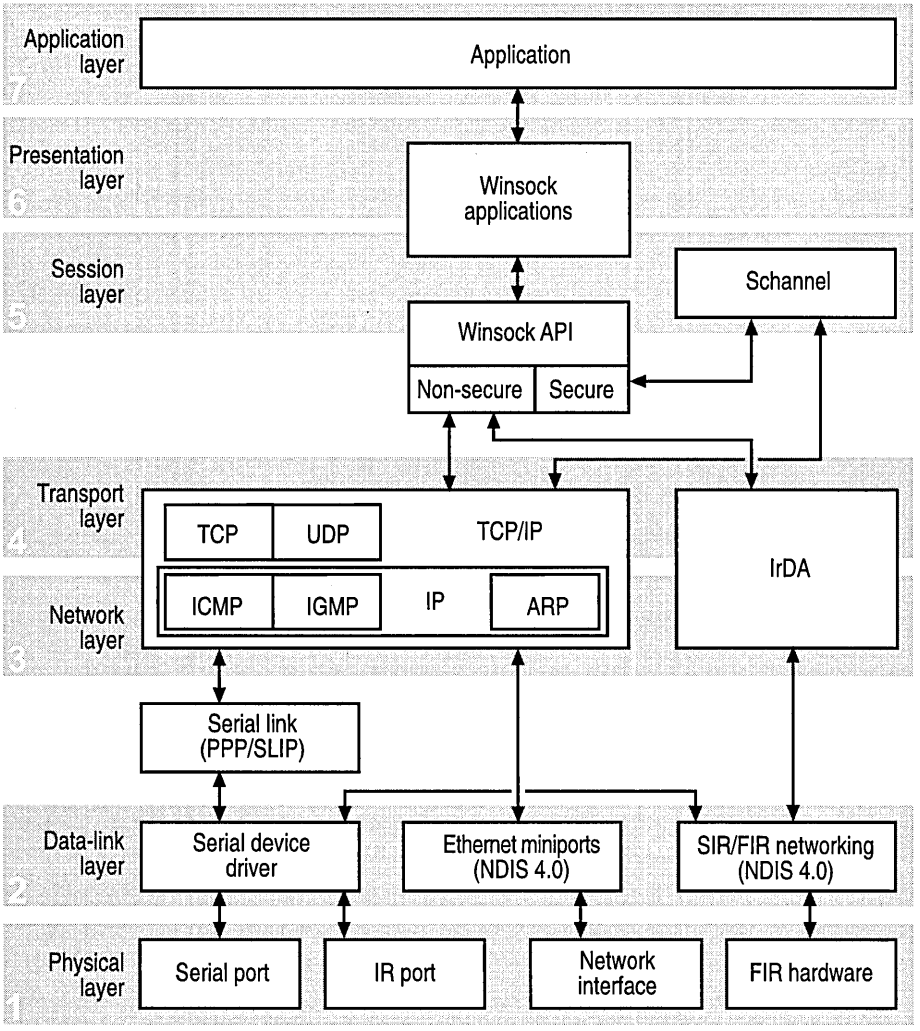
Winsock supports socket-based infrared communications using industry standard *Infrared Data Association* (IrDA) protocols. This support is referred to as *Infrared Sockets* (IrSock). Applications implement Infrared Sockets in the same way as conventional Winsock, although some Winsock functions are used differently. For more information on the using the IrDA protocols with Windows Sockets, see Using Winsock Functions with IrDA.

Windows CE supports Private Communication Technology 1.0 and secure socket layer (SSL) versions 2.0 and 3.0 security protocols. These protocols are available either directly from Winsock or through WinInet. For more information on using security protocols directly from Winsock, see Using Secure Sockets. For more information on using security protocols through WinInet, see Internet Connections.

Winsock and the OSI Model

In the International Organization for Standardization Open Systems Interconnection (ISO/OSI) model for network communications, Winsock operates at the session layer interface to the transport layer. Winsock is an interface between applications and the transport protocol and works as a conduit for data I/O.

The following illustration shows Winsock in relation to other Windows CE communications protocols within the context of the ISO/OSI model.



Winsock simplifies application development in the upper ISO/OSI layers by handling the details of network data exchange at the lower layers. Winsock provides a programmable interface between the upper layers, 5-7, and the lower layers, 1-4. Winsock applications reside in the upper, application, presentation, and session layers. Winsock application data is packaged and transmitted over a network by the lower, transport, network, data-link, and physical layers.

TCP/IP and IrDA are lower layer protocols used by Winsock. Specifically, TCP fits into the transport layer and IP fits into the network layer.

Note Winsock provides the only way for an application to access the TCP/IP or IrDA protocols on a Windows CE–based device.

TCP/IP

Winsock is above the TCP/IP protocol stack in the ISO/OSI network communications model. TCP/IP is an industry standard communications protocol that defines methods for packaging data into packets for transmission between computing devices on a heterogeneous network. TCP/IP is the standard for data transmission over networks, including the Internet. TCP establishes a connection for data transmission and IP defines the method for sending data packets.

TCP/IP Transport Layer Protocols

Winsock provides direct access to services in the transport layer of the ISO/OSI model. Two TCP/IP protocols, the TCP and the User Datagram Protocol (UDP), provide these transport services. In Windows CE, Winsock is implemented as a DLL that enables applications and transport services to be dynamically linked at run time.

TCP

TCP is a *connection-based*, stream-oriented delivery service with end-to-end error detection and correction. Connection-based means that a communications session between *hosts* is established before exchanging data. A host is any device on a TCP/IP network identified by a logical IP address.

TCP provides reliable data delivery and ease of use. Specifically, TCP notifies the sender of packet delivery, guarantees that packets are delivered in the same order in which they were sent, retransmits lost packets, and ensures that data packets are not duplicated.

User Datagram Protocol

UDP provides *connectionless*, but unreliable datagram transport services. Connectionless means that a communications session between hosts is not established before exchanging data. The UDP connectionless datagram delivery service is unreliable because it does not guarantee data packet delivery and no notification is sent if a packet is not delivered. Also, UDP does not guarantee that packets are delivered in the same order in which they were sent.

Although UDP appears to have some limitations, it is useful in certain situations. For example, Winsock IP multicasting is implemented with UDP datagram type sockets. UDP is very efficient because of low overhead. Unreliability can be overcome by building error handling into the application. For more information about IP multicasting, see Creating an IP Multicast Application.

TCP/IP Network Layer Protocols

The ISO/OSI model network layer, sometimes called the Internet layer, defines and handles the routing of datagrams. A *datagram* is a self-contained, independent packet, carrying sufficient data to be routed from source to destination without relying on exchanges between the source and destination computer and the transporting network. TCP/IP protocols residing in the network layer are: the Internet Protocol (IP), the Internet Control Message Protocol (ICMP), the Internet Group Membership Protocol (IGMP), and the Address Resolution Protocol (ARP).

Internet Protocol

IP is central to the TCP/IP stack—all other TCP/IP protocols use IP—and all data passes through it. IP is a connectionless protocol and has some limitations. If IP attempts packet delivery and, in the process, a packet is lost, delivered out of sequence, duplicated, or delayed, neither sender nor receiver is informed. Packet acknowledgment is handled by a higher-layer transport protocol, such as TCP.

IP is responsible for addressing and routing packets between hosts, and for *fragmentation*. Fragmentation is the process of breaking a datagram into smaller pieces for inter-network routing. IP fragments packets prior to sending them and reassembles them upon receipt.

Note Winsock applications can send packets, but cannot affect packet routing or fragmentation.

For more information, see IP Addressing.

Internet Control Message Protocol

ICMP is a network layer protocol that delivers flow control, error messages, routing, and other data between Internet hosts. ICMP is primarily used by application developers for a network *ping*, which is also known as Packet Internet Groper. A ping is the process of sending an echo message to an IP address and reading the reply to verify a connection between TCP/IP hosts.

Use the following Winsock API functions to write a ping application: **IcmpCreateFile**, **IcmpSendEcho**, and **IcmpCloseHandle**.

▶ **To send an ICMP request or determine if a host is available**

1. Call **IcmpCreateFile** to create a handle to issue requests.
2. Call **IcmpSendEcho** to send an ICMP echo request.

   It returns any ICMP response from the intended host recipient, or returns an error if the network is inaccessible. A time-out value may be specified to limit the wait time in case the destination is inaccessible.
3. Call **IcmpCloseHandle** to close the handle created by **IcmpCreateFile**.

Internet Group Membership Protocol

IGMP is used for IP *multicast*. A multicast is a communication between a single sender and multiple receivers on a network. IGMP is used to exchange membership status data between IP routers that support multicasting and members of multicast groups. A *router* is an intermediary device on a communications network that expedites message delivery by finding the most efficient route for a message packet within a network, or by routing packets from one *subnetwork* to another. A subnetwork is a separate part of an organization's network identified through IP addressing.

Host membership in a multicast group is reported by individual member hosts and membership status is periodically polled by multicast routers. Multicast addresses are reserved from within a standard specified range of all IP addresses to enable forwarding across routers configured to permit multicasting.

IP provides a mechanism to send and receive multicast IP traffic. Multicast IP traffic is sent to a single *media access control* address, but is processed by multiple IP hosts. A specified host listens on a specific IP multicast address and receives all packets on that address. This requires IP multicast support on the IP routers and the ability for hosts to register themselves with the router. Host registration is accomplished using IGMP.

For IP multicasting to span routers across networks, a protocol is required to inform routers that hosts of a specific multicast group are available on a specified network. This data is passed among routers through the use of multicast routing protocols. These protocols ensure that each router that supports the forwarding of multicasts is aware of which host groups are on which network. For more information about IP multicasting, see Creating an IP Multicast Application.

Address Resolution Protocol

Address Resolution Protocol (ARP) translates IP addresses to Ethernet hardware addresses. Each Ethernet network interface has a unique hardware address. Address resolution maps a host IP address to its unique network interface hardware address. An ARP request is sent to the network; the node that has the IP address returns its hardware address.

Note ARP is used to refer to the process of finding the hardware address and its opposite, Reverse Address Resolution Protocol (RARP). RARP finds the IP address using the hardware address. ARP is transparent to Winsock applications, but when address resolution fails, it usually causes a Winsock function error.

IP Addressing

A unique IP address is required for each host using TCP/IP. Network applications that use TCP/IP identify other network hosts using IP addresses. The IP address provides the directions to the exact location of a host device on a network.

If IP determines that a destination address is an address on the local network, IP transmits the packet directly to the network host. If IP determines that the destination IP address is not on the local network, IP looks for a route to a remote host. An address on the local network is a *local address* and an address not on the local network is a *remote address*. If a route is found, IP sends the packet using that route. If a route is not found, the packet is sent to the source host's default *gateway*. A gateway is a device that connects networks using different communications protocols.

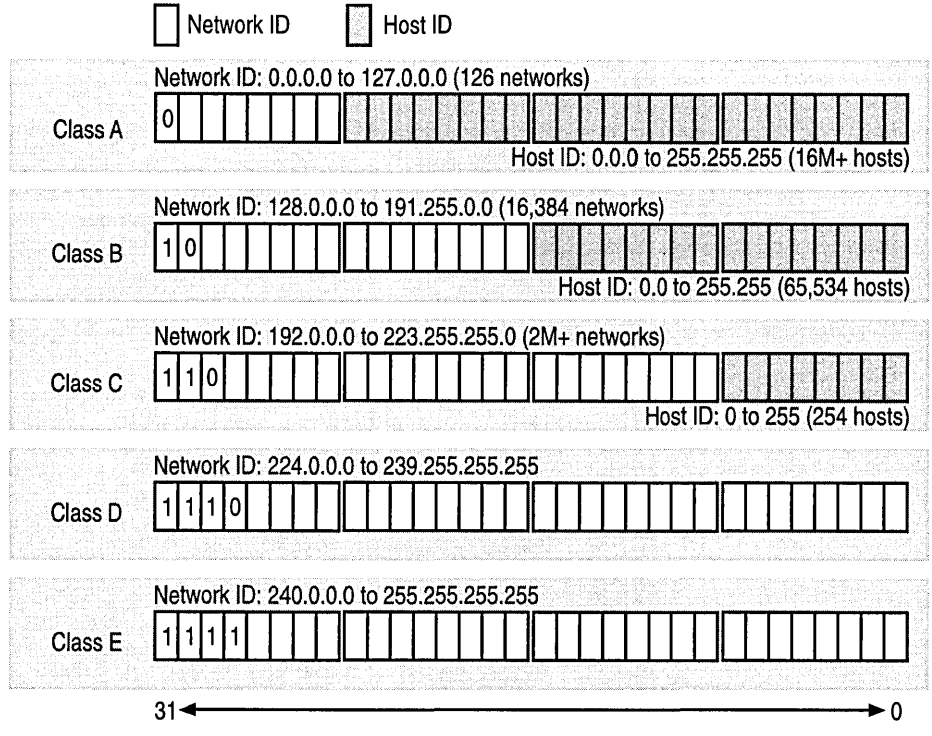
Note Windows CE supports one default gateway.

Each IP address defines a network identifier and a host identifier. The network identifier identifies systems located on the same physical network. All systems on the same physical network must have the same network identifier. The *host identifier* identifies a workstation, server, router, or other TCP/IP host within a network. The address for each host must be unique to the network identifier.

Each IP address is 32 bits long and composed of four *octets*, or 8-bit fields. An octet is a decimal number in the range from 0 through 255. Each octet is separated by a period. This format is called *dotted decimal notation*. The IP address 224.0.1.24 is an example of dotted decimal notation.

Internet Protocol Address Classes

The Internet community has defined IP address classes: A, B, C, D, and E. To accommodate varying network sizes, each address class handles network addressing for a network of a unique size. The class defines the possible number of networks and the number of hosts for each network. Each class also defines which bits of the IP address are used for the network identifier and which bits are used for the host identifier. The following illustration shows the IP address configurations for the five IP address classes.



The following table shows address classes supported by Windows CE.

Address class	Description
Class A	Assigned to networks with a large number of hosts. The high-order bit in a class A address is always set to 0. The next seven bits, completing the first octet, complete the network identifier. The remaining 24 bits—the last three octets—represent the host identifier. This accommodates 126 networks, 128 minus two reserved addresses, and over 16 million hosts for each network.
Class B	Assigned to networks with a medium to large number of hosts. The two high-order bits in a class B address are always set to 10. The next 14 bits—completing the first two octets—complete the network identifier. The remaining 16 bits—the last two octets—represent the host identifier. This accommodates 16,384 networks and more than 65,000 hosts for each network.
Class C	Assigned to networks with a small number of hosts, specifically, local area networks (LANs). The three high-order bits in a class C address are always set to 110. The next 21 bits—completing the first three octets—complete the network identifier. The remaining 8 bits—the last octet—represent the host identifier. This accommodates more than 2 million networks and 254 hosts for each network.
Class D	Used for multicasting to a number of hosts. Packets are passed to a selected subset of hosts on a network. Only those hosts registered for the multicast address accept the packet. The four high-order bits in a class D address are always set to 1110. The remaining bits are for the address that registered hosts will recognize. Windows CE supports class D addresses for applications to multicast data to hosts.
Class E	Reserved for future use. An experimental address. High-order bits in a class E address are set to 1111.

Host Name Resolution

Windows CE uses the *Domain Name System* (DNS) and the *Windows Internet Naming Service* (WINS) for host name resolution. DNS is a naming service that resolves system names to current IP addresses and uses a hierarchical model to pass name resolutions between domains. DNS enables a TCP/IP host to find the IP address of another host using only the host name. WINS provides a distributed database for registering and querying dynamic name-to-IP address mappings in a routed network environment. Also, a host name can be found using an IP address. A Windows CE–based application can access DNS and WINS using the Winsock functions **gethostbyname** and **gethostbyaddr**.

Configuring Dynamic Host Configuration Protocol

The *Dynamic Host Configuration Protocol* (DHCP) provides a framework for passing configuration data to hosts in a TCP/IP network, which eliminates the problems associated with the manual TCP/IP configuration. When a DHCP server receives a request, it automatically assigns an IP address from a pool of addresses, as well as the *address mask*, the default gateway, the DNS server, the domain name, and the WINS server.

The options to be sent are read from the registry under **HKEY_LOCAL_MACHINE\Comm*AdapterName*\Parms\Tcpip\ DhcpOptions**. You must create values under this key that have the names of the options to send. The default subkey under **DhcpOptions** must also be set; otherwise, DHCP internal default options are sent.

For example, creating the value names 1, 3, and 5 under the **DhcpOptions** subkey causes DHCP to query for these options: address mask, domain name, and router. The results that DHCP receives are stored as binary values under these value names. The data format is the same as the DHCP format.

Configuring TCP/IP for Wireless Networks

Because TCP/IP stacks are designed to work efficiently on wired networks, performance can degrade on wireless networks. For example, settings appropriate to a 10 Mbps Ethernet connection may consume excessive bandwidth by generating unnecessary retransmission requests.

To use wireless networking efficiently, you may need to set some TCP/IP parameters to the characteristics of the supporting network. Because network parameters are maintained on a per-adapter basis, your application must determine the appropriate adapter and query the user for associated registry settings.

The following table shows the parameters you may need to modify.

Registry parameter	Description
TcpWindowSize	TCP receive window size registry key: **HKEY_LOCAL_MACHINE\Comm*AdapterName*\Tcpip\Parms\TcpWindowSize**. In general, larger receive windows work better with high-delay, high-bandwidth networks. For greatest efficiency, the receive window should be an even multiple of the TCP maximum segment size. It should not exceed the system maximum value in **HKLM\Comm*AdapterName*\Tcpip\Parms\GlobalMaxTcpWindowSize**.
TcpInitialRTT	Initial round-trip time (RTT) registry key: **HKEY_LOCAL_MACHINE\Comm*AdapterName*\Tcpip\Parms\TcpInitialRTT**. The key value sets the initial RTT in milliseconds. The default value is 3000. The initial RTT is generally greater for wireless networks than for wired networks.
TcpDelAckTicks	Delayed acknowledgment timer registry key: **HKEY_LOCAL_MACHINE\Comm*AdapterName*\Tcpip\Parms\TcpDelAckTicks**. The default value is 200 milliseconds.

Resolving Device Suspension Issues

Connected TCP sockets, including loop back, prevent a device from suspending. The following operations enable a device to suspend:

- Any operations on UDP sockets.
- Unconnected open sockets.
- Listening sockets; after being accepted, they will keep the device from suspending.

Developing a Winsock Application

A socket enables network applications access to data on a data network. A computing device may have only one physical connection to a network, but many sockets may use the one physical connection simultaneously.

Winsock client and server applications provide endpoints of communication for network applications. A server application executes, then waits to receive a packet from the client application. Once communication is established, client and server applications can exchange data. A server application can handle multiple clients simultaneously.

Note Winsock client and server applications must be of the same socket type to communicate. They must both be using byte stream sockets that use TCP, or they must both be using unreliable datagram sockets that use UDP.

Winsock Functions

Windows CE supports the standard Winsock 1.1 functions, except the asynchronous functions. Some asynchronous notification support is available through the Microsoft Foundation Classes (MFC) **CCeSocket** class. For more information on the MFC for Windows CE, see the Windows CE Toolkit for Visual C++® 6.0.

Winsock functions are defined in the Winsock.h header file. The following table shows Winsock functions implemented for Windows CE.

Function	Description
accept	Accepts a socket connection
bind	Associates a local address with a socket
closesocket	Closes a socket
connect	Establishes a connection to a peer
gethostbyaddr	Retrieves host data that corresponds to a network address
gethostbyname	Retrieves host data that corresponds to a host name
gethostname	Returns the standard host name for the local computer
getpeername	Retrieves the address of the peer, to which a socket is connected
getsockname	Retrieves the local address for a socket
getsockopt	Retrieves a socket option
htonl	Converts a u_long from host byte order to network byte order
htons	Converts a u_short from host byte order to network byte order
inet_addr	Converts a string containing a dotted address into a network address in the format of an **IN_ADDR** structure
inet_ntoa	Converts a network address into a string in dotted format
ioctlsocket	Controls the socket mode
listen	Establishes a socket to listen for incoming connections
ntohl	Converts a u_long from network byte order to host byte order
ntohs	Converts a u_short from network byte order to host byte order
recv	Receives data from a socket

Function	Description
recvfrom	Receives a datagram and stores the source address
select	Determines the status of one or more sockets
send	Sends data on a connected socket
sendto	Sends data to a specific destination
setsockopt	Sets a socket option
shutdown	Disables send or receive operations on a socket
socket	Creates a socket
WSACleanup	Initiates no action and is provided only for compatibility
WSAGetLastError	Retrieves the error status for the last Winsock operation that failed
WSAIoctl	Controls the mode of a socket
WSASetLastError	Sets the error value retrieved by **WSAGetLastError**
WSAStartup	Initializes a **WSADATA** structure

Winsock Structures

Winsock structures are defined in the Winsock.h header file. The following table shows Winsock structures implemented for Windows CE.

Structure	Description
IN_ADDR	The IP address component of the **SOCKADDR_IN** structure in big-endian network byte order.
LINGER	Used by an application to set the linger socket option and specify the length of time to wait for unsent data before a socket is closed.
SOCKADDR	The generic address structure for all address families used in Winsock communications.
SOCKADDR_IN	In the Internet address family, this structure is used by Winsock to specify a local or remote endpoint address, to which to connect a socket. This is the form of the **SOCKADDR** structure specific to the Internet address family and can be cast to **SOCKADDR**.
SOCKADDR_IRDA	Specifies an Infrared Sockets address.
WSADATA	Used to store Winsock initialization data returned by a call to **WSAStartup**. It contains data about the Winsock.dll implementation.

Using Winsock Functions with IrDA

Some Winsock functions work differently with IrDA than with TCP/IP. The principal differences are described in this section.

IrSock Name Service

Conventional Winsock name service is best suited to fixed networks, in which the group of devices that can accept a socket connection is relatively static. Conversely, IrDA is designed to handle browsing for resources within range. It works in an extemporary manner, and devices disconnect and connect frequently as they move in and out of range.

Because of these differences, IrSock does not use the conventional Winsock name service functions. Instead, name service is incorporated into the communication stream.

IrSock Addressing

Addressing is based on *Logical Service Access Point Selectors* (LSAP-SELs), numbered from 1 through 127. Because of the small range of values available, it is usually better not to bind sockets directly to an LSAP-SEL. Instead, the *Information Access Service* (IAS) provides a means for dynamic binding of sockets to LSAP-SELs.

To use IAS, a server application binds a socket to an IAS service name. The client application uses the service name when using the **connect** function. Neither application must be notified of the LSAP-SEL assigned by the IAS.

IrSock Enhanced Socket Options

Windows CE includes socket options to access the unique features of the IrDA protocol, IRLMP_IAS_SET, IRMP_IRLPT_MODE, and IRLMP_EXCLUSIVE_MODE.

IRLMP_IAS_SET enables an application to set a single class in the local IAS. The application specifies the class to set, the attribute, and the attribute type. The application must allocate a buffer of the necessary size for the passed parameters. For more information on the using the IrDA protocols with Windows Sockets, see Creating an Infrared Winsock Application.

Using WSAStartup to Initialize Winsock

Calling the **WSAStartup** function initializes the Winsock.dll and a **WSADATA** structure that contains the details of the Winsock implementation. When an application or DLL has finished using the Winsock.dll, it must call **WSACleanup** to enable the Winsock.dll to free any resources for the application. For every call to **WSAStartup,** there must be a call to **WSACleanup**.

The following code example shows how to use **WSAStartup**:

```
if (WSAStartup (MAKEWORD(1,1), &WSAData) != 0)
{
  MessageBox (NULL, TEXT("WSAStartup failed!"), TEXT("Error"), MB_OK);
  return FALSE;
}
```

If successful, **WSAStartup** returns 0. After **WSAStartup** returns, an application cannot call **WSAGetLastError** to determine the error value as is normally done with Winsock functions.

The **WSADATA** structure pointed to by *lpWSAData* stores Winsock initialization data returned by a call to **WSAStartup. WSADATA** contains Winsock.dll implementation data. An application or DLL can call **WSAStartup** repeatedly if it needs to obtain the **WSADATA** structure data more than once.

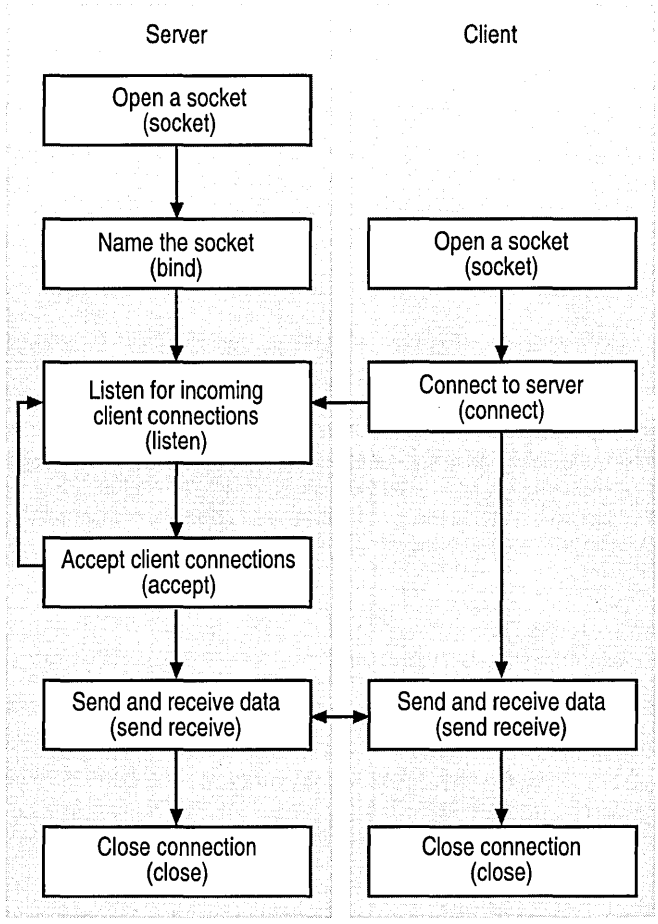
The following table shows values that **WSAStartup** assigns to the members of **WSADATA.**

WSADATA member	Assigned value
wVersion	1.1
wHighVersion	1.1
szDescription	NULL string
szSystemStatus	NULL string
iMaxSockets	20
iMaxUdpDg	0
lpVendorInfo	NULL

Creating a TCP Stream Socket Application

Use TCP to provide sequenced, reliable two-way connection-based byte streams. A TCP stream socket server application listens on the network for incoming client request packets. A TCP stream socket client application initiates communication with the server by sending a request packet. When the server receives the request, it processes it and responds. After this initial sequenced message exchange, client and server can exchange data.

The following illustration shows the interaction between the TCP stream socket server and TCP stream socket client.



▶ To create a TCP stream socket server application

1. Open a stream socket with the **socket** function.

   Use AF_INET for the *address format* parameter and SOCK_STREAM for the *type* parameter.

   The following code example shows how to open a socket.

```
if ((WinSocket = socket (AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
  wsprintf (szError, TEXT("Allocating socket failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  return FALSE;
}
```

2. Name the socket with the **bind** function, using a **SOCKADDR_IN** structure for the *address* parameter.

   When you open a socket with **socket**, the socket has no name assigned to it. However, a descriptor for the socket is allocated in an address family name space. To assign a name to the server socket, call **bind**. To be identified by the client socket, a TCP stream socket server application must name its socket. However, it is unnecessary to give a client socket a name. using **bind**.

   The **bind** function establishes the local socket association by assigning a local socket name to an unnamed socket. A socket name consists of three address fields when in the TCP/IP address family, known as the Internet address family: the address family protocol, a host address, and a port number that identifies the application. These address fields, *sin_family sin_addr sin_port* are members of the **SOCKADDR_IN** structure. You must initialize **SOCKADDR_IN** before calling **bind**.

   The following code example shows how to initialize **SOCKADDR_IN,** and then use **bind**.

```
// Fill out the local socket address data.
local_sin.sin_family = AF_INET;
local_sin.sin_port = htons (PORTNUM);
local_sin.sin_addr.s_addr = htonl (INADDR_ANY);
```

```
// Associate the local address with WinSocket.
if (bind (WinSocket,
          (struct sockaddr *) &local_sin,
          sizeof (local_sin)) == SOCKET_ERROR)
{
  wsprintf (szError, TEXT("Binding socket failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (WinSocket);
  return FALSE;
}
```

3. Listen for incoming client connections with the **listen** function.

   To prepare for a name association the TCP stream server must first listen for connection requests from the TCP client with **listen**.

   The following code example shows how to use **listen**.

```
if (listen (WinSocket, MAX_PENDING_CONNECTS) == SOCKET_ERROR)
{
  wsprintf (szError,
            TEXT("Listening to the client failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (WinSocket);
  return FALSE;
}
```

4. Accept a client connection with the **accept** function.

   The TCP stream server socket uses **accept** to accept the client connection that completes the name association between client and server.

   The **accept** function creates a new socket. The original socket opened by the server continues to listen and can be used to accept more connections until closed. Server applications must close the listening socket, in addition to any sockets created, by accepting a client connection.

The following code example shows how to use **accept**.

```
accept_sin_len = sizeof (accept_sin);

// Accept an incoming connection attempt on WinSocket.
ClientSock = accept (WinSocket,
                     (struct sockaddr *) &accept_sin,
                     (int *) &accept_sin_len);

// Stop listening for connections from clients.
closesocket (WinSocket);

if (ClientSock == INVALID_SOCKET)
{
  wsprintf (szError, TEXT("Accepting connection with client
            failed.") TEXT(" Error: %d"), WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  return FALSE;
}
```

5. Send and receive data with a client using the **send** and **recv** functions.

   Once client and server sockets are connected, you can use the **send** and **recv** functions to exchange data.

   The **send** function writes outgoing data on a connected socket. The **recv** function reads incoming data on connection-oriented sockets. The **recv** function can also be used with connectionless sockets. When using a TCP connection-oriented stream, a socket must be connected before calling **recv**.

   The following code shows an example of using the **send** and **recv** functions.

```
for (;;)
{
  // Receive data from the client.
  iReturn = recv (ClientSock, szServerA, sizeof (szServerA), 0);

  // Verify that data was received. If yes, display it.
  if (iReturn == SOCKET_ERROR)
  {
    wsprintf (szError, TEXT("No data is received, receive failed.")
              TEXT(" Error: %d"), WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Server"), MB_OK);
    break;
  }
  else if (iReturn == 0)
  {
    MessageBox (NULL, TEXT("Finished receiving data"),
                TEXT("Server"), MB_OK);
    break;
```

```
    }
    else
    {
      // Convert the ASCII string to Unicode.
      for (index = 0; index <= sizeof (szServerA); index++)
        szServerW[index] = szServerA[index];

      // Display the string received from the client.
      MessageBox (NULL, szServerW, TEXT("Received From Client"),
                  MB_OK);
    }
  }

  // Send a string from the server to the client.
  if (send (ClientSock, "To Client.", strlen ("To Client.") + 1, 0)
          == SOCKET_ERROR)
  {
    wsprintf (szError,
              TEXT("Sending data to the client failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  }
```

Successfully completing a call to **send** does not confirm that data was successfully delivered.

6. Close the connection with the **closesocket** function.

   When data exchange between the server and client ends, close the socket with **closesocket**. To ensure that all data is exchanged on a TCP connection, an application should call **shutdown** before calling **closesocket**.

   An application should always have a matching call to **closesocket** for each successful call to **socket** to return any socket resources to the system. For TCP stream sockets, when a socket connection ends the server closes the socket created by **accept**. The server does not close the socket originally returned by the first call to **socket**. That socket continues to listen for clients. When the server disconnects or is out of service it should call **closesocket** to close the listening socket.

For an example of a TCP stream socket server application, see TCP Stream Socket Server.

▶ To create a TCP stream socket client application

1. Open a stream socket with **socket**.

   Use AF_INET for the *address format* parameter and SOCK_STREAM for the *type* parameter.

2. Connect to the server with the **connect** function using a **SOCKADDR_IN** structure for the *name* parameter.

   The TCP stream client associates socket names by connecting to the stream server with **connect**.

   The **SOCKADDR_IN** structure must be initialized before calling **connect**. This is similar to using **bind**, but *sin_port* and *sin_addr* are initialized with the remote socket name, not the local socket name used for **bind**.

   The following code example shows the TCP client connecting with the server.

```
// Establish a connection to the server socket.
if (connect (ServerSock,
             (PSOCKADDR) &destination_sin,
             sizeof (destination_sin)) == SOCKET_ERROR)
{
  wsprintf (szError,
            TEXT("Connecting to the server failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (ServerSock);
  return FALSE;
}
```

3. Exchange data with a server, using the **send** and **recv** functions.
4. Close the connection with the **closesocket**.

For an example of a TCP stream socket client application, see TCP Stream Socket Client.

Creating an Infrared Winsock Application

The basic procedure for using IrSock is similar to that for Winsock. IrSock uses only the TCP stream socket connection-oriented communication method. Server applications and client applications have different procedures.

Note You must include the Af_irda.h header file in your application to access IrSock features in the Winsock functions.

▶ **To create and use a socket with a server application**

1. Open a stream socket with **socket**. Use AF_IRDA for the address format parameter, **SOCK_STREAM** for the *type* and NULL for the *protocol* parameter.
2. Bind the service name to the socket with **bind**. Pass a **SOCKADDR_IRDA** structure for the *address* parameter.
3. Listen for an incoming client connection with **listen**.
4. Accept an incoming client with **accept**.
5. Use **send** and **recv** to communicate with the client.
6. Close the socket with the **closesocket** function.

For an example of an Infrared Sockets server application, see Infrared Sockets Server.

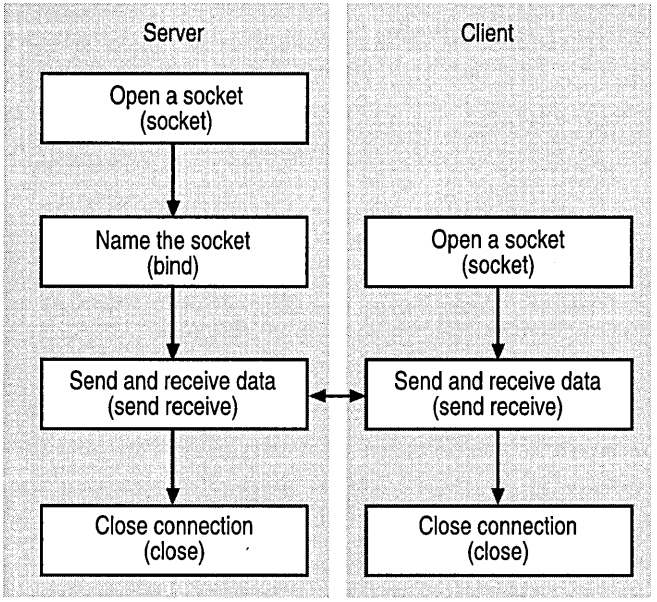
▶ **To create and use a socket with a client application**

1. Open a stream socket with **socket**, as with the server application. Use AF_IRDA for the address format parameter, **SOCK_STREAM** for the *type* and NULL for the *protocol* parameter
2. Search for the server, and retrieve its identifier with **getsockopt**.
3. Connect to the server with the **connect** function using **SOCKADDR_IRDA** for the *name* parameter.
4. Use **send** and **recv** to communicate with the server.
5. Close the socket with **closesocket**.

For an example of an Infrared Sockets client application, see Infrared Sockets Client.

Creating a UDP Datagram Socket Application

A datagram socket uses UDP, an unreliable connectionless protocol. A UDP server does not have to listen for and accept client connections, and a UDP client does not have to connect to a server. The following illustration shows the interaction between the UDP server and UDP client.



▶ **To create a UDP datagram socket server application**

1. Open a datagram socket with **socket**.

   Use AF_INET for the *address format* parameter and **SOCK_DGRAM** for the *type* parameter.

   To prepare for association with a client, a UDP datagram server need only create a socket and bind it to a name when preparing for association with a client.

2. Name the socket with the **bind** function, using a **SOCKADDR_IN** structure for the *address* parameter.

3. Exchange data with a client using the **sendto** and **recvfrom** functions.

   The UDP datagram socket server application calls **recvfrom** to prepare to receive data from a client The **recvfrom** function reads incoming data on unconnected sockets and captures the address from which the data was sent. To do this, the local address of the socket must be known.

   The **sendto** function is used on a connectionless socket to send a datagram to a specific peer socket identified by the *to* parameter. Successfully completing a **sendto** function call does not confirm data was successfully delivered.

4. Close the connection with the **closesocket** function.

   Calling the **shutdown** function is unnecessary for UDP sockets.

▶ **To create a UDP datagram socket client application**

1. Open a socket with the **socket** function.
2. Exchange data with server using **sendto** and **recvfrom**.

   A UDP datagram client socket is named when the client calls **sendto**.

3. Close the connection with the **closesocket** functions.

   Calling **shutdown** is unnecessary for UDP sockets.

Creating an IP Multicast Application

Although IP multicast traffic is sent to a single address, it is processed by multiple hosts. Collectively, the hosts listening to a specific IP multicast address are called a *multicast group*. With IP multicasting, only hosts that belong to a multicast group receive and process IP traffic sent to the group IP address. This process is similar to sending an e-mail message to an alias: only members of the alias receive the broadcast message. Multicasting is supported only on connectionless, UDP datagram sockets. For more information on membership in a multicast group, see Joining and Leaving a Multicast Group.

Other important aspects of IP multicasting include:

- Dynamic group membership. A host can join and leave the group at any time.
- A host can join a multicast group by sending an IGMP message.
- A group can be any size. It can have members spread out across multiple IP networks.
- A host can send IP traffic to a multicast group IP address without belonging to that group.

There are two types of messages used by IGMP:

1. Host membership report, which is used by a multicast router to poll a network for any members of a specified group.

   When a host joins a multicast group, it sends an IGMP message to the All Hosts IP multicast address, 224.0.0.1, declaring its membership in a specific host group.

2. Host membership query, which is used by a multicast router to poll a network for any members of a specified group.

   A router polls each network to verify members of a specific host group. If no hosts respond after the router makes several polls, it assumes no group members exist on that network. The router then stops propagating multicast traffic to that network and stops advertising to other routers data that it obtained previously about group members on that network.

The following section describes how to use Winsock functions in a Windows CE–based application to join a multicast group and receive IGMP support messages. It also describes how to send messages to a multicast address.

Mapping an IP Multicast Address

IP multicast addresses are mapped to a reserved set of Media Access Control multicast addresses and assigned from within the Class D address range from 224.0.0.0 through 239.255.255.255. A single IP address within the reserved range identifies each multicast group. Each multicast group IP address is shared by all host members of the group who listen and receive any IP messages sent to the group IP address.

The following table is a partial list of Class D addresses reserved for IP multicasting and registered with the Internet Assigned Numbers Authority (IANA).

Multicast group	IP multicast address	Description
Base address	224.0.0.0	Reserved
All hosts	224.0.0.1	Contains all systems on the same physical network
All routers	224.0.0.2	Contains all routers on the same physical network
Network Time Protocol	224.0.1.1	Distributes time-clock data used to synchronize time on a group of computers
RIP version 2	224.0.0.9	Distributes routing data between a group of routers that use a version 2 raster image processor (RIP)
WINS server	224.0.1.24	Supports autodiscovery and dynamic configuring of replication for WINS servers

Sending an IP Multicast Datagram

To send a multicast datagram, specify an IP multicast address with a range from 224.0.0.0 through 239.255.255.255 as the destination address in a **sendto** function call.

Use the **setsockopt** function to set options for IP multicasting.

By default, IP multicast datagrams are sent with a Time to Live (TTL) of 1, which prevents them from being forwarded beyond a single subnetwork. The following code example shows how to change this.

```
int ttl = 1 ; // Limits to subnet.
setsockopt(
            sock,
            IPPROTO_IP,
            IP_MULTICAST_TTL,
            (char *)&ttl,
            sizeof(ttl));
```

Multicast datagrams with a TTL of 0 are not transmitted on any subnetwork. Multicast datagrams with a TTL of greater than 1 may be delivered to more than one subnetwork, if there are one or more multicast routers attached to the first subnetwork.

A multicast router does not forward multicast datagrams with destination addresses from 224.0.0.0 through 224.0.0.255, inclusive, regardless of their TTL value. This address range is reserved for routing protocols and other low-level, topology discovery protocols, or maintenance protocols. These include gateway discovery protocols and group membership reporting protocols.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicasting-capable interface. The following code example shows how to use the **setsockopt** function to override the default for subsequent transmissions from a specified socket.

```
unsigned long addr = inet_addr("157.57.8.1");
setsockopt(
            sock,
            IPPROTO_IP,
            IP_MULTICAST_IF,
            (char *)&addr,
            sizeof(addr));
```

The *addr* parameter is the local IP address of the outgoing interface. An address of INADDR_ANY may be used to revert to the default interface. This address may be different from that which the socket is bound.

By default, if a multicast datagram is sent to a group to which the sending host belongs, a copy of the datagram on the outgoing interface is looped back by the IP layer for local delivery. Any attempt to disable this multicast loop-back results in the call failing, with the error message WSAENOPROTOOPT.

For an example of an application sending a multicast datagram, see Sending an IP Multicast Datagram Sample.

Joining and Leaving a Multicast Group

Before a host can receive IP multicast datagrams, it must become a member of one or more IP multicast groups.

The following code example shows how a process requests to join a multicast group by using the **setsockopt** function.

```
struct ip_mreq mreq;
// mreq is the ip_mreqstructure
{
  struct in_addr imr_multiaddr;  //The multicast group to join
  struct in_addr imr_interface;  //The interface to join on
}

#define RECV_IP_ADDR  "225.6.7.8" // An arbitrary multicast address

mreq.imr_multiaddr.s_addr = inet_addr(RECV_IP_ADDR);
mreq.imr_interface.s_addr = INADDR_ANY;
err = setsockopt(
                    sock,
                    IPPROTO_IP,
                    IP_ADD_MEMBERSHIP,
                    (char*)&mreq,
                    sizeof(mreq));
```

A socket must bind to an address before calling **setsockopt**.

Every multicast group membership is associated with a single interface. It is possible to join the same group on more than one interface. To choose the default multicast interface, use INADDR_ANY as the address for *imr_interface*. To choose a specific interface capable of multicasting, use one of the host's local addresses.

The following code example shows how to leave a multicast group.

```
struct ip_mreq mreq;
setsockopt(
            sock,
            IPPROTO_IP,
            IP_DROP_MEMBERSHIP,
            (char*)&mreq,
            sizeof(mreq));
```

The memberships associated with a socket are dropped when the socket is closed, or the process holding the socket is terminated. However, more than one socket may claim a membership in a particular group, and the host remains a member of that group until the last membership is dropped.

Receiving an IP Multicast Datagram

The members associated with a socket do not necessarily determine which datagrams are received by that socket. Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram. However, delivery of a multicast datagram to a particular socket is based on the destination port. To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified, that is, INADDR_ANY.

More than one process may bind to the same SOCK_DGRAM UDP port if the **bind** call is preceded by the following code.

```
int one = 1;
getsockopt(
            sock,
            SOL_SOCKET,
            SO_REUSEADDR,
            (char *)&one,
            sizeof(one));
```

In this case, every incoming multicast, or broadcast UDP datagram destined for the shared port is delivered to all sockets bound to the port.

The **getsockopt** function retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options can exist at multiple protocol levels, but they are always present at the uppermost socket level. Options affect socket operations. If the option was never set with **setsockopt**, then **getsockopt** returns the default value for the option. The definitions required for the multicast-related socket options are located in the Winsock.h file. All IP addresses are passed in network byte-order.

For an example of an application receiving a multicast datagram, see Receiving an IP Multicast Datagram Sample.

Reading Socket Options

The TTL of a multicast for a socket can be determined by reading the value from the socket options. The following code example shows how the TTL value is read.

```
int ttl;                          // Allocate space for TTL.
int sizeofttl = sizeof(ttl);      // Create an integer that contains the
                                  // size of the TTL value.
getsockopt(
            sock,
            IPPROTO_IP,
            IP_MULTICAST_TTL,
            (char *)&ttl,
            &sizeofttl);
```

The *ttl* parameter in the preceding code example contains the current TTL set value for the multicasts through a socket defined as sock.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicasting-capable interface. The following code example shows how a socket option is available to determine which interface is currently used for transmissions from a specified socket.

```
unsigned long addr;               //Allocate space for address
int sizeofaddr = sizeof(addr);    //Create an inter containing size of the
                                  // address.
getsockopt(
            sock,
            IPPROTO_IP,
            IP_MULTICAST_IF,
            (char *)&addr,
            &sizeofaddr );
```

The *addr* parameter contains the local IP address of the current outgoing interface after a **getsockopt** call.

By default, if a multicast datagram is sent to a group, to which the sending host belongs, a copy of the datagram on the outgoing interface is looped back by IP for local delivery. Any attempt to disable this multicast loop-back results in the call failing with the error message WSAENOPROTOOPT.

By default, if a multicast datagram is sent to a group, to which the sending host belongs, a copy of the datagram on the outgoing interface is looped back by IP for local delivery. This can be modified using socket options.

The following code example shows the option available to check the behavior status of the socket.

```
unsigned long status;                  //Allocate space for the status.
int sizeofstatus = sizeof(status);     //Create an integer.
                                       //Contains the size of the status
getsockopt(
            sock,
            IPPROTO_IP,
            IP_MULTICAST_LOOP,
            (char *)&status,
            &sizeofstatus );
```

The *status* parameter contains the status of the multicast loop-back after the **getsockopt** function call.

Using Secure Sockets

Windows CE supports the Private Communication Technology protocol 1.0 and SSL versions 2.0 and 3.0 security protocols. These protocols are available either through WinInet or directly from Winsock. Adding security to an application using these Winsock extensions requires few changes to an application. Once a secure socket is connected, the application may send and receive data on that socket unaware that the data over the wire is encoded.

Certificate Authentication

Authentication is the process of determining if a remote host can be trusted. To establish its trustworthiness, the remote host must provide an acceptable authentication certificate.

Remote hosts establish their trustworthiness by obtaining a certificate from a Certification Authority (CA). The CA may, in turn, have certification from a higher authority, and so on, creating a chain of trust. To determine whether a certificate is trustworthy, an application must determine the identity of the root CA, and then determine if it is trustworthy.

Windows CE maintains a database of trusted CAs. When a secure connection is attempted by an application, Windows CE extracts the root certificate from the certification chain and checks it against the CA database. It delivers the root certificate to the application through a certificate validation callback function, along with the results of the comparison against the CA database.

Applications bear ultimate responsibility for verifying that a certificate is acceptable. Applications can accept or reject any certificate. If a certificate is rejected, the connection is not completed. At a minimum, a certificate should meet two requirements: The certificate is current, and the identity contained in the certificate matches the root CA identity.

The following root certificate authorities are included in the Windows CE 2.1x Schannel Certificate Authority database:

- VeriSign/RSA Commercial
- VeriSign/RSA Secure Server
- VeriSign Class 2 Public Primary CA
- VeriSign Class 3 Public Primary CA
- VeriSign Class 4 Public Primary CA
- Keywitness Canada, Inc.
- GTE Cybertrust ROOT
- Thawte Server CA
- Thawte Premium Server CA
- Thawte Personal Basic CA
- Thawte Personal Freemail CA
- Thawte Personal Premium CA
- Microsoft Root Authority
- Root SGC Authority

▶ **To add certificates to the CA database through the registry:**

1. Create the key **HKLM\Comm\SecurityProviders\SCHANNEL\CAs** if one does not already exist.
2. Create a subkey with the name of the certificate, for example **My Certificate**.
3. Create the following values under the **My Certificate** key.

```
DWORD:Enabled = 1
DWORD:Type = 1
BINARY:CACert = X509 certificate bytes
```

Schannel will pick up the root certificate next time it is loaded.

The certificate validation callback function must be implemented by all client applications that use secure sockets. The value it returns determines if the connection will be completed by Winsock. It must have the following syntax:

```
int SslValidate (
        DWORD       dwType
        LPVOID      pvArg
        DWORD       dwChainLen
        LPBLOB      pCertChain
        DWORD       dwFlags
);
```

The parameters contain the following data:

- The *dwType* parameter specifies the data type pointed to by *pCertChain*. This must be SSL_CERT_X.509, specifying that *pCertChain* is a pointer to an X.509 style certificate.
- The *pvArg* parameter is the application-defined context, passed by the **SSLVALIDATECERTHOOK** structure.
- The *dwChainLen* parameter is the number of certificates pointed to by *pCertChain*. It will always be equal to one.
- The *pCertChain* parameter is a pointer to the root certificate. The BLOB struct is defined in Sslsock.h in the SDK. The **pBlobData** field points to a X.509 certificate (ISO standard). The certificate is not the root certificate but the server certificate. The caller must parse the certificate to extract the pertinent data like the subject and issuer names.
- If the root issuer of the certificate could not be found in the CA database, the *dwFlags* parameter will contain SSL_CERT_FLAG_ISSUER_UNKNOWN. The application can either attempt to verify the issuer itself, or return SSL_ERR_CERT_UNKNOWN.

The following table shows values returned by the callback function.

Return value	Description
SSL_ERR_BAD_DATA	The certificate is not properly formatted.
SSL_ERR_BAD_SIG	The signature check failed.
SSL_ERR_CERT_EXPIRED	The certificate has expired.
SSL_ERR_CERT_REVOKED	The certificate has been revoked.
SSL_ERR_CERT_UNKNOWN	The issuer is unknown, or some unspecified problem arose in the certificate processing, rendering it unacceptable.
SSL_ERR_OKAY	The certificate is acceptable.

Implementing a Secure Socket

The following procedure describes how to establish a secure socket connection.

▶ **To implement a secure socket**

1. Create a socket with the **socket** function.
2. Set the socket in secure mode with the **setsockopt** function. Set the *level* parameter to SO_SOCKET, *optname* to SO_SECURE, and *optval* to a **DWORD** set to SO_SEC_SSL.
3. Specify the certificate validation callback function by calling **WSAIoctl** with the SO_SSL_SET_VALIDATE_CERT_HOOK control code.
4. To specify a particular security protocol, call **WSAIoctl** with the SO_SSL_GET_PROTOCOLS control code to determine the default protocols. Then call **WSAIoctl** with the SO_SSL_SET_PROTOCOLS control code to select the protocols to be enabled. Otherwise, Windows CE selects the protocol.
5. Make a connection with the **connect** function.

   The certificate callback function is automatically called. The connection can be completed only if the callback function verifies the acceptability of the certificate by returning SSL_ERR_OKAY.
6. Transmit and send.

   The **send** and **recv** functions automatically encode and decode data.
7. When finished, close the socket with the **closesocket** function.

Using a Deferred Handshake

A deferred handshake enables an application to create an unsecured connection and then later convert it to a secure connection.

▶ **To implement secure sockets with a deferred handshake**

1. Create a socket with the **socket** function.
2. Set the socket in secure mode with **setsockopt**.

   The *level* parameter should be set to SO_SOCKET, *optname* should be set to SO_SECURE, and *optval* should be a **DWORD** set to SO_SEC_SSL.
3. Specify the certificate validation callback function by calling **WSAIoctl** with the SO_SSL_SET_VALIDATE_CERT_HOOK control code.
4. Set the socket in deferred handshake mode with **WSAIoctl**. The control code should be set to SO_SSL_SET_FLAGS and the flag to SSL_FLAG_DEFER_HANDSHAKE.

5. Establish a nonsecure connection with the remote party using **connect**.
6. Transmit and receive unencoded data.
7. To switch to secure mode, call **WSAIoctl** with the SO_SSL_PERFORM_HANDSHAKE control code passing in the target server name.

   The certificate callback function is automatically called. The handshake is successful only if the callback function verifies the acceptability of the certificate by returning SSL_ERR_OKAY.
8. Transmit and receive.

   The **send** and **recv** functions encode and decode the data automatically.
9. Close the socket with **closesocket** when finished.

Winsock Sample Applications

This section contains Winsock sample applications of both TCP stream type sockets and UDP datagram sockets. The UDP datagram socket applications show how to send and receive IP multicast datagrams.

TCP Stream Socket Server

This sample application shows how to implement a Winsock TCP stream socket server. It checks an incoming message sent by the client and sends a message to the client. This sample can be run on different Windows CE–based devices or the same device with Client.exe.

```
#include <windows.h>
#include <winsock.h>

#define PORTNUM               5000    // Port number
#define MAX_PENDING_CONNECTS  4       // Maximum length of the queue
                                      // of pending connections
int WINAPI WinMain (
              HINSTANCE hInstance,    // Handle to the current instance
              HINSTANCE hPrevInstance,// Handle to the previous instance
              LPTSTR lpCmdLine,       // Pointer to the command line
              int nCmdShow)           // Show state of the window
{
  int index = 0,                      // Integer index
      iReturn;                        // Return value of recv function
  char szServerA[100];                // ASCII string
  TCHAR szServerW[100];               // Unicode string
  TCHAR szError[100];                 // Error message string
```

```
SOCKET WinSocket = INVALID_SOCKET,    // Window socket
       ClientSock = INVALID_SOCKET; // Socket for communicating
                                    // between the server and client
  SOCKADDR_IN local_sin,            // Local socket address
              accept_sin;           // Receives the address of the
                                    // connecting entity
  int accept_sin_len;               // Length of accept_sin

  WSADATA WSAData;                  // Contains details of the Winsock
                                    // implementation

  // Initialize Winsock.
  if (WSAStartup (MAKEWORD(1,1), &WSAData) != 0)
  {
    wsprintf (szError, TEXT("WSAStartup failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    return FALSE;
  }

  // Create a TCP/IP socket, WinSocket.
  if ((WinSocket = socket (AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
  {
    wsprintf (szError, TEXT("Allocating socket failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    return FALSE;
  }

  // Fill out the local socket's address information.
  local_sin.sin_family = AF_INET;
  local_sin.sin_port = htons (PORTNUM);
  local_sin.sin_addr.s_addr = htonl (INADDR_ANY);

  // Associate the local address with WinSocket.
  if (bind (WinSocket,
            (struct sockaddr *) &local_sin,
            sizeof (local_sin)) == SOCKET_ERROR)
  {
    wsprintf (szError, TEXT("Binding socket failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    closesocket (WinSocket);
    return FALSE;
  }
```

```
// Establish a socket to listen for incoming connections.
if (listen (WinSocket, MAX_PENDING_CONNECTS) == SOCKET_ERROR)
{
  wsprintf (szError,
            TEXT("Listening to the client failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (WinSocket);
  return FALSE;
}

accept_sin_len = sizeof (accept_sin);

// Accept an incoming connection attempt on WinSocket.
ClientSock = accept (WinSocket,
                     (struct sockaddr *) &accept_sin,
                     (int *) &accept_sin_len);

// Stop listening for connections from clients.
closesocket (WinSocket);

if (ClientSock == INVALID_SOCKET)
{
  wsprintf (szError, TEXT("Accepting connection with client failed.")
            TEXT(" Error: %d"), WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  return FALSE;
}

for (;;)
{
  // Receive data from the client.
  iReturn = recv (ClientSock, szServerA, sizeof (szServerA), 0);

  // Check if there is any data received. If there is, display it.
  if (iReturn == SOCKET_ERROR)
  {
    wsprintf (szError, TEXT("No data is received, recv failed.")
              TEXT(" Error: %d"), WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Server"), MB_OK);
    break;
  }
  else if (iReturn == 0)
  {
    MessageBox (NULL, TEXT("Finished receiving data"), TEXT("Server"),
                MB_OK);
    break;
  }
```

```
    else
    {
      // Convert the ASCII string to a Unicode string.
      for (index = 0; index <= sizeof (szServerA); index++)
        szServerW[index] = szServerA[index];

      // Display the string received from the client.
      MessageBox (NULL, szServerW, TEXT("Received From Client"), MB_OK);
    }
  }

  // Send a string from the server to the client.
  if (send (ClientSock, "To Client.", strlen ("To Client.") + 1, 0)
          == SOCKET_ERROR)
  {
    wsprintf (szError,
              TEXT("Sending data to the client failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  }

  // Disable both sending and receiving on ClientSock.
  shutdown (ClientSock, 0x02);

  // Close ClientSock.
  closesocket (ClientSock);

  WSACleanup ();

  return TRUE;
}
```

TCP Stream Socket Client

This sample application shows how to implement a Winsock client. It sends a message to the server and checks the incoming message sent by the server. This sample can be run on the same device with the TCP Stream Socket Server sample application. In this case, define HOSTNAME as "localhost". Otherwise, define HOSTNAME as the full server name.

```
#include <windows.h>
#include <winsock.h>

#define PORTNUM         5000          // Port number
#define HOSTNAME        "localhost"   // Server name string
                                      // This should be changed
                                      // according to the server.
int WINAPI WinMain (
              HINSTANCE hInstance,    // Handle to the current instance
              HINSTANCE hPrevInstance,// Handle to the previous instance
              LPTSTR lpCmdLine,       // Pointer to the command line
              int nCmdShow)           // Show state of the window
{
  int index = 0,                      // Integer index
      iReturn;                        // Return value of recv function
  char szClientA[100];                // ASCII string
  TCHAR szClientW[100];               // Unicode string
  TCHAR szError[100];                 // Error message string

  SOCKET ServerSock = INVALID_SOCKET; // Socket bound to the server
  SOCKADDR_IN destination_sin;        // Server socket address
  PHOSTENT phostent = NULL;           // Points to the HOSTENT structure
                                      // of the server
  WSADATA WSAData;                    // Contains details of the Winsock
                                      // implementation

  // Initialize Winsock.
  if (WSAStartup (MAKEWORD(1,1), &WSAData) != 0)
  {
    wsprintf (szError, TEXT("WSAStartup failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    return FALSE;
  }
```

```
// Create a TCP/IP socket that is bound to the server.
if ((ServerSock = socket (AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
  wsprintf (szError, TEXT("Allocating socket failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  return FALSE;
}

// Fill out the server socket's address information.
destination_sin.sin_family = AF_INET;

// Retrieve the host information corresponding to the host name.
if ((phostent = gethostbyname (HOSTNAME)) == NULL)
{
  wsprintf (szError, TEXT("Unable to get the host name. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (ServerSock);
  return FALSE;
}

// Assign the socket IP address.
memcpy ((char FAR *)&(destination_sin.sin_addr),
        phostent->h_addr,
        phostent->h_length);

// Convert to network ordering.
destination_sin.sin_port = htons (PORTNUM);

// Establish a connection to the server socket.
if (connect (ServerSock,
             (PSOCKADDR) &destination_sin,
             sizeof (destination_sin)) == SOCKET_ERROR)
{
  wsprintf (szError,
            TEXT("Connecting to the server failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (ServerSock);
  return FALSE;
}

// Send a string to the server.
if (send (ServerSock, "To Server.", strlen ("To Server.") + 1, 0)
        == SOCKET_ERROR)
{
```

```
    wsprintf (szError,
              TEXT("Sending data to the server failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  }

  // Disable sending on ServerSock.
  shutdown (ServerSock, 0x01);

  for (;;)
  {
    // Receive data from the server socket.
    iReturn = recv (ServerSock, szClientA, sizeof (szClientA), 0);

    // Check if there is any data received. If there is, display it.
    if (iReturn == SOCKET_ERROR)
    {
      wsprintf (szError, TEXT("No data is received, recv failed.")
                TEXT(" Error: %d"), WSAGetLastError ());
      MessageBox (NULL, szError, TEXT("Client"), MB_OK);
      break;
    }
    else if (iReturn == 0)
    {
      MessageBox (NULL, TEXT("Finished receiving data"), TEXT("Client"),
                  MB_OK);
      break;
    }
    else
    {
      // Convert the ASCII string to a Unicode string.
      for (index = 0; index <= sizeof (szClientA); index++)
        szClientW[index] = szClientA[index];

      // Display the string received from the server.
      MessageBox (NULL, szClientW, TEXT("Received From Server"), MB_OK);
    }
  }

  // Disable receiving on ServerSock.
  shutdown (ServerSock, 0x00);

  // Close the socket.
  closesocket (ServerSock);

  WSACleanup ();

  return TRUE;
}
```

Infrared Sockets Server

In this sample an IrSock server allocates a socket and binds it to the IAS name, "ServerOne". It then allocates a single connection object and prepares the server to listen for incoming connections. When the client contacts the server, the server accepts the connection. It then receives a string from the client, passes one back, and closes the connection.

```
#include <windows.h>
#include <af_irda.h>

int WINAPI WinMain (
              HINSTANCE hInstance,     // Handle to the current instance
              HINSTANCE hPrevInstance,// Handle to the previous instance
              LPTSTR lpCmdLine,        // Pointer to the command line
              int nCmdShow)            // Show state of the window
{
  SOCKET ServerSock,                 // IR socket bound to the server
         ClientSock;                 // IR socket bound to the client

  SOCKADDR_IRDA address = {AF_IRDA, 0, 0, 0, 0, "IRServer"};
                                     // Specifies the server socket address
  int index = 0,                     // Integer index
      iReturn;                       // Return value of recv function
  char szServerA[100];               // ASCII string
  TCHAR szServerW[100];              // Unicode string
  TCHAR szError[100];                // Error message string


  // Create a socket bound to the server.
  if ((ServerSock = socket (AF_IRDA, SOCK_STREAM, 0)) == INVALID_SOCKET)
  {
    wsprintf (szError, TEXT("Allocating socket failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    return FALSE;
  }

  // Associate the server socket address with the server socket.
  if (bind (ServerSock, (struct sockaddr *)&address, sizeof (address))
          == SOCKET_ERROR)
  {
    wsprintf (szError, TEXT("Binding socket failed. Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    closesocket (ServerSock);
    return FALSE;
  }
```

```
// Establish a socket to listen for incoming connections.
if (listen (ServerSock, 5) == SOCKET_ERROR)
{
  wsprintf (szError,
            TEXT("Listening to the client failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (ServerSock);
  return FALSE;
}

// Accept a connection on the socket.
if ((ClientSock = accept (ServerSock, 0, 0)) == INVALID_SOCKET)
{
  wsprintf (szError, TEXT("Accepting connection with client failed.")
            TEXT(" Error: %d"), WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (ServerSock);
  return FALSE;
}

// Stop listening for connections from clients.
closesocket (ServerSock);

// Send a string from the server socket to the client socket.
if (send (ClientSock, "To Client!", strlen ("To Client!") + 1, 0)
        == SOCKET_ERROR)
{
  wsprintf (szError,
            TEXT("Sending data to the client failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
}

// Receive data from the client.
iReturn = recv (ClientSock, szServerA, sizeof (szServerA), 0);

// Check if there is any data received. If there is, display it.
if (iReturn == SOCKET_ERROR)
{
  wsprintf (szError, TEXT("No data is received, recv failed.")
            TEXT(" Error: %d"), WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Server"), MB_OK);
}
else if (iReturn == 0)
{
  MessageBox (NULL, TEXT("Finished receiving data"), TEXT("Server"),
              MB_OK);
}
```

```
  else
  {
    // Convert the ASCII string to a Unicode string.
    for (index = 0; index <= sizeof (szServerA); index++)
      szServerW[index] = szServerA[index];

    // Display the string received from the client.
    MessageBox (NULL, szServerW, TEXT("Received From Client"), MB_OK);
  }

  // Close the client and server sockets.
  closesocket (ClientSock);

  return 0;
}
```

Infrared Sockets Client

In this sample application, an IrSock client opens a socket and makes five attempts to locate a server. If no server is found, it displays a dialog box to inform the user of the failure. When a server is detected, the client queries the server for its device identifier and sends a greeting to the service named My Server. It then waits for the server to respond, displays a dialog box with the response, and closes the socket.

```
#include <windows.h>
#include <af_irda.h>

#define NUMRETYR  5                    // Maximum number of retries

int WINAPI WinMain (
              HINSTANCE hInstance,     // Handle to the current instance
              HINSTANCE hPrevInstance,// Handle to the previous instance
              LPTSTR lpCmdLine,        // Pointer to the command line
              int nCmdShow)            // Show window state.
{
```

```
SOCKET sock;                    // Socket bound to the server
DEVICELIST devList;             // Device list
SOCKADDR_IRDA address = {AF_IRDA, 0, 0, 0, 0, "IRServer"};
                                // Specifies the server socket address
int iCount = 0,                 // Number of retries
    index = 0,                  // Integer index
    iReturn,                    // Return value of recv function
    iDevListLen = sizeof (devList);
                                // Size of the device list
char szClientA[100];            // ASCII string
TCHAR szClientW[100];           // Unicode string
TCHAR szError[100];             // Error message string

// Create a socket that is bound to the server.
if ((sock = socket (AF_IRDA, SOCK_STREAM, 0)) == INVALID_SOCKET)
{
  wsprintf (szError, TEXT("Allocating socket failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  return FALSE;
}

// Initialize the number of devices to zero.
devList.numDevice = 0;

while ( (devList.numDevice == 0) && (iCount <= NUMRETYR))
{
  // Retrieve the socket option.
  if (getsockopt (sock, SOL_IRLMP, IRLMP_ENUMDEVICES,
                  (char *)&devList, &iDevListLen) == SOCKET_ERROR)
  {
    wsprintf (szError, TEXT("Server could not be located, getsockopt")
              TEXT(" failed. Error: %d"), WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    closesocket (sock);
    return FALSE;
  }

  iCount++;

  // Wait one second before retrying.
  Sleep (1000);
}
```

```
if (iCount > NUMRETYR)
{
  MessageBox (NULL, TEXT ("Server could not be located!"),
              TEXT ("Error"), MB_OK);
  closesocket (sock);
  return FALSE;
}

// Get the server socket address.
for (index = 0; index <= 3; index++)
{
  address.irdaDeviceID[index] = devList.Device[0].irdaDeviceID[index];
}

// Establish a connection to the socket.
if (connect (sock, (struct sockaddr *)&address,
             sizeof (SOCKADDR_IRDA)) == SOCKET_ERROR)
{
  wsprintf (szError,
            TEXT("Connecting to the server failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (sock);
  return FALSE;
}

// Send a string from the client socket to the server socket.
if (send (sock, "To Server.", strlen ("To Server.") + 1, 0)
        == SOCKET_ERROR)
{
  wsprintf (szError,
            TEXT("Sending data to the server failed. Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
}

// Receive data from the server socket.
iReturn = recv (sock, szClientA, sizeof (szClientA), 0);

// Check if there is any data received. If there is, display it.
if (iReturn == SOCKET_ERROR)
{
  wsprintf (szError, TEXT("No data is received, recv failed.")
            TEXT(" Error: %d"), WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Client"), MB_OK);
}
```

```
  else if (iReturn == 0)
  {
    MessageBox (NULL, TEXT("Finished receiving data"), TEXT("Client"),
                MB_OK);
  }
  else
  {
    // Convert the ASCII string to a Unicode string.
    for (index = 0; index <= sizeof (szClientA); index++)
      szClientW[index] = szClientA[index];

    // Display the string received from the server.
    MessageBox (NULL, szClientW, TEXT("Received From Server"), MB_OK);
  }

  // Close the socket.
  closesocket (sock);

  return 0;
}
```

Receiving an IP Multicast Datagram Sample

The following sample application shows how to receive an IP multicast datagram.

```
#include <windows.h>
#include <winsock.h>

#define RECV_IP_ADDR      "234.5.6.7"
#define DEST_PORT         4567

int WINAPI WinMain (
              HINSTANCE hInstance,     // Handle to the current instance
              HINSTANCE hPrevInstance,// Handle to the previous instance
              LPTSTR lpCmdLine,        // Pointer to the command line
              int nCmdShow)            // Show state of the window.
{
  int index = 0,                       // Integer index
      iRecvLen;                        // Length of recv_sin
  char szMessageA[100];                // ASCII string
  TCHAR szMessageW[100];               // Unicode string
  TCHAR szError[100];                  // Error message string

  SOCKET Sock = INVALID_SOCKET;        // Datagram window socket
```

```
struct ip_mreq mreq;                    // Used in adding or dropping
                                        // multicasting addresses
SOCKADDR_IN local_sin,                  // Local socket's address
            recv_sin;                   // Holds the source address upon
                                        // recvfrom function returns
WSADATA WSAData;                        // Contains details of the Winsock
                                        // implementation

// Initialize Winsock.
if (WSAStartup (MAKEWORD(1,1), &WSAData) != 0)
{
  wsprintf (szError, TEXT("WSAStartup failed! Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  return FALSE;
}

// Create a datagram socket, Sock.
if ((Sock = socket (AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
{
  wsprintf (szError, TEXT("Allocating socket failed! Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  return FALSE;
}

// Fill out the local socket's address information.
local_sin.sin_family = AF_INET;
local_sin.sin_port = htons (DEST_PORT);
local_sin.sin_addr.s_addr = htonl (INADDR_ANY);

// Associate the local address with Sock.
if (bind (Sock,
          (struct sockaddr FAR *) &local_sin,
          sizeof (local_sin)) == SOCKET_ERROR)
{
  wsprintf (szError, TEXT("Binding socket failed! Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (Sock);
  return FALSE;
}

// Join the multicast group from which to receive datagrams.
mreq.imr_multiaddr.s_addr = inet_addr (RECV_IP_ADDR);
mreq.imr_interface.s_addr = INADDR_ANY;
```

```
  if (setsockopt (Sock,
                  IPPROTO_IP,
                  IP_ADD_MEMBERSHIP,
                  (char FAR *)&mreq,
                  sizeof (mreq)) == SOCKET_ERROR)
  {
    wsprintf (szError, TEXT("setsockopt failed! Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    closesocket (Sock);
    return FALSE;
  }

  iRecvLen = sizeof (recv_sin);

  // Receive data from the multicasting group server.
  if (recvfrom (Sock,
                szMessageA,
                100,
                0,
                (struct sockaddr FAR *) &recv_sin,
                &iRecvLen) == SOCKET_ERROR)
  {
    wsprintf (szError, TEXT("recvfrom failed! Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    closesocket (Sock);
    return FALSE;
  }
  else
  {
    // Convert the ASCII string to a Unicode string.
    for (index = 0; index <= sizeof (szMessageA); index++)
      szMessageW[index] = szMessageA[index];

    MessageBox (NULL, szMessageW, TEXT("Info"), MB_OK);
  }

  // Disable receiving on Sock before closing it.
  shutdown (Sock, 0x00);

  // Close Sock.
  closesocket (Sock);

  WSACleanup ();

  return TRUE;
}
```

Sending an IP Multicast Datagram Sample

The following sample application shows how to send an IP multicast datagram.

```
#include <windows.h>
#include <winsock.h>

#define DEST_MCAST        "234.5.6.7"
#define DESTINATION_PORT  4567
#define SOURCE_PORT       0

int WINAPI WinMain (
              HINSTANCE hInstance,     // Handle to the current instance
              HINSTANCE hPrevInstance,// Handle to the previous instance
              LPTSTR lpCmdLine,        // Pointer to the command line
              int nCmdShow)            // Show window state.
{
  int iOptVal = 64;
  char szMessage[] = "Multicasting message!";
                                       // Sent message string
  TCHAR szError[100];                  // Error message string
  SOCKET Sock = INVALID_SOCKET;        // Datagram window socket

  SOCKADDR_IN source_sin,              // Source socket address
              dest_sin;                // Destination socket address

  WSADATA WSAData;                     // Contains details of the Winsock
                                       // implementation

  // Initialize Winsock Sockets.
  if (WSAStartup (MAKEWORD(1,1), &WSAData) != 0)
  {
    wsprintf (szError, TEXT("WSAStartup failed! Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    return FALSE;
  }

  // Create a datagram window socket, Sock.
  if ((Sock = socket (AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
  {
    wsprintf (szError, TEXT("Allocating socket failed! Error: %d"),
              WSAGetLastError ());
    MessageBox (NULL, szError, TEXT("Error"), MB_OK);
    return FALSE;
  }
```

```
// Fill out source socket's address information.
source_sin.sin_family = AF_INET;
source_sin.sin_port = htons (SOURCE_PORT);
source_sin.sin_addr.s_addr = htonl (INADDR_ANY);

// Associate the source socket's address with the socket, Sock.
if (bind (Sock,
          (struct sockaddr FAR *) &source_sin,
          sizeof (source_sin)) == SOCKET_ERROR)
{
  wsprintf (szError, TEXT("Binding socket failed! Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (Sock);
  return FALSE;
}

// Set the Time-to-Live of the multicast.
if (setsockopt (Sock,
                IPPROTO_IP,
                IP_MULTICAST_TTL,
                (char FAR *)&iOptVal,
                sizeof (int)) == SOCKET_ERROR)
{
  wsprintf (szError, TEXT("setsockopt failed! Error: %d"),
            WSAGetLastError ());
  MessageBox (NULL, szError, TEXT("Error"), MB_OK);
  closesocket (Sock);
  return FALSE;
}

// Fill out the desination socket's address information.
dest_sin.sin_family = AF_INET;
dest_sin.sin_port = htons (DESTINATION_PORT);
dest_sin.sin_addr.s_addr = inet_addr (DEST_MCAST);

// Send a message to the multicasting address.
if (sendto (Sock,
            szMessage,
            strlen (szMessage) + 1,
            0,
            (struct sockaddr FAR *) &dest_sin,
            sizeof (dest_sin)) == SOCKET_ERROR)
```

```
    {
      wsprintf (szError, TEXT("sendto failed! Error: %d"),
                WSAGetLastError ());
      MessageBox (NULL, szError, TEXT("Error"), MB_OK);
      closesocket (Sock);
      return FALSE;
    }
    else
      MessageBox (NULL, TEXT("Sending data succeeded!"), TEXT("Info"),
                  MB_OK);

    // Disable sending on Sock before closing it.
    shutdown (Sock, 0x01);

    // Close Sock.
    closesocket (Sock);

    WSACleanup ();

    return TRUE;
}
```

CHAPTER 6

Windows Networking

A Windows CE–based application can use Windows networking functions to establish and terminate network connections and to retrieve current configuration data for the Microsoft Network. Access to this data is made possible by way of the Windows CE networking API (WNet). WNet communicates through the *Common Internet File System* (CIFS) *redirector* to the remote host. A CIFS redirector is a module through which one computer can access another. An application can use WNet functions to manage network connections anywhere in the network hierarchy.

An application can access network resources using the *Universal Naming Convention* (UNC). UNC is a system for naming files on a network so that a file on a computer have the same path when accessed from any other computer. For example \\Servername\Sharename\Filename.ext; Servername is the server name, and Sharename is a directory on Servername that contains the file Filename.ext.

The Windows CE WNet API is similar to WNet for Windows-based desktop platforms with the following exceptions:

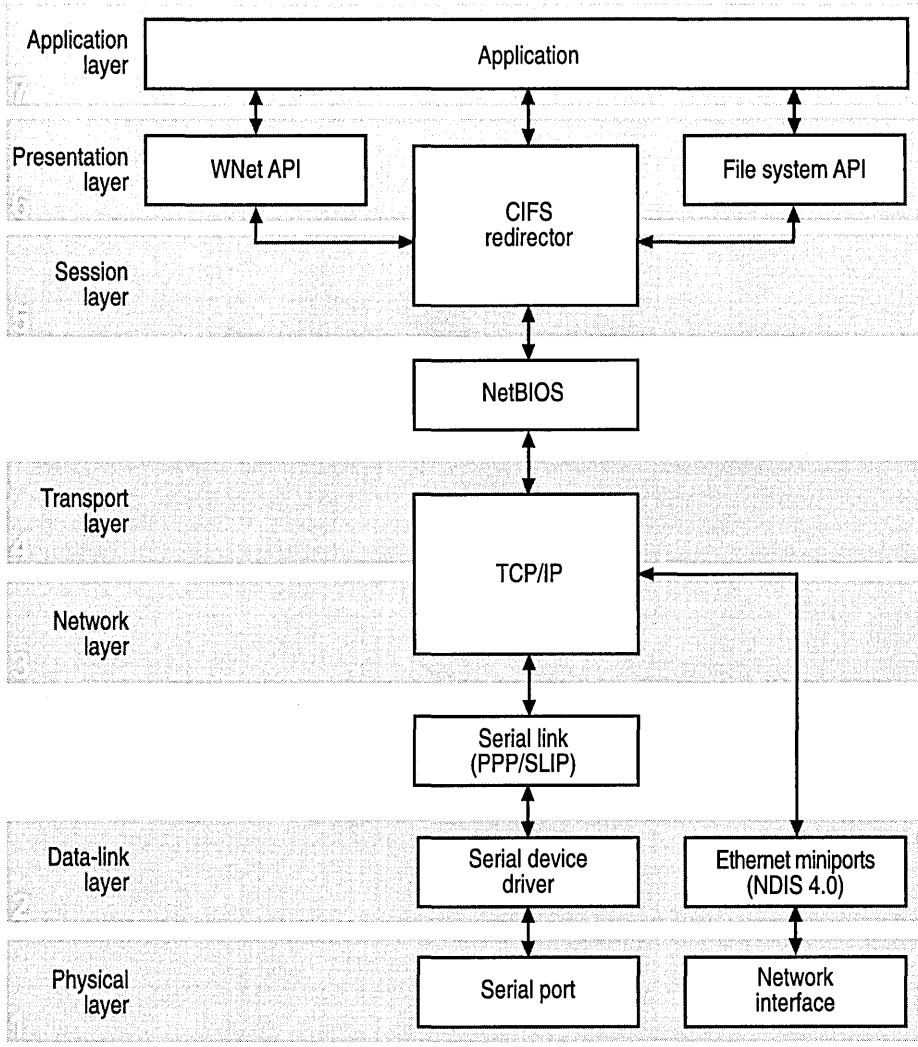
- Windows CE does not support drive letters. WNet supports mapping a remote UNC name to a local name, but whereas for the desktop operating systems the local name is drive-based, for example, H:<path>, Windows CE local names may take any form, for example, Myshare\pPath. Local names can be up to 64 characters in length, thus expanding the number of mapped network resources beyond 26.
- The only network provider currently supported by Windows CE is the Microsoft Windows Network.

- No connections are restored when the device is warm booted. A persistent connection is stored in the registry and the connection appears in the list of resources. This data is enumerated and retrieved by calling the **WNetOpenEnum** function with the *dwScope* parameter set to RESOURCE_REMEMBERED.
- Windows CE does not expose APIs for a mail slots or named pipes.
- Only a subset of the full WNet API set is supported by Windows CE. For example, the WNet function **WNetGetLastError** is not supported. This function is redundant because supported WNet functions do not return extended error data. The user can use the **GetLastError** function. All WNet functions return an ERROR_xxx value. This is not the same as Windows NT, which returns WN_*XXX* error value. ERROR _*XXX* error values are mapped to the appropriate WN_xxx error values for backward compatability.
- LAN Manager functions are not exposed.
- Windows CE does not support the concept of a computer or device belonging to a specific network context.

Windows Networking and the OSI Model

In the International Organization for Standardization Open Systems Interconnection (ISO/OSI) model for network communications, WNet functions operate across the presentation and session layers.

The following illustration shows how WNet functions fit into the ISO/OSI model for Windows CE communications.



Accessing Remote File Systems

Windows CE supports the CIFS redirector for accessing remote file systems and remote printers. The CIFS protocol is also referred to as the Server Message Block (SMB) protocol. The redirector's purpose is twofold: to reestablish a disrupted connection and to handle remote file system requests by packaging them and sending them to the target host for processing. The target host returns results to the computer making the request.

The Windows CE redirector supports connections to Windows-based desktop platforms, or servers compliant with the Windows NT LM 0.12 dialect of the CIFS specification. Applications access network drives either by way of WNet or a standard file system API with UNC paths.

To use the WNet functions under Windows CE, the redirector DLL, known as Redir.dll, and the NetBIOS DLL, known as Netbios.dll, must be installed in the Windows directory. If these DLLs are not installed, WNet functions return ERROR_NO_NETWORK.

Note The NetBIOS DLL contains only what is necessary to support the CIFS redirector. Windows CE does not support the NetBIOS interface.

Naming a Device

To use the redirector, a Windows CE–based device must have a unique name. The device name can be changed in the Control Panel communications properties, or preconfigured by setting the REG_SZ value under the **HKEY_LOCAL_MACHINE\Ident\Name** key. The name must include:

- From 1 to 15 characters.
- A leading character in the range 'a'-'z' or 'A'-'Z.'
- The remaining characters in the range 'a'-'z', 'A'-'Z', '0'-'9', or '-.'

Note Check with your network administrator to determine a valid name.

Control Panel does not check for invalid names, however the redirector will not function correctly if an invalid name is entered. This restriction also applies to other computer names accessed through the redirector, so servers whose names do not fit the name restrictions mentioned earlier may not be accessible to Windows CE.

This name is registered on the network the first time a file server is accessed over the network using either UNC file operations or WNet functions. The first file operation may take 10-15 seconds while the device name is registered. If the device name is not changed from the default, or if there is an error registering the name, a dialog box appears prompting the user to change the device name and retry the operation.

Once the name has been changed in the Control Panel, any active network connections must be reconnected before the new name is registered on the network. For example, if a network adapter PC Card is used, it must be removed and reinserted before the new name is registered. If an installed network adapter is used, the device must be warm booted for the new name to take effect.

Setting a User Name and Password

For dial-up connections, the default user name and password are those used to establish the connection. For network adapter PC Cards, default user logon data may be configured in the network Control Panel. If no default name and password are specified, or if an authentication error occurs while connecting to a server, the user logon dialog box appears. Enter your name and password in the dialog box and indicate whether to update the default values. If you choose to update the values, another dialog box asks whether to save the password in the registry. The password is stored in encrypted form; however, if you are concerned about having it stored, answer no to this dialog.

Modifying Registry Keys Used by the Redirector

The following table shows the modifiable registry settings for the **HKEY_LOCAL_MACHINE\Comm\Redir** key.

Registry key	Description	DWORD Values
ClearTxtPwdAllowed	Specifies if cleartext password is accepted	The default value is 0. If 0, refuse to connect to a server that negotiates clear text passwords.
ServerTimoutMs	Specifies the number of milliseconds to wait for SMB responses	The default value is 10000.
FindCacheMaxSize	Specifies the maximum memory to use for find caching	The default value is 8192. A value of 0 disables all caching.
ResourceExpiryInt	Specifies the number of seconds before releasing unused resources	The default value is 600.
RecvBufSize	Specifies the size of the receiving buffer	The default value is 4096.

Note If settings change, the Windows CE–based device must be warm booted before they take effect. Exercise caution when changing default values—this may affect system performance or the redirector operation.

Network Folder

Windows CE does not support drive letters; mounted file systems supply defined roots exposed in a top-level directory. For the redirector file system, the reserved root is \Network. To facilitate writing shells that are alert to network resources, the redirector maps paths in \network to locally connected resources.

A registry key was added to enable a project to expose the \Network folder. This is disabled by default, but can be overridden by setting the **DWORD** value **RegisterFSRoot** under the **HKEY_LOCAL_MACHINE\Comm\Redir** key to a nonzero value. If this value exists and is nonzero, the \Network folder appears in top-level enumerations, for example, dir *.*, on the device.

The \network folders can only be managed with the WNet API functions **FindFirstFile** and **FindNextFile**. Create folders in \Network with the **WNetAddConnection3** function and remove folders with the **WNetCancelConnection2** function.

Folders are added to and removed from \Network using the standard **WNetAddConnection** and **WNetCancelConnection** functions. Only persistent connections are enumerated in \Network, so if the device is warm booted, folders in \Network remain the same.

WNet Functions

The WNet functions are defined in the Winnetwk.h header file. The following table shows the WNet functions supported by Windows CE.

Function	Description
WNetAddConnection3	Makes a connection to a network resource and can specify a local name for the resource
WNetCancelConnection2	Terminates an existing network connection
WNetCloseEnum	Ends a network enumeration started by **WNetOpenEnum**
WNetConnectionDialog1	Displays a general browsing dialog box for connecting to a network

Function	Description
WNetDisconnectDialog	Displays a dialog box listing all currently connected resources and permits the user to select which resources to disconnect
WNetDisconnectDialog1)	Attempts to disconnect from a network; notifies the user of any errors
WNetEnumResource	Continues a network enumeration started by **WNetOpenEnum**
WNetGetConnection	Retrieves the remote name of a network resource associated with a local name
WNetGetUniversalName	Maps a local path for a network resource to a data structure containing the UNC based name
WNetGetUser	Retrieves user name used to establish a network connection
WNetOpenEnum	Starts an enumeration of network resources or existing connections

WNet Structures

WNet structures are defined in the Winnetwk.h header file. The following table shows the WNet structures supported by Windows CE.

Structure	Description
CONNECTDLGSTRUCT	Specifies browsing dialog box parameters for **WNetConnectionDialog1**.
DISCDLGSTRUCT	Used in the **WNetDisconnectDialog1** function to describe the required behavior for the disconnect attempt.
NETRESOURCE	Holds the network resource data. It is returned during enumeration of resources on the network and currently connected resources.
REMOTE_NAME_INFO	Contains path and name data about a network resource. The structure contains a member that points to a UNC name string for the resource and two members that point to additional network connection data strings.
UNIVERSAL_NAME_INFO	Contains a pointer to a UNC name string.

Managing Network Connections with WNet

Applications can use WNet to control network connections and retrieve network configuration data. Specifically, the WNet API provides functions for performing the following tasks:

- Determining available network resources
- Connecting to a network
- Retrieving network data
- Locating a printer on a network
- Printing on a network

Determining Available Network Resources

To determine the resources available on a network, an application passes the address of a **NETRESOURCE** structure to the **WNetOpenEnum** function. Calling **WNetOpenEnum** initializes an enumeration instance with the **NETRESOURCE** structure specifying the enumeration parameters.

▶ To identify available network resources

1. Set up the **NETRESOURCE** structure to define enumeration.

   To enumerate a *shared directory* on a server, change the *lpRemoteName* member in **NETRESOURCE** to the name of the server. Similarly, to enumerate servers in a domain, change the *lpRemoteName* to the name of the domain.

   The following code example shows how to enumerate a shared directory on a server.

```
NETRESOURCE nr;

nr.dwScope = RESOURCE_GLOBALNET;
nr.dwType = RESOURCETYPE_DISK;
nr.dwUsage = RESOURCEUSAGE_CONTAINER;
nr.lpLocalName = TEXT("");
nr.lpRemoteName = TEXT("\\\\MyServer");
nr.lpComment = TEXT("");
nr.lpProvider = TEXT("");

EnumerateFunc (hwnd, &nr);
```

The following code example shows how to enumerate servers in a domain.

```
NETRESOURCE nr;

nr.dwScope = RESOURCE_GLOBALNET;
nr.dwType = RESOURCETYPE_DISK;
nr.dwUsage = RESOURCEUSAGE_CONTAINER;
nr.lpLocalName = TEXT("");
nr.lpRemoteName = TEXT("MyDomain");
nr.lpComment = TEXT("");
nr.lpProvider = TEXT("");

EnumerateFunc (hwnd, &nr);
```

2. Call **WNetOpenEnum** to create an enumeration handle to the resource defined by **NETRESOURCE**.
3. Call **WNetEnumResource** with the enumeration handle created in the previous step to package the resource data in an array of **NETRESOURCE** structures.
4. Call **WNetCloseEnum** to close the enumeration handle.

   An application can continue enumerating any container's resources.

If the *dwUsage* member of **NETRESOURCE** returned by **WNetEnumResource** is RESOURCEUSAGE_CONTAINER, an application can continue enumerating that container by passing the address of that structure to **WNetOpenEnum**. If *dwUsage* is RESOURCEUSAGE_CONNECTABLE, an application can add a connection to the resource by passing the structure address to the **WNetAddConnection3** function. For more information, see Establishing a Network Connection.

The following code example shows how an application-defined function, **EnumerateFunc**, that enumerates all the resources in a subdirectory on a network. This function takes a pointer to a **NETRESOURCE** structure describing the subdirectory. Whenever **WNetEnumResource** returns a **NETRESOURCE** structure with the *dwUsage* member corresponding to RESOURCEUSAGE_CONTAINER, **EnumerateFunc** calls itself and passes a pointer to this structure in its call to **WNetOpenEnum**.

```
BOOL WINAPI EnumerateFunc (HWND hwnd, LPNETRESOURCE lpnr)
{
  HANDLE hEnum;

  DWORD dwIndex,
        dwResult,
        dwBufferSize = 16384,
        dwNumEntries = 0xFFFFFFFF;  // Enumerate all possible entries.

  LPNETRESOURCE lpnrLocal;          // Pointer to enumerated structures.

  dwResult = WNetOpenEnum (
                    RESOURCE_GLOBALNET, // All resources on the network
                    RESOURCETYPE_ANY,   // All resources
                    0,                  // Enumerate all resources.
                    lpnr,               // The container to enumerate
                    &hEnum);            // Handle to resource

  if (dwResult != ERROR_SUCCESS)
  {
    ErrorHandler (hwnd, dwResult, TEXT("WNetOpenEnum"));
    return FALSE;
  }

  // Allocate memory for NETRESOURCE structures.
  if (!(lpnrLocal = (LPNETRESOURCE) LocalAlloc (LPTR, dwBufferSize)))
    MessageBox (hwnd, TEXT("Not enough memory"), TEXT("Error"), MB_OK);

  do
  {
    dwResult = WNetEnumResource (
                    hEnum,              // Resource handle
                    &dwNumEntries,      // Number of entries
                    lpnrLocal,          // LPNETRESOURCE
                    &dwBufferSize);     // Buffer size

    if (dwResult == ERROR_SUCCESS)
    {
      for (dwIndex = 0; dwIndex < dwNumEntries; dwIndex++)
      {
        // Insert code here to perform operations with lpnrLocal;
        // for example, to display contents of NETRESOURCE structures.
        // ...
```

```
            // If this NETRESOURCE is a container, call the function
            // recursively.
            if (RESOURCEUSAGE_CONTAINER ==
                (lpnrLocal[dwIndex].dwUsage & RESOURCEUSAGE_CONTAINER))
            {
              if(!EnumerateFunc (hwnd, &lpnrLocal[dwIndex]))
                  MessageBox (hwnd, TEXT("EnumerateFunc returned FALSE."),
                              TEXT("Error"), MB_OK);
            }
          }
        }
        else if (dwResult != ERROR_NO_MORE_ITEMS)
        {
          ErrorHandler (hwnd, dwResult, TEXT("WNetEnumResource"));
          break;
        }
      } while (dwResult != ERROR_NO_MORE_ITEMS);

      LocalFree (lpnrLocal);

      dwResult = WNetCloseEnum (hEnum);

      if (dwResult != ERROR_SUCCESS)
      {
        ErrorHandler (hwnd, dwResult, TEXT("WNetCloseEnum"));
        return FALSE;
      }

      return TRUE;
    }
```

Connecting to a Network

Applications can use the WNet functions to establish and terminate network connections once they have identified available network resources.

Establishing a Network Connection

To establish a network connection, the user must know something about the network to be connected to. At a minimum, the user must know the local name or UNC of the network. For more information on functions you can use to retrieve network data, see Retrieving Network Data. An application can use one of two functions to establish network connections: **WNetAddConnection3** and **WNetConnectionDialog1**. Network connections are not automatically restored in Windows CE when a user logs on.

If a user knows the data needed to identify the network resource, an application can call **WNetAddConnection3**. The following code example shows how **WNetAddConnection3** is used to establish a connection with a network resource described by a **NETRESOURCE** structure.

```
DWORD dwResult;

// Make a connection to the network resource.
dwResult = WNetAddConnection3 (
                hwnd,                       // Handle to the owner window
                &nr,                        // Retrieved from enumeration
                NULL,                       // No password
                NULL,                       // Logged-in user
                CONNECT_UPDATE_PROFILE);    // Update profile with
                                            // connection data.
if (dwResult == ERROR_ALREADY_ASSIGNED)
{
  MessageBox (hwnd, TEXT("Already connected to specified resource."),
              TEXT("Info"), MB_OK);
  return FALSE;
}

else if (dwResult == ERROR_DEVICE_ALREADY_REMEMBERED)
{
  MessageBox (hwnd, TEXT("Attempted reassignment of remembered device"),
              TEXT("Info"), MB_OK);
  return FALSE;
}
else if (dwResult != ERROR_SUCCESS)
{
  ErrorHandler (hwnd, dwResult, TEXT("WNetAddConnection3"));
  return FALSE;
}

MessageBox (hwnd, TEXT("Connected to specified resource."),
            TEXT("Info"), MB_OK);
```

The **WNetConnectionDialog1** function creates a dialog box that enables a user to browse and connect to network resources. This function prompts the user to choose a local name or UNC in a dialog box.

The following code example shows how **WNetConnectionDialog1** is being called to create a dialog box that displays all disk resources on a network.

```
DWORD dwResult = WNetConnectionDialog1 (lpConnectDlgStruc);

if (dwResult != ERROR_SUCCESS)
{
  ErrorHandler (hwnd, dwResult, TEXT("WNetConnectionDialog1"));
  return FALSE;
}
```

Terminating a Network Connection

An application can use the following functions to terminate network connections: **WNetDisconnectDialog**, **WNetDisconnectDialog1**, or **WNetCancelConnection2**. The **WNetDisconnectDialog** displays a dialog listing all currently connected resources and permits user to select which resources to disconnect from. **WNetDisconnectDialog1** attempts to disconnect from a network and notifies the user of any errors presents. **WNetCancelConnection2** enables applications to disconnect from network resources and can be used to remove remembered network connections not currently in use. The following code example shows how **WNetCancelConnection2** is being used to disconnect from a network resource called MyDevice.

```
DWORD dwResult;

dwResult = WNetCancelConnection2 (
               TEXT("MyDevice"),       // Device name
               CONNECT_UPDATE_PROFILE, // Remove persistent connection.
               FALSE);

if (dwResult == ERROR_NOT_CONNECTED)
{
  MessageBox (hwnd, TEXT("MyDevice is not connected."),
              TEXT("Info"), MB_OK);
  return FALSE;
}
else if (dwResult != ERROR_SUCCESS)
{
  ErrorHandler (hwnd, dwResult, TEXT("WNetCancelConnection2"));
  return FALSE;
}

MessageBox (hwnd, TEXT("Connection closed for MyDevice"),
            TEXT("Info"), MB_OK);
```

Retrieving Network Data

WNet provides functions to retrieve network data in order to connect or disconnect from network resources. These functions perform the following tasks:

- Retrieving a connection name
- Retrieving a user name
- Retrieving network errors

Retrieving a Connection Name

WNet provides two functions for retrieving a connection name: **WNetGetUniversalName** and **WNetGetConnection**. **WNetGetUniversalName** takes a drive-based path for a network connection and obtains a more universal name in the form of a data structure.

WNetGetConnection enables an application to retrieve the name of a network resource associated with a local device. The following code example shows how **WNetGetConnection** retrieves the network name of a local device called MyDevice.

```
TCHAR szDeviceName[80];
DWORD dwResult,
      cchBuffer = sizeof (szDeviceName);

dwResult = WNetGetConnection (TEXT("MyDevice"), szDeviceName,
                              &cchBuffer);
switch (dwResult)
{
  case ERROR_SUCCESS:
    MessageBox (hwnd, szDeviceName, TEXT("Info"), MB_OK);
    break;

  case ERROR_NOT_CONNECTED:
    MessageBox (hwnd, TEXT("MyDevice is not connected."),
                TEXT("Info"), MB_OK);
    break;

  case ERROR_CONNECTION_UNAVAIL:
    // A connection is remembered, but not connected.
    MessageBox (hwnd, TEXT("Connection unavailable."),
                TEXT("Info"), MB_OK);
    break;

  default:
    ErrorHandler (hwnd, dwResult, TEXT("WNetGetConnection"));
    break;
}
```

Retrieving a User Name

An application can use the **WNetGetUser** function to retrieve the name of the user associated with a connected network resource. The following code example shows how **WNetGetUser** retrieves a user name based on a device name called MyDevice.

```
TCHAR szUserName[80];
DWORD dwResult,
      cchBuffer = 80;

dwResult = WNetGetUser (TEXT("MyDevice"), szUserName, &cchBuffer);

if (dwResult == ERROR_SUCCESS)
  MessageBox (hwnd, szUserName, TEXT("Info"), MB_OK);
else
{
  ErrorHandler (hwnd, dwResult, TEXT("WNetGetUser"));
  return FALSE;
}
```

Retrieving Network Errors

An application can call the **GetLastError** function to get extended data on an error, including the error value. Error data is usually provider-specific; however, the only provider supported by Windows CE is the Microsoft Windows Network provider.

The following code example shows a sample function, **ErrorHandler**, for application-defined error handling. This function calls **GetLastError** to get extended error data. **ErrorHandler** takes two parameters: a window handle and the name of the function that produced the error.

```
BOOL WINAPI ErrorHandler (HWND hwnd, DWORD dwError, LPTSTR lpszFunction)
{
  DWORD dwLastError;
  TCHAR szError[256],
        szCaption[256];

  // The following code performs extended error handling.
  dwLastError = GetLastError ();

  wsprintf (szError,
            TEXT("%s failed with error code %ld \n")
            TEXT("and extended error code %ld."),
            TEXT("Microsoft Windows Network"), dwError, dwLastError);
```

```
    wsprintf (szCaption, TEXT("%s error"), lpszFunction);

    MessageBox (hwnd, szError, szCaption, MB_OK);

    return TRUE;
}
```

Locating a Printer on a Network

Windows CE supports basic network printing and enables a user to select a network printer by UNC name. Specifying a printer by UNC requires a user to provide the network printer location in the format \\Server\Share.

Windows CE makes no attempt to retrieve data about the target printer or the type of output that it requires. When a network print job is queued, the redirector creates a thread to track the status of the print job. Windows CE does not support print queue manipulation. Also, some Windows-based desktop platform print servers do not support the Windows CE print job status completion notification. When using these print servers the user will not receive notification when a print job completes.

▶ **To find a printer on a network using WNet**

1. Create a network resource list.

   For more information on creating a network resource list, see Determining Available Network Resources.

2. To enumerate only the print resources, change the *dwType* parameter in **WNetOpenEnum** to RESOURCETYPE_PRINT.

Printing on a Network

Once you have the name of a printer, to print on a network use the **CopyFile** function or the **CreateFile** and **WriteFile** functions.

▶ **To print using CopyFile**

- Call **CopyFile**, specifying the network printer to use and which file to print.

  CopyFile has the following syntax:

  CopyFile (szSrcFile, szUNCPrinterShare, FALSE);

  The file to print is *szSrcFile*. The network printer to use is *szUNCPrinterShare*, which can accept a UNC name returned by the **WNetEnumResource** function.

▶ To print using CreateFile and Writefile

1. Create a file on the network printer by calling **CreateFile**.
2. Write the data or document to be printed to the newly created file by calling **WriteFile**.
3. Close the file to queue the print job.

CHAPTER 7

Internet Connections

The Windows CE Internet API (WinInet), is an API used for Internet client application development. The Wininet.dll module exports WinInet functions used to develop Internet applications like Web browsers and FTP applications.

WinInet is similar to WinInet for Windows-based desktop platforms with the following exceptions:

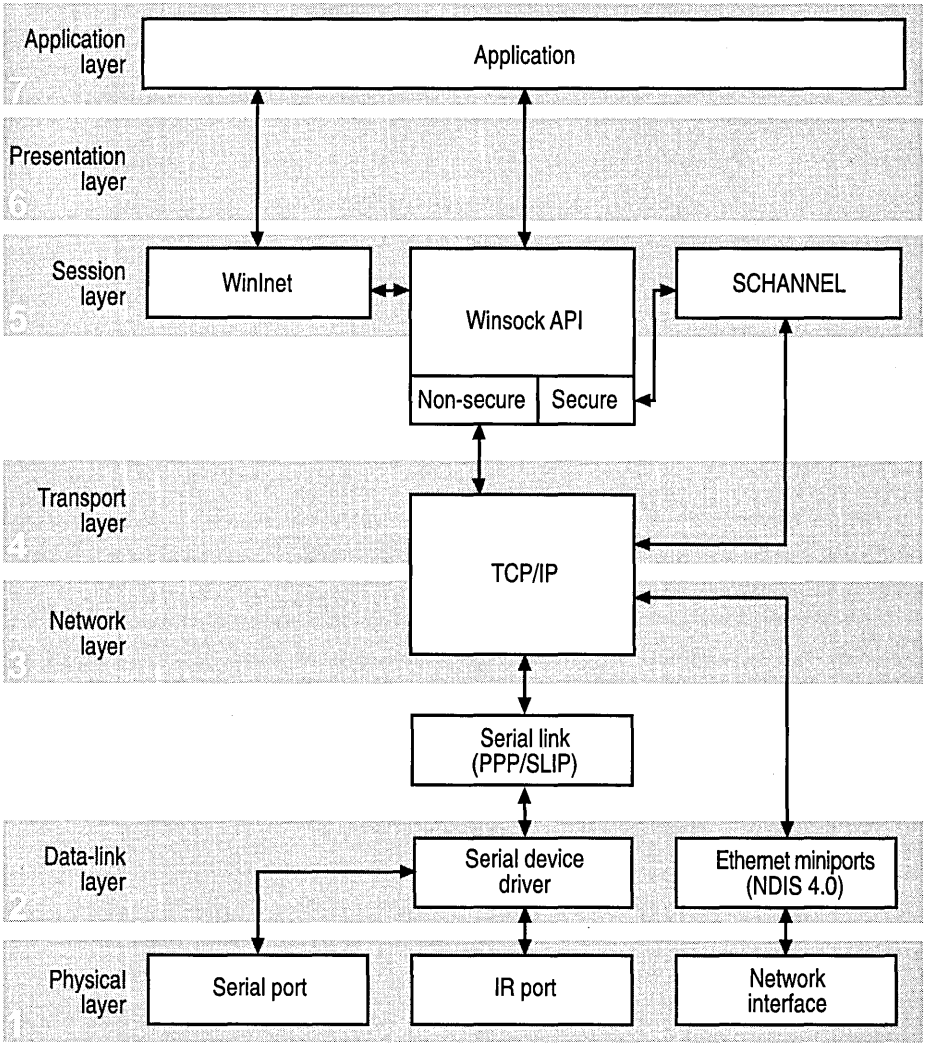
- Microsoft® ActiveX™ controls are not supported.
- The Gopher protocol and Gopher functions are not supported.

WinInet enables multithreaded applications to make concurrent calls to WinInet functions from different threads. WinInet functions synchronize if necessary, but do not validate parameters.

For more information about Windows Internet programming, see the Microsoft Developer Network Web site at http://msdn.microsoft.com/developer/.

WinInet and the OSI Model

In the International Organization for Standardization Open Systems Interconnection (ISO/OSI) model for network communications, WinInet operates at the session layer. WinInet handles programming Windows Sockets (Winsock), TCP/IP, and Internet protocols.



The Windows CE WinInet API uses the Hypertext Transfer Protocol (HTTP) version 1.1 and FTP Internet protocols. When navigating a Web site, the Internet browser uses HTTP to communicate with the Web server and read associated Web page files. Transmitting HTML is the primary purpose for HTTP, although HTTP can transmit any data format.

HTTP defines the format of a client request and a server response. A basic HTTP transaction consists of the following steps:

1. Client establishes a TCP/IP connection.
2. Client sends a request to the server.
3. Server sends a response to the client.
4. Client closes the TCP/IP connection.

FTP is used for sending and receiving files over a network.

Secure Hypertext Transfer Protocol (HTTPS) is a communication protocol designed to transfer encrypted data over the Internet. HTTPS is an HTTP extension using the *secure socket layer* (SSL). SSL is an encryption protocol invoked on a Web server that uses HTTPS.

WinInet Functions

WinInet functions and structures are defined in the WinInet.h header file. HTTP and FTP use several of the same WinInet functions to handle data. These common functions handle tasks consistently, regardless of the protocol they are applied to.

These common WinInet functions can be used to create general-purpose functions that handle tasks for FTP and HTTP. For example, these functions can perform the following HTTP and FTP tasks:

- Downloading files from another computer over the Internet can be handled by the **InternetReadFile**, **InternetFindNextFile**, and **InternetQueryDataAvailable** functions.
- Viewing and changing options are handled by the **InternetSetOption** and **InternetQueryOption** functions.
  - **InternetSetOption** accepts an unsigned long integer value that indicates the option to set, a buffer to hold the option setting, and the buffer length.
  - **InternetQueryOption** accepts an unsigned long integer value that indicates the option to query, a buffer to hold the option setting, and a pointer that contains the variable address containing the buffer length.
- The **InternetCloseHandle** function closes all HINTERNET handles. For more information about Internet handles, see HINTERNET Handles.

For applications that download multiple files or handle multiple tasks, waiting for each task to complete before moving on to the next task can be extremely inefficient. To avoid waiting, many of the Internet functions provide a way to perform tasks asynchronously. For information on using the Internet functions asynchronously, see the Microsoft Developer Network Web site at http://msdn.microsoft.com/developer/.

The following table shows WinInet functions supported by Windows CE.

Function	Description
InternetCanonicalizeUrl	Converts a Uniform Resource Locator (URL) to a canonical form, including conversion of unsafe characters into escape sequences.
InternetCloseHandle	Closes a single Internet handle or a subtree of Internet handles.
InternetCombineUrl	Combines a base and relative URL into a single canonicalized URL.
InternetConnect	Opens an FTP or HTTP session for a specified site.
InternetCrackUrl	Parses a URL into its component parts.
InternetCreateUrl	Creates a URL from its component parts.
InternetErrorDlg	Displays a dialog box for the error passed to the **InternetErrorDlg** function.
InternetFindNextFile	Continues a file search from a previous call to the **FtpFindFirstFile** function.
InternetGetCookie	Retrieves the cookie for the specified URL.
InternetGetLastResponseInfo	Retrieves the last Windows CE Internet function error description or server response on the thread calling this function.
InternetLockRequestFile	Enables the user to place a lock on the file being used.
InternetOpen	Initializes Windows CE Internet functions.
InternetOpenUrl	Begins reading a complete FTP or HTTP URL. For a relative URL and a base URL separated by blank spaces, **InternetCanonicalizeUrl** must be called first.
InternetQueryDataAvailable	Queries for the amount of data available.
InternetQueryOption	Queries for an Internet option on the specified handle.
InternetReadFile	Reads data from a handle opened by the **FtpOpenFile**, **FtpFindFirstFile**, or **HttpOpenRequest** functions.
InternetReadFileEx	Reads data from a handle opened by **InternetOpenUrl**, **FtpOpenFile**, **GopherOpenFile**, or **HttpOpenRequest**.
InternetSetCookie	Creates a cookie associated with the specified URL.
InternetSetFilePointer	Sets a file position for the **InternetReadFile** function.
InternetSetOption	Sets an Internet option.

Function	Description
InternetSetStatusCallback	Sets up a callback function that Windows CE Internet functions can use during an operation.
InternetTimeFromSystemTime	Formats the time and date.
InternetTimeToSystemTime	Converts a HTTP time and date string to a **SYSTEMTIME** structure.
InternetUnlockRequestFile	Unlocks a file that was locked using the **InternetLockRequestFile** function.
InternetWriteFile	Writes data to an open Internet file.

HTTP and FTP Functions

The following table shows HTTP WinInet functions supported by Windows CE.

Function	Description
HttpAddRequestHeaders	Adds a HTTP request header to the HTTP request handle
HttpEndRequest	Ends an HTTP request
HttpOpenRequest	Opens an HTTP request handle
HttpQueryInfo	Queries for data about an HTTP request
HttpSendRequest	Sends the specified request to the HTTP server
HttpSendRequestEx	Sends the specified request to the HTTP server and enables chunked transfers

The following table shows FTP WinInet functions supported by Windows CE.

Function	Description
FtpCommand	Issues an arbitrary command to the FTP server
FtpCreateDirectory	Creates a new directory on the server
FtpDeleteFile	Deletes a file on the server
FtpFindFirstFile	Starts file enumeration in the current directory
FtpGetCurrentDirectory	Returns the client's current directory on the server
FtpGetFile	Retrieves an entire file from the server
FtpOpenFile	Initiates access to a file on the server for either reading or writing
FtpPutFile	Writes an entire file to the server
FtpRemoveDirectory	Deletes a directory on the server
FtpRenameFile	Renames a file on the server
FtpSetCurrentDirectory	Changes the client's current directory on the server

Persistent Caching Functions

Persistent URL cache functions are used to access and manipulate data stored in the cache. The following table shows persistent caching WinInet functions supported by Windows CE.

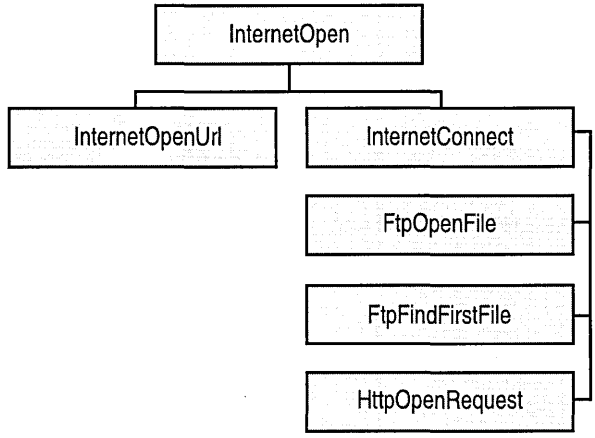
Function	Description
CommitUrlCacheEntry	Stores data in the specified file in the Internet cache and associates it with the specified URL
CreateUrlCacheEntry	Creates a local file name for saving the cache entry based on the specified URL and the file extension
CreateUrlCacheGroup	Generates a cache group identification
DeleteUrlCacheEntry	Removes the file associated with the source name from the cache, if the file exists
DeleteUrlCacheGroup	Releases the specified GROUPID and any associated state in the cache index file
FindCloseUrlCache	Closes the specified cache enumeration handle
FindFirstUrlCacheEntry	Begins the enumeration of the Internet cache
FindFirstUrlCacheEntryEx	Starts a filtered enumeration of the Internet cache
FindNextUrlCacheEntry	Retrieves the next entry in the Internet cache
FindNextUrlCacheEntryEx	Finds the next cache entry in a cache enumeration started by **FindFirstUrlCacheEntryEx**
GetUrlCacheEntryInfo	Retrieves cache entry data
GetUrlCacheEntryInfoEx	Searches for the URL after translating any cached redirections that would be applied in offline mode by **HttpSendRequest**
RetrieveUrlCacheEntryFile	Locks the cache entry file associated with the specified URL
SetUrlCacheEntryGroup	Adds entries to or removes entries from a cache group
SetUrlCacheEntryInfo	Sets the specified members of the **INTERNET_CACHE_ENTRY_INFO** structure
UnlockUrlCacheEntryFile	Unlocks the cache entry locked while the file was retrieved for use from the cache

HINTERNET Handles

Handles created and used by the WinInet functions are called HINTERNETs. These HINTERNET handles returned by the WinInet functions in Windows CE control only Internet functions. They are not native system handles. Therefore, only an Internet function works with its corresponding Internet handle. HINTERNET handles cannot be used with functions such as **ReadFile** or **CloseHandle**. Similarly, native system handles cannot be used with the WinInet functions. For example, a handle returned by **CreateFile** cannot be passed to **InternetReadFile**.

HINTERNET handles are maintained in a tree hierarchy in which a higher-level function must be called before a dependent function is called. The handle returned by the **InternetOpen** function is the root node. Handles returned by the **InternetConnect** function occupy the next level. Handles returned by the **FtpOpenFile**, **FtpFindFirstFile**, and **HttpOpenRequest** functions are the leaf nodes.

The following illustration shows the hierarchy of the HINTERNET handles. Each box represents a WinInet function that returns an HINTERNET handle.



At the top level is **InternetOpen**, which creates the root HINTERNET handle. The next level contains **InternetOpenUrl** and **InternetConnect**. The functions that use the HINTERNET handle returned by **InternetConnect** make up the last level.

All HINTERNET handles can be closed by using **InternetCloseHandle**. Client applications must close all HINTERNET handles derived from the HINTERNET handle to be closed before calling **InternetCloseHandle**.

The following code example shows the handle hierarchy for the WinInet functions.

```
HINTERNET hRootHandle, hOpenUrlHandle;
hRootHandle = InternetOpen(
                          TEXT("Example"),
                          INTERNET_OPEN_TYPE_DIRECT,
                          NULL,
                          NULL,
                          0);
hOpenUrlHandle = InternetOpenUrl(
                             hRootHandle,
                             TEXT("http://www.server.com/default.htm"),
                             NULL,
                             0,
                             INTERNET_FLAG_RAW_DATA,
                             0);

// Close the handle created by InternetOpenUrl, so that the
// InternetOpen handle can be closed.
InternetCloseHandle(hOpenUrlHandle);

// Close the handle created by InternetOpen.
InternetCloseHandle(hRootHandle);
```

Note Handle values are recycled quickly. Therefore, if a handle is closed and a new handle is generated immediately, the new handle can have the same value as the handle just closed.

Handling Uniform Resource Locators

A *Uniform Resource Locator* (URL) is a compact representation of the location and access method for a resource located on the Internet. Each URL consists of a scheme (HTTP, HTTPS, or FTP) and a scheme-specific string. This string can include a combination of a directory path, search string, or resource name. The WinInet functions provide the ability to create, combine, parse, and *canonicalize* URLs.

URLs must follow the accepted syntax and semantics to access resources through the Internet. Canonicalization is the process that converts a URL, that might contain unsafe characters, such as blank spaces, and reserved characters, into an accepted format.

The **InternetCanonicalizeUrl** function can be used to canonicalize URLs. **InternetCanonicalizeUrl** does not verify that the URL passed to it is canonicalized or that the URL it returns is valid.

The URL functions operate in a task-oriented manner. The content and format of the URL passed to the function is not verified. The calling application should track the use of these functions to ensure the data is in the intended format. For example, the **InternetCanonicalizeUrl** function would convert the character % into the escape sequence "%25" when using no flags. If **InternetCanonicalizeUrl** is used on the canonicalized URL, the escape sequence "%25" would be converted into the escape sequence "%2525" which is invalid.

Characters that must be encoded include any characters that have no corresponding graphic character in the US-ASCII coded character set; hexadecimal 80-FF, which are not used in the US-ASCII coded character set, and hexadecimal 00-1F and 7F, which are control characters; blank spaces, %, which is used to encode other characters, and unsafe characters such as <, >, ", #, {, }, |, \, ^, ~, [,], and '.

Note A relative URL is a compact representation of the location of a resource relative to an absolute base URL. The base URL must be known to the parser and usually includes the scheme, network location, and parts of the URL path. An application can call the **InternetCombineUrl** function to combine the relative URL with its base URL. **InternetCombineUrl** also canonicalizes the resulting URL.

Creating and Cracking URLs

The **InternetCreateUrl** function uses the data in the **URL_COMPONENTS** structure to create a URL.

The components that make up **URL_COMPONENTS** are the scheme, host name, port number, user name, password, URL path, and additional data, such as search parameters. Each component, except the port number, has a string member that holds the data, and a member that holds the length of the string member.

For each required component, the pointer member should contain the address of the buffer holding the data. The length member should be set to zero if the pointer member contains the address of a zero-terminated string; the length member should be set to the string length if the pointer member contains the address of a string that is not zero-terminated. The pointer member of any components that are not required must be set to NULL.

The **InternetCrackUrl** function separates a URL into its component parts and returns the components indicated by the **URL_COMPONENTS** structure passed to the function. The scheme and port numbers have only a member that stores the corresponding value; they are both returned on all successful calls to **InternetCrackUrl**.

To get the value of a particular component in **URL_COMPONENTS**, the member that stores the string length of that component must be set to a nonzero value. The string member can be either the address of a buffer or NULL.

If the pointer member contains the address of a buffer, the string length member must contain the size of that buffer. **InternetCrackUrl** returns the component data as a string in the buffer and stores the string length in the string length member.

If the pointer member is set to NULL, the string length member can be set to any nonzero value. **InternetCrackUrl** stores the address of the first character of the URL string that contains the component data and sets the string length to the number of characters in the remaining part of the URL string that pertains to the component.

All pointer members set to NULL with a nonzero length member point to the appropriate starting point in the URL string. The length stored in the length member must be used to determine the end of the individual component's data.

To finish initializing **URL_COMPONENTS** properly, the *dwStructSize* member must be set to the size of the **URL_COMPONENTS** structure.

Accessing URLs Directly

FTP and HTTP resources on the Internet can be accessed directly by using the **InternetOpenUrl**, **InternetReadFile**, and **InternetFindNextFile** functions. **InternetOpenUrl** opens a connection to the resource at the URL passed to the function.

When a successful connection is established, two things can happen:

1. If the resource is a file, **InternetReadFile** can download it.
2. If the resource is a directory, **InternetFindNextFile** can enumerate the files within the directory, except when using CERN proxies.

The **InternetReadFile** function is used to download resources from an HINTERNET handle returned by the **InternetOpenUrl**, **FtpOpenFile** or the **HttpOpenRequest** function.

InternetReadFile accepts a void pointer variable that contains the address of a buffer and a pointer to an unsigned long integer variable that contains the buffer length. It returns the data in the buffer and the amount of data downloaded into the buffer. It returns zero bytes read and completes successfully when all available data has been read. This enables an application to use **InternetReadFile** in a loop to download the data and exit when it returns zero bytes read and completes successfully.

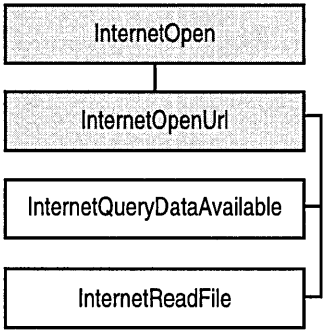
The **InternetQueryDataAvailable** function can be used with **InternetReadFile**. **InternetQueryDataAvailable** takes the HINTERNET handle created by **InternetOpenUrl**, **FtpOpenFile**, or **HttpOpenRequest**, after **HttpSendRequest** has been called on the handle, and returns the number of bytes available. The application should allocate a buffer equal to the number of bytes available and use that buffer with **InternetReadFile**. This method does not always work because **InternetQueryDataAvailable** is checking the file size listed in the header and not the actual file. The data in the header file could be outdated, or the header file could be missing, since it is not currently required under all standards.

For applications that need to operate through a CERN proxy, **InternetOpenUrl** can be used to access FTP directories and files. FTP requests are packaged to appear like an HTTP request, which the CERN proxy accepts.

InternetOpenUrl uses the HINTERNET handle created by the **InternetOpen** function and the resource URL. The URL must include the scheme: http:, ftp:, file:, or https: and network location, for example, http://www.microsoft.com. The URL can also include a path, for example, /windows/feature/ and a resource name, for example, Default.htm. For HTTP or HTTPS requests, additional headers can be included.

InternetQueryDataAvailable, **InternetFindNextFile**, and **InternetReadFile**, can use the handle created by **InternetOpenUrl** to download the resource.

The following illustration shows what handles to use with each function.



The root HINTERNET handle created by **InternetOpen** is used by **InternetOpenUrl**. The HINTERNET handle created by **InternetOpenUrl** can be used by **InternetQueryDataAvailable**, **InternetReadFile**, and **InternetFindNextFile**, which is not pictured.

Use the **InternetOpenUrl** function to parse the URL string, establish an Internet connection, and prepare for data download. This is practical for applications that are not concerned with protocol details, but only retrieve data related to a specific URL.

▶ **To use InternetOpenUrl to access a URL**

1. Call **InternetOpen** to initialize an Internet handle.
2. Call **InternetOpenUrl** to open a connection to the URL, using the handle created by **InternetOpen**.

   InternetOpenUrl returns a handle that subsequent functions can use.
3. Call **InternetReadFile** to download the resource file.

   InternetQueryDataAvailable can use the handle returned by **InternetOpenUrl** to query how much data is available to be read by a subsequent call to **InternetReadFile**.
4. Call **InternetCloseHandle** to close the handle created by **InternetOpenUrl**.
5. Call **InternetCloseHandle** to close the handle created by **InternetOpen**.

   InternetCloseHandle terminates existing activity on the handle and discards remaining data. If **InternetCloseHandle** closes a parent handle, all child handles are also closed. If an operation requires more than one Internet handle, multiple calls may need to be made to **InternetCloseHandle**.

Handling Authentication

Authentication is sometimes required before accessing resources on the Internet. Windows CE supports functions for server and proxy authentication for HTTP sessions. Authentication for FTP servers must be handled by the **InternetConnect** function.

HTTP Authentication

If authentication is required, the server sends a status code of 401—if the server requires authentication, or 407—if the proxy requires authentication. Along with the status code, the proxy or server sends one or more authenticate response headers—Proxy-Authenticate, for proxy authentication, or WWW-Authenticate, for server authentication.

Each authenticate response header contains an available authentication scheme and a realm. If multiple authentication schemes are supported, the server returns multiple authenticate response headers. The realm value is case-sensitive and defines a protection space on the proxy or server. For example, the header "WWW-Authenticate: Basic Realm="example"" would be an example of a header returned when server authentication is needed.

The client application that sent the request can authenticate itself by including an Authorization header field with the request. The Authorization header would contain the authentication scheme and the appropriate response required by that scheme.

The Windows CE Internet functions support the Basic authentication scheme, which is based on the model that a client must authenticate itself with a user name and password for each realm. The server services the request if it is resent with an Authorization header that includes a valid user name and password.

For anything other than Basic authentication, you must use the *Security Support Provider Interface* (SSPI), which enables applications to access security DLLs called *Security Support Providers* (SSPs). For more information on SSPI, see Security Support Provider Interface. The registry keys must be set up in addition to installing the appropriate DLL(s). For more information on setting these registry keys, see Registering Authentication Keys.

The application should call the **HttpOpenRequest** function if authentication is required. The INTERNET_FLAG_KEEP_CONNECTION flag should be used for NTLM and other types of authentication to maintain the connection while completing the authentication process. If the connection is not maintained, the authentication process must be restarted with the proxy or server.

InternetOpenUrl and **HttpSendRequest** complete successfully even when authentication is required. However, the data returned in the header files and **InternetReadFile** would receive an HTML page informing the user of the status code.

Registering Authentication Keys

The INTERNET_OPEN_TYPE_PRECONFIG flag in the **InternetOpen** function looks at the registry values ***ProxyEnable***, ***ProxyServer***, and ***ProxyOverride***. These values are located under the **HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\ Internet Settings** key.

For authentication schemes other than Basic, a key needs to be added to the registry under the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\Security** key. A string value, DLLFile, should contain the name of the DLL that supports the authentication scheme. A **DWORD** value, **Flags**, should be set with the appropriate value.

The following table shows the possible values for the **Flags** value.

Flag value	Description
PLUGIN_AUTH_FLAGS_UNIQUE_CONTEXT_PER_TCPIP (value=0x01)	Each TCP/IP socket contains a different context. Otherwise, a new context is passed for each realm or block URL template.
PLUGIN_AUTH_FLAGS_CAN_HANDLE_UI (value=0x02)	This DLL can handle its own user input.
PLUGIN_AUTH_FLAGS_CAN_HANDLE_NO_PASSWD (value=0x04)	This DLL might be capable of doing an authentication without prompting the user for a password.
PLUGIN_AUTH_FLAGS_NO_REALM (value=0x08)	This DLL does not use a standard HTTP realm string. Any data that appears to be a realm is scheme-specific.
PLUGIN_AUTH_FLAGS_KEEP_ALIVE_NOT_REQUIRED (value=0x10)	This DLL does not require a persistent connection for its challenge-response sequence.

For example, to add NTLM authentication, the subkey **NTLM** would need to be added to **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ Internet Explorer\Security** key. Under the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ Internet Explorer\Security\NTLM** key, the string value, DLLFile, and a **DWORD** value, **Flags**, would need to be added. DLLFile would need to be set to Winsspi.dll, and **Flags** would need to be set to 0x08.

Server Authentication

When a server receives a request that requires authentication, the server returns a 401 status code message. In that message, the server should include one or more WWW-Authenticate response headers. These headers include the authentication methods the server has available. The Windows CE Internet functions pick the first method they recognize.

Basic authentication provides weak security unless the channel is first link-encrypted with SSL or Private Communication Technology (PCT).

The **InternetErrorDlg** function can be used to obtain the user name and password data from the user, or a custom control can be designed to obtain the data.

A custom control can use the **InternetSetOption** function to set the INTERNET_OPTION_PASSWORD and INTERNET_OPTION_USERNAME values and then resend the request to the server.

Proxy Authentication

A proxy server is used as a security barrier between the internal network and the Internet. This protects the internal network from unauthorized access. A proxy server also enhances speed performance since it caches recently used documents. To gain access to a proxy server, the Windows CE–based application needs a proxy authentication scheme.

When a Windows CE–based application tries to use a proxy server that requires authentication, the proxy returns a 407 status code message to the client. In that message, the proxy should include one or more Proxy-Authenticate response headers. These headers include the authentication methods available from the proxy. The Windows CE Internet functions pick the first method they recognize.

To get user name and password from the user the application can use the **InternetErrorDlg** function.

A custom interface can use **InternetSetOption** to set the INTERNET_OPTION_PROXY_PASSWORD and INTERNET_OPTION_PROXY_USERNAME values and then resend the request to the proxy.

If no proxy user name and password are set, the Windows CE Internet functions attempt to use the user name and password for the server.

Handling HTTP Authentication

When handling HTTP authentication, the application can use several functions: **InternetErrorDlg**, or a function that uses **InternetSetOption** that sets its own Internet options. **InternetErrorDlg** checks the headers associated with an HINTERNET handle to find hidden errors, such as status codes from a proxy or server. **InternetSetOption** can be used to set the user name and password for the proxy and server.

For any customized function that adds its own WWW-Authenticate or Proxy-Authenticate headers, the INTERNET_FLAG_NO_AUTH flag should be set to disable the Windows CE Internet API authentication.

In the example, *dwErrorCode* is used to store any errors associated with the call to **HttpSendRequest**. **HttpSendRequest** completes successfully, even if the proxy or server requires authentication. When the FLAGS_ERROR_UI_FILTER_FOR_ERRORS flag is passed to **InternetErrorDlg**, the function checks the headers for any hidden errors. These hidden errors would include any requests for authentication. **InternetErrorDlg** displays the appropriate dialog box to prompt the user for the necessary data. The FLAGS_ERROR_UI_FLAGS_GENERATE_DATA and FLAGS_ERROR_UI_FLAGS_CHANGE_OPTIONS flags should also be passed to **InternetErrorDlg**, so that the function constructs the appropriate data structure for the error and stores the results of the dialog box in the HINTERNET handle.

Managing Cookies

Cookies are used for tracking data settings, or data for a particular Web site. The cookies are saved on the client device, and when the browser requests a page, it sends the data settings for that page along with the request. The browser can only send the data back to the server that originally created them; therefore, cookies are a secure way of maintaining user-specific data. Cookies can be temporary or persistent.

The Windows CE Internet functions have a persistent cookie database for this purpose. The Windows CE Internet cookie functions are used to set cookies into and access cookies from the cookie database. For more information, see HTTP Cookies.

The Windows CE Internet functions **InternetSetCookie** and **InternetGetCookie** can be used to manage cookies.

Unlike most Windows CE Internet functions, the cookie functions do not require a call to **InternetOpen**. Cookies that have an expiration date, persistent cookies, are stored in the Windows\Cookies directory. Cookies that do not have an expiration date, session cookies, are stored in memory and are available only to the process in which they were created.

InternetGetCookie returns the cookies for the specified URL, and all its parent URLs. **InternetGetCookie** checks persistent storage for cookies and searches memory for any cookies that do not have an expiration date, since these cookies are not written to any files.

InternetSetCookie is used to set a cookie on the specified URL. **InternetSetCookie** can create both persistent and session cookies.

HTTP Cookies

HTTP cookies provide the server with a mechanism to store and retrieve state data on the client application's system. This mechanism enables Web-based applications to store data about selected items, user preferences, registration data, and other data that can be retrieved later.

Cookie-Related Headers

There are two HTTP headers, Set-Cookie and Cookie, that are related to cookies. The Set-Cookie header is sent by the server in response to an HTTP request, which is used to create a cookie on the user's system. The Cookie header is included by the client application with an HTTP request sent to a server, if there is a cookie that has a matching domain and path.

Set-Cookie Header

The Set-Cookie response header uses the following syntax:

```
Set-Cookie: <name>=<value>[; <name>=<value>]...
[; expires=<date>][; domain=<domain_name>]
[; path=<some_path>][; secure]
```

One or more string sequences, separated by semicolons, that follow the pattern <name>=<value> must be included in the Set-Cookie response header. The server can use these string sequences to store data on the client's system.

The expiration date is set by using the format expires=<date>, where <date> is the expiration date in Greenwich Mean Time (GMT). If the expiration date is not set, the cookie expires after the Internet session ends. Otherwise, the cookie persists in the cache until the expiration date.

The following table shows the expiration date format.

Format	Description
DD-MMM-YYYY HH:MM:SS GMT	Complete format.
DD	DD is the day in the month—such as 01 for the first day of the month.
MMM	MMM is the three-letter abbreviation for the month: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec.
YYYY	YYYY is the year.
HH:MM:SS GMT	HH is the hour value in military time—22 would be 10:00 P.M., for example—MM is the minute value, and SS is the second value.

Specifying the domain name, using the pattern domain=<domain_name>, is optional for persistent cookies and is used to indicate the end of the domain for which the cookie is valid. Session cookies that specify a domain are rejected. If the specified domain name ending matches the request, the cookie tries to match the path to determine if the cookie should be sent. For example, if the domain name ending is .microsoft.com, requests to home.microsoft.com and support.microsoft.com would be checked to see if the specified pattern matches the request. The domain name must have at least two or three periods in it to prevent cookies from being set for widely used domain name endings, such as .com and .edu. Acceptable domain names would be similar to .microsoft.com, .someschool.edu, and .someserver.co.jp. Only hosts within the specified domain can set a cookie for a domain.

Setting the path, using the pattern path=<some_path>, is optional and can be used to specify a subset of the URLs for which the cookie is valid. If a path is specified, the cookie is considered valid for any requests that match that path. For example, if the specified path is /example, requests with the paths /examplecode and /example/code.htm would match. If no path is specified, the path is assumed to be the path of the resource associated with the Set-Cookie header.

The cookie can also be marked as secure, which specifies that the cookie can be sent only to HTTPS servers.

Cookie Header

The Cookie header is included with any HTTP requests that have a cookie whose domain and path match the request. The Cookie header has the following syntax:

```
Cookie: <name>=<value> [;<name>=<value>]...
```

One or more string sequences, using the format <name>=<value>, contain the data that was set in the cookie.

Generating Cookies

There are three methods for generating cookies:

- Using the DHTML Object Model, compatible with ECMA 262 language specification
- Using the Windows CE Internet functions
- Using a CGI script

All of the methods need to set the data that is included in the Set-Cookie header.

Generating a Cookie Using the DHTML Object Model

Using the DHTML object model, cookies can be set by calling the cookie property of the document object, as shown in the following code example.

```
<SCRIPT language="JavaScript">
<!--
    document.cookie = "SomeValueName = Some_Value";
-->
</SCRIPT>
```

Generating a Cookie Using the Windows CE Internet Functions

Cookies can be created by applications using the **InternetSetCookie** function. For more information about setting cookies using the Windows CE Internet functions, see WinInet Functions.

Generating a Cookie Using a CGI Script

Cookies are generated by including a Set-Cookie header as part of a CGI script included in the HTTP response to a request.

The following example is a CGI script that includes a Set-Cookie header using Perl.

```
print "Set-Cookie:Test=test_value; expires=Sat, 01-Jan-2000 00:00:00
GMT;
path=/;"
```

Caching

The Windows CE Internet functions supports built-in caching. Any data retrieved from the network is cached on the Windows CE–based device and retrieved for subsequent requests. The application using the Windows CE Internet functions can control the caching on each request. For HTTP requests from the server, most headers received are also cached. When an HTTP request is satisfied from the cache, the cached headers are also returned to the caller.

Using Flags to Control Caching

The Windows CE Internet function flags enable an application to control when and how it uses the cache. These flags can be used alone or in combination with the *dwFlags* parameter in functions that access data or resources on the Internet. The Windows CE Internet functions store all data downloaded from the Internet by default.

The following table shows values that can be used with the Windows CE Internet functions to control caching.

Value	Description
INTERNET_FLAG_DONT_CACHE	Does not cache the data, either locally or in any gateways. Identical to the preferred value, INTERNET_FLAG_NO_CACHE_WRITE.
INTERNET_FLAG_HYPERLINK	Forces the application to reload a resource if no expire time and no last-modified time was returned when the resource was stored in the cache
INTERNET_FLAG_MUST_CACHE_REQUEST	Causes a temporary file to be created if the file cannot be cached. Identical to the preferred value, INTERNET_FLAG_NEED_FILE.
INTERNET_FLAG_NEED_FILE	Causes a temporary file to be created if the file cannot be cached

Value	Description
INTERNET_FLAG_NO_CACHE_WRITE	Rejects any attempt by the function to store data downloaded from the Internet in the cache. This flag is necessary if the application does not require downloaded resources to be stored locally.
INTERNET_FLAG_OFFLINE	Prevents the application from making requests to the network. All requests are resolved using the resources stored in the cache. If the resource is not in the cache, a suitable error, such as ERROR_FILE_NOT_FOUND, is returned.
INTERNET_FLAG_RELOAD	Forces the function to retrieve the requested resource directly from the Internet. The data that is downloaded is stored in the cache.
INTERNET_FLAG_RESYNCHRONIZE	Causes an application to perform a conditional download of the resource from the Internet. If the version stored in the cache is current, the data is downloaded from the cache. Otherwise, the data is reloaded from the server.

Using Persistent Caching Functions

This section describes how to use the functions used by applications that need persistent caching services. These functions enable an application to save data in the local file system for subsequent use, such as in situations where access to the data is over a low-bandwidth link or the access is not available at all. The calling application that inserts data into the persistent cache assigns a source name that is used to do operations such as retrieve the data, set and get some properties on the data, and delete data.

The Windows CE Internet functions use the cache functions to provide persistent caching. Unless explicitly specified not to cache through the INTERNET_FLAG_NO_CACHE_WRITE flag, Windows CE Internet functions cache all data downloaded from the network. The responses to POST data are not cached.

Clients that need persistent caching services use the persistent caching functions to enable their applications to save data in the local file system for subsequent use, such as in situations where a low-bandwidth link limits access to the data or the access is not available at all. The calling application that inserts data into the persistent cache assigns a source name that is used to perform operations, including retrieving, setting, and getting some properties and deleting the data.

Enumerating the Cache

The **FindFirstUrlCacheEntry** and **FindNextUrlCacheEntry** functions enumerate the data stored in the cache. **FindFirstUrlCacheEntry** starts the enumeration by taking a search pattern, a buffer, and a buffer size to create a handle and return the first cache entry. **FindNextUrlCacheEntry** takes the handle created by **FindFirstUrlCacheEntry**, a buffer, and a buffer size to return the next cache entry.

Both functions store an **INTERNET_CACHE_ENTRY_INFO** structure in the buffer. The size of this structure varies for each entry. If the buffer size passed to either function is insufficient, the function fails, and **GetLastError** returns ERROR_INSUFFICIENT_BUFFER. The buffer size variable contains the buffer size that was needed to retrieve that cache entry. A buffer of the size indicated by the buffer size variable should be allocated, and the function should be called again with the new buffer.

INTERNET_CACHE_ENTRY_INFO contains the structure size; URL of the cached data; local file name; cache entry type; use count; hit rate; size; last modified, expire, last access, and last synchronized times; header data and header data size; and file extension.

FindFirstUrlCacheEntry takes a search pattern, a buffer that stores the **INTERNET_CACHE_ENTRY_INFO** structure, and the buffer size. Currently, only the default search pattern, which returns all cache entries, is implemented.

After the cache is enumerated, the application should call **FindCloseUrlCache** to close the cache enumeration handle.

Retrieving Cache Entry Data

The **GetUrlCacheEntryInfo** function lets you retrieve the **INTERNET_CACHE_ENTRY_INFO** structure for the specified URL. This structure contains the structure size, URL of the cached data, local file name, cache entry type, use count, hit rate, size, last modified, expire, last access, and last synchronized times, header data and header data size, and file extension.

GetUrlCacheEntryInfo accepts a URL, a buffer for an **INTERNET_CACHE_ENTRY_INFO** structure, and the buffer size. If the URL is found, the data is copied into the buffer. Otherwise, the function fails and **GetLastError** returns ERROR_FILE_NOT_FOUND. If the buffer size is insufficient to store the cache entry data, the function fails and **GetLastError** returns ERROR_INSUFFICIENT_BUFFER. The size required to retrieve the data is stored in the buffer size variable.

GetUrlCacheEntryInfo does not do any URL parsing, so a URL containing an anchor (#) is not found in the cache, even if the resource is cached. For example, if the URL http://example.com/example.htm#sample was passed, the function would return ERROR_FILE_NOT_FOUND even if http://example.com/example.htm is in the cache.

Creating a Cache Entry

An application uses the **CreateUrlCacheEntry** and **CommitUrlCacheEntry** functions to create a cache entry.

CreateUrlCacheEntry accepts the URL, expected file size, and file extension. The function then creates a local file name for saving the cache entry corresponding to the URL and file extension.

Using the local file name, write the data into the local file using standard C/C++ functions or Windows CE functions. After the data has been written to the local file, the application should call **CommitUrlCacheEntry**.

CommitUrlCacheEntry accepts the URL, local file name, expire and last modified times, cache entry type, header data and header data size, and file extension. The function then caches data in the file specified in the cache storage and associates it with the given URL.

Deleting a Cache Entry

The **DeleteUrlCacheEntry** function takes a URL and removes the associated cache file. If the cache file does not exist, the function fails and **GetLastError** returns ERROR_FILE_NOT_FOUND. If the cache file is currently locked or in use, the function fails and **GetLastError** returns ERROR_ACCESS_DENIED, and the file will be deleted when unlocked.

Retrieving Cache Entry Files

For applications that require the file name of a resource to launch, the Windows CE Internet API provides the **RetrieveUrlCacheEntryFile** and **UnlockUrlCacheEntryFile** functions.

RetrieveUrlCacheEntryFile accepts a URL, a buffer that stores the **INTERNET_CACHE_ENTRY_INFO** structure, and the buffer size. After the file data has been used, the application should call **UnlockUrlCacheEntryFile** to unlock the file.

Cache Groups

The Internet functions support cache groups. To create a cache group, the **CreateUrlCacheGroup** function must be called to generate a GROUPID for the cache group. Entries can be added to the cache group by supplying the cache entry URL and the INTERNET_CACHE_GROUP_ADD flag to the **SetUrlCacheEntryGroup** function. To remove a cache entry from a group, pass the cache entry URL and the INTERNET_CACHE_GROUP_REMOVE flag to **SetUrlCacheEntryGroup**.

The **FindFirstUrlCacheEntryEx** and **FindNextUrlCacheEntryEx** functions can be used to enumerate the entries in a specified cache group. After the enumeration is complete, the function should call **FindCloseUrlCache**.

Handling Structures with Variable Size Data

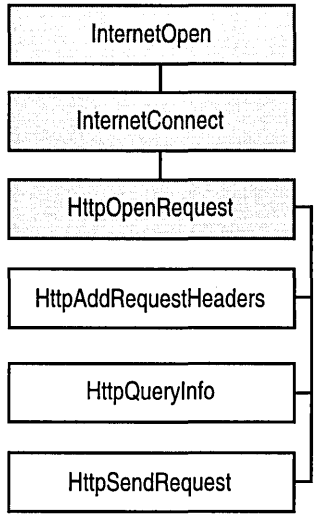
The cache can contain variable size data for each URL stored. This is reflected in the **INTERNET_CACHE_ENTRY_INFO** structure. When the cache functions return this structure, they create a buffer that is always the size of **INTERNET_CACHE_ENTRY_INFO** plus any variable size data. If a pointer member is not NULL, it points to the memory area immediately after the structure. While copying the returned buffer from a function into another buffer, the pointer members should be fixed to point to the appropriate place in the new buffer. The following code example shows how to copy into another buffer.

```
lpDstCEInfo->lpszSourceUrlName = (LPINTERNET_CACHE_ENTRY_INFO) (
    (LPBYTE) lpSrcCEInfo +
    ((DWORD) (lpOldCEInfo->lpszSourceUrlName) - (DWORD) lpOldCEInfo))
```

Some cache functions fail with the ERROR_INSUFFICIENT_BUFFER error message if you specify a buffer that is too small to contain the cache-entry data retrieved by the function. In this case, the function also returns the required size of the buffer. You can then allocate a buffer of the appropriate size and call the function again.

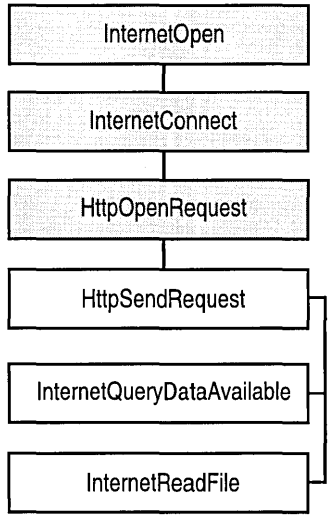
Accessing the HTTP Protocol

Use the HTTP functions provided by WinInet to use the HTTP protocol to access resources on the Internet. The following illustration shows the relationships of the WinInet functions used to access the HTTP protocol. Shaded boxes represent functions that return HINTERNET handles, while the plain boxes represent functions that use the HINTERNET handle created by the function on which they depend.



HttpAddRequestHeaders, **HttpQueryInfo**, and **HttpSendRequest**, are dependent on the HINTERNET handle created by **HttpOpenRequest**.

The following illustration shows the WinInet functions that use the HINTERNET handle created by **HttpOpenRequest** after it is sent by **HttpSendRequest**. The shaded boxes represent functions that return HINTERNET handles, while the plain boxes represent functions that use the HINTERNET handle created by the function on which they depend.



After **HttpSendRequest** has been used on the handle returned by **HttpOpenRequest**, **InternetQueryDataAvailable**, and **InternetReadFile**, can be used on that handle.

▶ **To use the HTTP WinInet functions**

1. Call the **InternetOpen** function to initialize an Internet handle.

   InternetOpen creates the root HINTERNET handle used to establish the HTTP session. The HINTERNET is used by all subsequent functions.

2. Call **InternetConnect** using the HINTERNET returned by **InternetOpen** to create an HTTP session.

   When calling **InternetConnect**, specify INTERNET_DEFAULT_HTTP for the *nServerPort* parameter and INTERNET_SERVICE_HTTP for the *dwService* parameter.

   InternetConnect uses the handle returned by **InternetOpen** to create a specific HTTP session. **InternetConnect** initializes an HTTP session for the specified site, using the arguments passed to it and creates HINTERNET that is a branch off the root handle. **InternetConnect** does not attempt to access or establish a connection to the specified site.

3. Call **HttpOpenRequest** to open an HTTP request handle.

   HttpOpenRequest uses the handle created by **InternetConnect** to establish a connection to the specified site.

4. Call **HttpSendRequest**, using the handle created by **HttpOpenRequest** to send an HTTP request to the HTTP server.
5. Call **InternetReadFile** to download data.

   –Or–

   Call **InternetQueryDataAvailable** to query how much data is available to be read by a subsequent call to **InternetReadFile**.
6. Call **InternetCloseHandle** to close the handle created by **HttpOpenRequest**.
7. Call **InternetCloseHandle** to close the HTTP session created by **InternetConnect**.
8. Call **InternetCloseHandle** to close the handle created by **InternetOpen**.

The following example **GetInternetFile** function shows how to use the HTTP functions to establish an HTTP session and retrieve a file. Two global **Boolean** variables g_bproxy, use proxy server, and g_bOpenURL, use URL, are options set in the CeHttp application dialog.

```
/*****************************************************************************

FUNCTION:
  GetInternetFile

PURPOSE:
  This function demonstrates how to create and submit an HTTP request.
  It requests the default HTML document from the server, and then
  displays it along with the HTTP transaction headers.

*****************************************************************************/
BOOL GetInternetFile (LPTSTR lpszServer, LPTSTR lpszProxyServer)
{
  BOOL bReturn = FALSE;

  HINTERNET hOpen = NULL,
            hConnect = NULL,
            hRequest = NULL;

  DWORD dwSize = 0,
        dwFlags = INTERNET_FLAG_RELOAD | INTERNET_FLAG_NO_CACHE_WRITE;

  TCHAR szErrMsg[200];

  char *lpBufferA,
       *lpHeadersA;
```

```
TCHAR *lpBufferW,
      *lpHeadersW;

LPTSTR AcceptTypes[2] = {TEXT("*/*"), NULL};


// Initialize the use of the Windows CE Internet functions.
if (g_bProxy)
{
  hOpen = InternetOpen (TEXT("CeHttp"), INTERNET_OPEN_TYPE_PROXY,
                        lpszProxyServer, 0, 0);
}
else
{
  hOpen = InternetOpen (TEXT("CeHttp"), INTERNET_OPEN_TYPE_PRECONFIG,
                        NULL, 0, 0);
}

if (!hOpen)
{
  wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("InternetOpen Error"),
            GetLastError());
  return FALSE;
}

if (g_bOpenURL)
{
  if (!(hRequest = InternetOpenUrl (hOpen, lpszServer, NULL, 0,
                                    INTERNET_FLAG_RELOAD, 0)))
  {
    wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("InternetOpenUrl Error"),
              GetLastError());
    goto exit;
  }
}
else
{
  // Open an HTTP session for a specified site by using lpszServer.
  if (!(hConnect = InternetConnect (hOpen,
                                    lpszServer,
                                    INTERNET_INVALID_PORT_NUMBER,
                                    NULL, NULL,
                                    INTERNET_SERVICE_HTTP,
                                    0, 0)))
  {
    wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("InternetConnect Error"),
              GetLastError());
    goto exit;
  }
```

```
  // Open an HTTP request handle.
  if (!(hRequest = HttpOpenRequest (hConnect,
                                    TEXT("GET"),
                                    NULL,
                                    HTTP_VERSION,
                                    NULL,
                                    (LPCTSTR*)AcceptTypes,
                                    dwFlags, 0)))
  {
    wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("HttpOpenRequest Error"),
              GetLastError());
    goto exit;
  }

  // Send a request to the HTTP server.
  if (!HttpSendRequest (hRequest, NULL, 0, NULL, 0))
  {
    wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("HttpSendRequest Error"),
              GetLastError());
    goto exit;
  }
}

// Call HttpQueryInfo to find out the size of the headers.
HttpQueryInfo (hRequest, HTTP_QUERY_RAW_HEADERS_CRLF, NULL, &dwSize,
               NULL);

// Allocate a block of memory for lpHeadersA.
lpHeadersA = new CHAR [dwSize];

// Call HttpQueryInfo again to get the headers.
if (!HttpQueryInfo (hRequest,
                    HTTP_QUERY_RAW_HEADERS_CRLF,
                    (LPVOID) lpHeadersA, &dwSize, NULL))
{
  wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("HttpQueryInfo"),
            GetLastError());
  goto exit;
}
else
{
  // Clear all of the existing text in the edit control and prepare
  // to put the new information in it.
  SendMessage (g_hwndEdit, EM_SETSEL, 0, -1);
  SendMessage (g_hwndEdit, WM_CLEAR, 0, 0);
  SendMessage (g_hwndEdit, WM_PAINT, TRUE, 0);
}
```

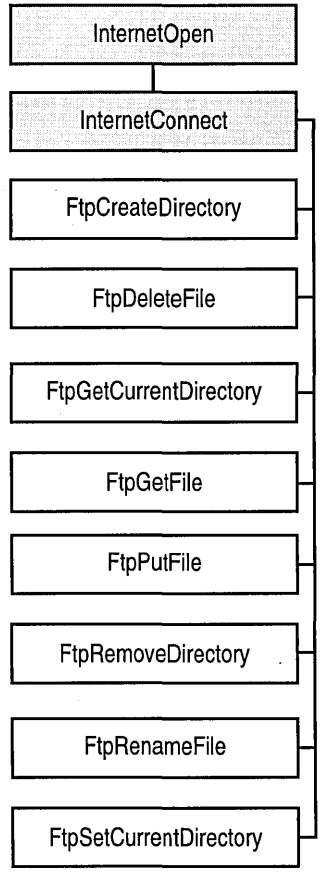
```
// Terminate headers with NULL.
lpHeadersA [dwSize] = '\0';

// Get the required size of the buffer that receives the Unicode
// string.
dwSize = MultiByteToWideChar (CP_ACP, 0, lpHeadersA, -1, NULL, 0);

// Allocate a block of memory for lpHeadersW.
lpHeadersW = new TCHAR [dwSize];

// Convert headers from ASCII to Unicode
MultiByteToWideChar (CP_ACP, 0, lpHeadersA, -1, lpHeadersW, dwSize);

// Put the headers in the edit control.
SendMessage (g_hwndMain, WM_PUTTEXT, NULL, (LPARAM) lpHeadersW);

// Free the blocks of memory.
delete[] lpHeadersA;
delete[] lpHeadersW;

// Allocate a block of memory for lpHeadersW.
lpBufferA = new CHAR [32000];

do
{
  if (!InternetReadFile (hRequest, (LPVOID)lpBufferA, 32000, &dwSize))
  {
    wsprintf(szErrMsg, TEXT("%s: %x"), TEXT("InternetReadFile Error"),
             GetLastError());
    goto exit;
  }

  if (dwSize != 0)
  {
    // Terminate headers with NULL.
    lpBufferA [dwSize] = '\0';

    // Get the required size of the buffer which receives the Unicode
    // string.
    dwSize = MultiByteToWideChar (CP_ACP, 0, lpBufferA, -1, NULL, 0);

    // Allocate a block of memory for lpBufferW.
    lpBufferW = new TCHAR [dwSize];

    // Convert the buffer from ASCII to Unicode.
    MultiByteToWideChar (CP_ACP, 0, lpBufferA, -1, lpBufferW, dwSize);
```

```
            // Put the buffer in the edit control.
            SendMessage (g_hwndMain, WM_PUTTEXT, NULL, (LPARAM) lpBufferW);

            // Free the block of memory.
            delete[] lpBufferW;
        }
    } while (dwSize);

    // Free the block of memory.
    delete[] lpBufferA;

    bReturn = TRUE;

exit:

    // Close the Internet handles.
    if (hOpen)
    {
        if (!InternetCloseHandle (hOpen))
            wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("CloseHandle Error"),
                      GetLastError());
    }

    if (hConnect)
    {
        if (!InternetCloseHandle (hConnect))
            wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("CloseHandle Error"),
                      GetLastError());
    }

    if (hRequest)
    {
        if (!InternetCloseHandle (hRequest))
            wsprintf (szErrMsg, TEXT("%s: %x"), TEXT("CloseHandle Error"),
                      GetLastError());
    }

    return bReturn;
}
```

Accessing the FTP Protocol

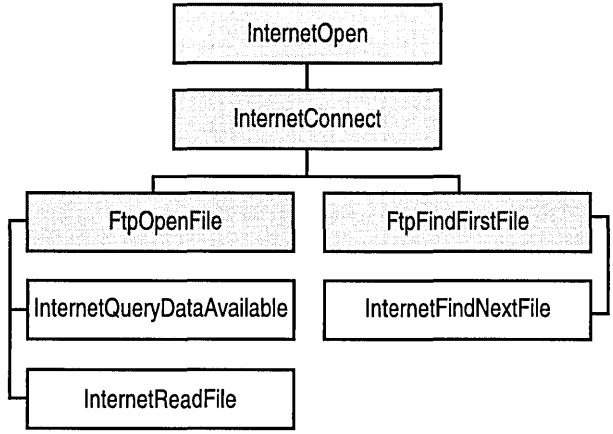
Wininet.dll and the corresponding Wininet.lib do not include the FTP APIs. To develop FTP applications on a version of the Windows CE OS that does not export FTP APIs from the Wininet.dll, you can use Winsock. FTP APIs are included in the emulation library, Wininetm.lib, and will work in the emulation environment. You can use the FTP APIs with Windows CE version 2.12 which is included in Wininet.dll and Wininet.lib.

The following illustration shows the FTP functions dependent on the FTP session HINTERNET returned by **InternetConnect**. The shaded boxes represent functions that return HINTERNETs and the plain boxes represent functions that use the HINTERNET created by the function on which they depend.



The **FtpCreateDirectory**, **FtpDeleteFile**, **FtpGetCurrentDirectory**, **FtpGetFile**, **FtpPutFile**, **FtpRemoveDirectory**, **FtpRenameFile**, and **FtpSetCurrentDirectory** functions use the HINTERNET handle created by **InternetConnect**.

The following illustration shows the two FTP functions that return HINTERNET handles and the functions dependent on the HINTERNET handles created by them. The shaded boxes represent functions that return HINTERNET handles, while the plain boxes represent functions that use the HINTERNET handle created by the function on which they depend.



InternetFindNextFile is dependent on the HINTERNET handle created by **FtpFindFirstFile**, and **InternetReadFile** uses the HINTERNET handle created by **FtpOpenFile**.

Use WinInet to perform the following tasks on an FTP server:

- Navigating directories
- Enumerating, creating, removing, and renaming directories
- Renaming, uploading, downloading, and deleting files

▶ To access an FTP server with WinInet

1. Call **InternetOpen** to initialize an Internet handle.

   InternetOpen creates the root HINTERNET handle that is used to establish the FTP session. The HINTERNET Internet handle is used by all subsequent functions.

2. Call **InternetConnect** to create an FTP session.

   When calling **InternetConnect** specify INTERNET_DEFAULT_FTP_PORT for the *nServerPort* parameter and INTERNET_SERVICE_FTP for the *dwService* parameter.

   This function uses the handle returned by **InternetOpen** to create a specific FTP session. **InternetConnect** initializes an FTP session for the specified site, using the arguments passed to it and creates a HINTERNET that is a branch off the root handle. In the case of an FTP session, **InternetConnect** attempts to establish a connection to the specified site.

3. Call **FtpGetFile** or **FtpFindFirstFile**.

   InternetConnect returns a handle that subsequent functions can use, such as **FtpGetFile** or **FtpFindFirstFile**.

   Use the **InternetFindNextFile** function with **FtpFindFirstFile** to find the next file in a file search, using the search parameters and HINTERNET handle from **FtpFindFirstFile**.

   To complete a file search, continue to call **InternetFindNextFile** using the HINTERNET handle returned by **FtpFindFirstFile** until function fails with the extended error message ERROR_NO_MORE_FILES. To get the extended error data, call the **GetLastError** function.

4. Call **InternetCloseHandle** to close the FTP session created by calling **InternetConnect**.

5. Call **InternetCloseHandle** to close the handle created by calling **InternetOpen**.

Note Applications must specify a directory relative to the current directory or include the full directory path.

Accessing Security Protocols

Windows CE supports Private Communication Technology (PCT) 1.0 and secure socket layer (SSL) versions 2.0 and 3.0, and Server Gated Crypto (SGC) security protocols. These protocols are available through WinInet or directly from Winsock.

The simplest approach to using the security protocols is to use WinInet.

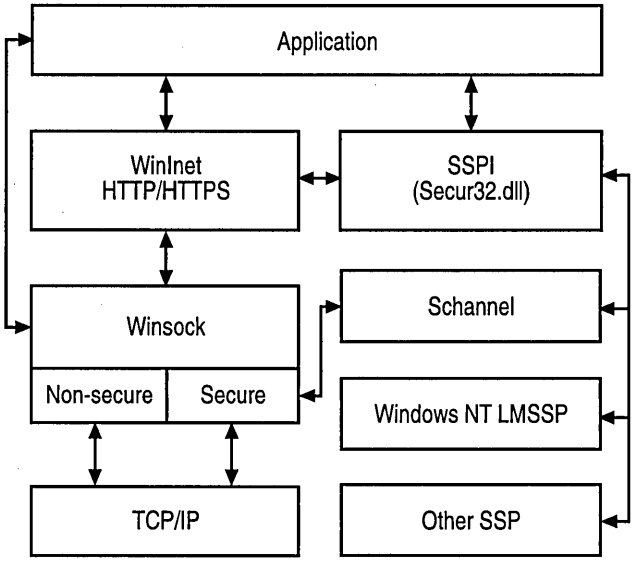
▶ **To access security protocols with WinInet**

1. Call **InternetOpen** to get an Internet handle.
2. Connect with **InternetConnect**, using INTERNET_DEFAULT_HTTPS_PORT as the *nServerPort* parameter.
3. For HTTPS, invoke **HttpOpenRequest** with the INTERNET_FLAG_SECURE flag set.
4. Proceed with the remainder of the session.

CHAPTER 8

Security Support Provider Interface

As intranets become more secure, client applications, such as Web browsers and e-mail applications, and their servers become more complex. Different applications require different ways of identifying or authenticating users, and different ways of encrypting data as it travels across a network. To avoid coding every available security option into an application, Windows CE supports the *Security Support Provider Interface* (SSPI), which enables applications to access dynamic-link libraries (DLLs) containing common authentication and cryptographic data schemes. These DLLs are called *Security Support Providers* (SSPs). The following illustration shows the relationship of the SSP DLLs to the SSPI Secur32.dll, Winsock, and WinInet.



SSPs make one or more security solutions, called *security packages*, available to applications. A security package maps various SSPI functions to the security protocols specified in the package. An application implementing the SSPI can use any security package available on a system without knowing details about the security protocols that the security package implements. The application programming interfaces (APIs) contained in the SSPI are divided into the following functional areas:

- Package management
- Credential management
- Context management
- Message support

Package management functions enumerate and query the attributes of the security packages of an SSP. They list the security packages available on a system and enable an application to select one to support its requirements.

Credential management functions enable applications to gain access to the *credentials* of a *principal*. A principal is an entity recognized by the security system. This includes human users as well as autonomous processes. A credential is data used by a principal to establish the identity of the principal, such as a password or user name.

Context management functions enable applications to create and use *security contexts*. A security context is the security data relevant to a connection, and contains such data as a session key and the session duration. Both client and server must cooperate to create a security context. The client and the server can then use the security context with message support functions to ensure message integrity and privacy during the connection.

Message support functions enable an application to transmit messages that cannot be tampered with. The functions work with one or more buffers that contain a message and an associated security context created by the context management functions.

These sections describe how to initialize and use the functions contained in the SSPI to create a secure network connection. This process contains the following primary tasks:

- Initializing the SSPI
- Establishing an authentic connection
- Ensuring communication integrity during message exchange
- Calling the Windows NT LAN Manager Security Support Provider (Windows NT LMSSP)

The following example shows how to update the registry to install an SSP.

```
[HKEY_LOCAL_MACHINE\Comm\SecurityProviders]
    Providers=REG_SZ:provider1.dll, provider2.dll,...
```

A single DLL can contain multiple providers. Provider.dll can contain two security packages; for example, Protocol 1 and Protocol 2.

SSPI Functions and Structures

The following table shows the SSPI functions supported by Windows CE.

Function	Description
AcceptSecurityContext	Establishes a security context between the server and a remote client.
AcquireCredentialsHandle	Enables an application to acquire a handle to preexisting credentials associated with the user on whose behalf the call is made.
ApplyControlToken	Provides a way to apply a control token to a security context.
DeleteSecurityContext	Deletes local data structures associated with the specified security context.
EnumerateSecurityPackages	Returns an array of **SECPKGINFO** structures that describe the security packages available to the client.
FreeContextBuffer	Enables callers of security provider functions to free a memory buffer allocated by the security provider.
FreeCredentialsHandle	Notifies the security system that the credentials are no longer needed.
InitSecurityInterface	Returns a pointer to an SSPI dispatch table.

Function	Description
InitializeSecurityContext	Initiates the outbound security context from a credential handle. This function establishes a security context between the client application and a remote peer.
MakeSignature	Generates a cryptographic checksum of the message and includes sequencing data to prevent message loss or insertion. This function enables the application to choose from several cryptographic algorithms.
QueryContextAttributes	Enables a transport application to query a security package for certain security context attributes.
QueryCredentialsAttributes	Retrieves the credential attributes.
QuerySecurityPackageInfo	Retrieves data about a specified security package.
VerifySignature	Verifies the signature of a peer client message.

The following table shows the SSPI structures supported by Windows CE.

Structure	Description
SecPkgInfo	Provides general security package data, such as its name and capabilities
SecPkgContext_Sizes	Indicates the sizes of important structures used in the message support functions
SecPkgContext_Names	Indicates the user name associated with a security context
SecPkgContext_Lifespan	Indicates the life span of a security context
SecPkgContext_DceInfo	Contains authorization data used by DCE services
SecPkgCredentials_Names	Indicates the name of the user associated with a context

Initializing the SSPI

Before initializing the SSPI, an application must load the security provider, using the **LoadLibrary** function.

The following code example shows how to load the security provider DLL.

```
HINSTANCE DllHandle;
TCHAR szError[100];

// Load the security provider DLL.
DllHandle = LoadLibrary (TEXT("secur32.dll"));

if (!DllHandle)
{
  wsprintf (szError,
            TEXT("Failed in loading secur32.dll, Error: %x"),
            GetLastError ());
  return 0;
}
```

When the SSP has loaded successfully, the client and the server call the **GetProcAddress** function to get a pointer to **InitSecurityInterface**, the initialization function for the provider. You can use **InitSecurityInterface** to return a pointer to the **SecurityFunctionTable** function. The table contains pointers to SSPI functions implemented by the SSP. A provider may choose not to implement some of the SSPI functions, if the provider does not support the underlying operation.

Before using a specific security package, both client and server must call the **EnumerateSecurityPackages** function to enumerate the security packages available from the security provider. Applications usually know which provider to use and specify it by name in a call to the **AcquireCredentialsHandle** function. **EnumerateSecurityPackages** returns an array of **SECPKGINFO** structures that describe attributes of the available security packages. A client or server can also use the **QuerySecurityPackageInfo** function to determine the attributes of the specified security package.

The following code example shows how to initialize the SSPI.

```
BOOL InitSspi (HINSTANCE DllHandle)
{
  DWORD dwIndex,
        dwNumOfPkgs,
        dwPkgToUse;
  TCHAR szError[100];
```

```
INIT_SECURITY_INTERFACE InitSecurityInterface;
PSecurityFunctionTable pSecurityInterface = NULL;
PSecPkgInfo pSecurityPackages = NULL;
SECURITY_STATUS status;
ULONG ulCapabilities;

// Get the address of the InitSecurityInterface function.
InitSecurityInterface = (INIT_SECURITY_INTERFACE) GetProcAddress (
                                        DllHandle,
                                        TEXT("InitSecurityInterfaceW"));

if (!InitSecurityInterface)
{
  wsprintf (szError,
            TEXT("Failed in getting the function address, Error: %x"),
            GetLastError ());
  return FALSE;
}

// Use InitSecurityInterface to get the function table.
pSecurityInterface = (*InitSecurityInterface)();

if (!pSecurityInterface)
{
  wsprintf (szError,
            TEXT("Failed in getting the function table, Error: %x"),
            GetLastError ());
  return FALSE;
}

if (!(pSecurityInterface->EnumerateSecurityPackages))
{
  wsprintf (szError,
            TEXT("Failed in getting the function table, Error: %x"),
            GetLastError ());
  return FALSE;
}

// Retrieve security packages supported by the provider.
status = (*pSecurityInterface->EnumerateSecurityPackages)(
                                                    &dwNumOfPkgs,
                                                    &pSecurityPackages);
if (status != SEC_E_OK)
{
  wsprintf (szError,
            TEXT("Failed in retrieving security packages, Error: %x"),
            GetLastError ());
  return FALSE;
}
```

```
    // Initialize dwPkgToUse.
    dwPkgToUse = -1;

    // Assume the application requires integrity, privacy, and
    // impersonation on messages.
    ulCapabilities = SECPKG_FLAG_INTEGRITY | SECPKG_FLAG_PRIVACY |
                     SECPKG_FLAG_IMPERSONATION;

    // Determine which package should be used.
    for (dwIndex = 0; dwIndex < dwNumOfPkgs; dwIndex++)
    {
      if ((pSecurityPackages[dwIndex].fCapabilities & ulCapabilities) ==
          ulCapabilities)
      {
        dwPkgToUse = dwIndex;
        break;
      }
    }

    if (!AuthConn (pSecurityInterface, pSecurityPackages, dwPkgToUse))
    {
      MessageBox (NULL,
                  TEXT("Failed in authenticating a connection."),
                  TEXT("Error"),
                  MB_OK);
      return FALSE;
    }

    return TRUE;
}
```

Authenticating a Connection

In a typical client/server application protocol, the server waits for the client to connect and request service. Upon connection, the server must be able to authenticate the client and the client must be able to authenticate the server. The protocol used to establish an authentic connection involves the exchange of one or more security tokens between the client and server SSP. The tokens are sent as messages by the client and the server and include protocol-specific data.

When a message is received, the application protocol reads the token from the message and passes it to its security provider to ensure authentication is complete or to determine if further exchange of tokens is required. The client and the server continue to exchange messages until either the authentication is successfully established or an error occurs.

To begin the authentication process, the client needs to obtain an outbound credentials handle so that it can send an authentication request to the server. This is accomplished by calling the **AcquireCredentialsHandle** function.

Using a reference to the **SecurityFunctionTable** function initialized during the SSPI initialization procedure, the following code example shows how the client obtains an outbound credentials handle.

```
// Get an outbound credentials handle.
status = (*pSecurityInterface->AcquireCredentialsHandle)(
                                    NULL,
                ,                   pSecurityPackages[dwPkgToUse].Name,
                                    SECPKG_CRED_OUTBOUND,
                                    NULL,
                                    NULL,
                                    NULL,
                                    NULL,
                                    &hCredential,
                                    &tsExpiry);
```

When the client accesses credential data, it starts the authentication protocol to establish a connection with the server. The client calls the **InitializeSecurityContext** function to obtain a security token to send a connection request message to the server.

The following code example shows how the client calls **InitializeSecurityContext**.

```
// Initialize the OutSecBuffer structure.
OutSecBuffer.cbBuffer = BUFFERLEN;
OutSecBuffer.BufferType = SECBUFFER_TOKEN;
OutSecBuffer.pvBuffer = pszOutBuffer;

// Initialize the OutBufferDesc structure.
OutBufferDesc.ulVersion = 0;
OutBufferDesc.cBuffers = 1;
OutBufferDesc.pBuffers = &OutSecBuffer;

ulContextReq = ISC_REQ_MUTUAL_AUTH | ISC_REQ_CONNECTION |
               ISC_REQ_SEQUENCE_DETECT | ISC_REQ_REPLAY_DETECT |
               ISC_REQ_CONFIDENTIALITY | ISC_REQ_ALLOCATE_MEMORY;

// Assign the target server name.
// wcscpy (szTargetName, TEXT("..."));
```

```
// Get the authentication token from the security package
// to pass to the server to request an authenticated token.
status = (*pSecurityInterface->InitializeSecurityContext)(
                                        &hCredential,
                                        NULL,
                                        szTargetName,
                                        ulContextReq,
                                        0,
                                        SECURITY_NATIVE_DREP,
                                        NULL,
                                        0,
                                        &hNewContext,
                                        &OutBufferDesc,
                                        &ulContextAttributes,
                                        &tsExpiry);
```

The client uses the security token data received in the output buffer descriptor to generate a message to send to the server. The construction of the message, in terms of placement of various buffers, is decided by the application protocol and is adhered to by both client and server.

Most SSPI functions have variable length arguments that enable an application to provide message data to a security package and enable a security package to return security data, such as a token, to the application. These functions use a parameter type, called a buffer descriptor, to define the size and location of the variable length data. For more information on buffers, see Memory Use and Buffers.

When the client sends a message to the server, it checks the return status from the call to **InitializeSecurityContext** to see if authentication is complete. If not, it expects to receive an authentication token from the server in a response message to continue the security protocol. The return status SEC_I_CONTINUE_NEEDED, indicates that the security protocol requires additional authentication messages.

To establish an authenticated connection, the server must also obtain a handle to its credentials by calling the **AcquireCredentialsHandle** function. When received, you should assign a global variable to the handle to use for the duration of the server process.

When the server receives a connection request from the client, it calls the **AcceptSecurityContext** function, passing the data that it received from the client as input. The server initializes the security buffer descriptors to refer to specific sections of the data, rather than copying the data to an alternate buffer.

The server checks the return status and output buffer descriptor to ensure that no errors exist. If errors exist, it rejects the connection request. If no errors exist, it checks the output buffer for data. If data is present in the buffer, the server bundles the data according to the application protocol and places it into a response message to the client.

If the return status is SEC_I_CONTINUE_NEEDED or SEC_I_COMPLETE_AND_CONTINUE, another message exchange with the client is required. Otherwise, authentication is complete. If authentication must continue, the server waits for the client to respond with another message. Place a time out on this waiting period in case the client does not respond with a message. This will avoid the possibility of hanging this and, subsequently, all server threads.

When the client receives a reply from the server, it deconstructs the message and calls **InitializeSecurityContext** again; be sure to use the continue status from the previous call. Depending on the security package and the context requirements, the transmission of messages between the client and the server can go on indefinitely, but is usually limited to three attempts.

Context Semantics

The SSPI supports three types of security contexts: connection, datagram, and stream contexts.

With a connection context, the caller of the function is responsible for formatting messages. The caller also relies on the security provider to authenticate connections and to ensure the integrity of specific parts of the message. Most context options are available to connection contexts. These options include mutual authentication, replay detection, and sequence detection. A security package sets the SECPKG_FLAG_CONNECTION flag to indicate that it supports connection semantics.

A datagram context, or connectionless context, has slightly different semantics from a connection context. A connectionless context implies that the server has no way of determining when the client has shut down or otherwise terminated the connection. In other words, no termination notice is passed from the transport application to the server, as would occur in a connection context. A security package sets the SECPKG_FLAG_DATAGRAM flag to indicate that it supports datagram semantics. If a client specifies the ISC_REQ_DATAGRAM flag in its call to the **InitializeSecurityContext** function, the following characteristics apply.

- The security package does not produce an authentication binary large object (BLOB) on the first call to **InitializeSecurityContext**. However, the client can immediately use the returned security context in a call to the **MakeSignature** function to generate a message signature.
- The security package must enable the context to be reestablished multiple times to enable the server to drop the connection without notice. This also implies that any keys used in the **MakeSignature** and **VerifySignature** functions can be reset to a consistent state. For more information on key states, see Cryptography.
- The security package must enable the caller to specify sequence data, and must enable the receiver to return that same sequence data back to the caller. This is not exclusive of any sequence data maintained by the package.

A stream context is different from a connection context and a datagram context. A stream context handles secure stream protocols. A security package that supports stream contexts has the following characteristics:

- The package sets the SECPKG_FLAG_STREAM flag to indicate that it supports stream semantics, just as it would set a flag to indicate support for connection and datagram semantics.
- A transport application requests stream semantics by setting the ISC_REQ_STREAM and ASC_REQ_STREAM flags in the calls to the **InitializeSecurityContext** and **AcceptSecurityContext** functions.
- The application calls the **QueryContextAttributes** function with a **SecPkgContext_StreamSizes** structure to query the security context for the number of buffers to provide, and the sizes to reserve for headers or trailers.
- The application provides extra buffer descriptors during actual data processing.

By specifying stream semantics, the caller indicates it will perform extra operations so that the security provider can block messages. These extra operations include passing a list of buffers when the **MakeSignature** and **VerifySignature** functions are called. When a message is received from a stream-oriented channel, the caller passes a buffer. The following table shows the buffers.

Buffer	Length	Buffer type
1	Message length	SECBUFFER_DATA
2	0	SECBUFFER_EMPTY
3	0	SECBUFFER_EMPTY
4	0	SECBUFFER_EMPTY
5	0	SECBUFFER_EMPTY

The security package then authenticates the BLOB. The following table shows what the buffer list looks like, if the function returns successfully.

Buffer	Length	Buffer type
1	Header length	SECBUFFER_STREAM_HEADER
2	Data length	SECBUFFER_DATA
3	Trailer length	SECBUFFER_STREAM_TRAILER
4	0	SECBUFFER_EMPTY
5	0	SECBUFFER_EMPTY

The following table shows an alternate return value for buffer 4.

Buffer	Length	Buffer type
4	*x*	SECBUFFER_EXTRA

The buffer listed in the previous table indicates data in this buffer is part of the next record, and has not yet been processed.

Conversely, the following table shows what the returned buffer list would look like if the message function returns the SEC_E_INCOMPLETE_MESSAGE error message.

Buffer	Length	Buffer type
1	*x*	SECBUFFER_MISSING

The buffer described in this table indicates that more data is needed to process the record. Unlike most errors returned from a message function, this buffer type does not indicate that the context has been compromised. Security providers must not update their state in this condition.

Similarly, on the sender's side of the communication, the caller can call **MakeSignature**, in which case the security package may need to reallocate the buffer. The following table shows the buffer list that the caller can provide to be more efficient.

Buffer	Length	Buffer type
1	Header length	SECBUFFER_STREAM_HEADER
2	Data length	SECBUFFER_DATA
3	Trailer Llength	SECBUFFER_STREAM_TRAILER

Using a buffer, like the one described in the previous table, enables the caller to use buffers more efficiently. By calling the **QueryContextAttributes** function to determine the amount of space to reserve before calling **MakeSignature**, the operation is more efficient for the application and the security package.

Context Requirements

Context requirements are expressed as a combination of bit flags passed to either the **InitializeSecurityContext** or the **AcceptSecurityContext** function. These flags affect the context many ways: not all flags apply to all contexts, some flags are valid only for the server, and other flags are valid only for the client.

The caller uses the *fContextReq* parameter of the **InitializeSecurityContext** or the **AcceptSecurityContext** function to specify a set of flags that indicate the required capabilities. When the function returns, the *pfContextAttr* parameter indicates the attributes of the established context. The caller is responsible for determining if the final context attributes are acceptable. For example, if the caller requested mutual authentication, but the security package indicates that it did not perform such authentication, the caller must decide whether to cancel the context or continue without authentication.

The following table shows the various context requirements.

Type	Description
DELEGATE	Indicates that the server in the transport application requires simple delegation rights, that is, impersonation of the client on the node at which the server is executing.
MUTUAL_AUTH	Indicates that both client and server must authenticate the peer identity.
REPLAY_DETECT	Indicates that the context should be established to enable detection of replayed packets later through the message support functions: **MakeSignature** and **VerifySignature**. This context implies INTEGRITY.
SEQUENCE_DETECT	Indicates that the context should be established to enable detection of out-of-order delivery of packets later through the message support functions. This context implies INTEGRITY.
CONFIDENTIALITY	Indicates that the context should be established to protect data while in transit. This context type is reserved for future use.
USE_SESSION_KEY	Indicates that a new session key should be negotiated.
PROMPT_FOR_CREDS	Indicates that the security package should prompt the user for the appropriate credentials to use, if possible and if the client is an interactive user.

Type	Description
USE_SUPPLIED_CREDS	Indicates that package-specific credential data is available in the input buffer. The security package should use these credentials to authenticate the connection.
ALLOCATE_MEMORY	Indicates that the security package should allocate memory. The caller must eventually call the **FreeContextBuffer** function to free memory allocated by the security package.
USE_DCE_STYLE	Indicates that the caller expects a three transfer authentication transaction.
DATAGRAM	Indicates datagram semantics should be used.
CONNECTION	Indicates connection semantics should be used.
STREAM	Indicates stream semantics should be used.
EXTENDED_ERROR	Indicates that if the context fails, the application generates an error reply message for the peer.
INTEGRITY	Indicates that buffer integrity can be verified, but no sequencing or reply detection is enabled.

Memory Use and Buffers

Memory is handled through a list of descriptors for the buffers being passed to the functions. Because certain protocols require access to an entire message, the entire message is available. To ensure application integrity, however, you can prohibit a package from modifying an area of a message.

The context functions use the **SECBUFFER** and **SECBUFFERDESC** structures to pass memory buffers. The client creates an array of **SECBUFFER** structures that references only the buffers that the application will be passing to the package. The security package may indicate that it looks at only the security portion of a message, and that the SSPI client need not provide the other portions of the message. By passing only portions of a message instead of an entire message, performance improves.

SECBUFFERDESC is a header that includes a pointer to the array of **SECBUFFER** structures. The following code example shows how the server initializes an array of buffers when it calls the **AcceptSecurityContext** function. The last buffer contains the opaque security token received by the client. The SECBUFFER_READONLY flag is also set.

```
SecBuffer Buffers[3];
SecBufferDesc BufferDesc;

// Set up the buffer descriptors.
BufferDesc.ulVersion = SECBUFFER_VERSION;
BufferDesc.cBuffers = 3;
BufferDesc.pBuffers = &Buffers[0];

Buffers[0].cbBuffer = sizeof (Protocol_Header);
Buffers[0].BufferType = SECBUFFER_READONLY | SECBUFFER_DATA;
Buffers[0].pvBuffer = pHeader;

Buffers[1].cbBuffer = pHeader->MessageSize;
Buffers[1].BufferType = SECBUFFER_DATA;
Buffers[1].pvBuffer = pMessage;

Buffers[2].cbBuffer = pHeader->TrailerSize;
Buffers[2].BufferType = SECBUFFER_READONLY | SECBUFFER_TOKEN;
Buffers[2].pvBuffer = pSecurityTrailer;
```

Securing the Message Exchange

After a security context is established, the application can use the SSPI message support functions to transmit signed messages. The process of signing a message ensures that the message cannot be tampered with.

If an application wants to generate signed messages, the client must set the ISC_REQ_REPLAY_DETECT or ISC_REQ_SEQUENCE_DETECT flag of the context attribute argument when first calling the **InitializeSecurityContext** function.

Signing messages requires that the client and the server service providers establish a common session key used to sign messages on the sender side of the communication and verify messages on the receiver side of the communication. Algorithms used in message signatures are known only to the security package.

After an authenticated connection has been established, the client or server can pass the security context and a message to the **MakeSignature** function to generate a secure signature. **MakeSignature** generates a *checksum* of the message and also provides sequencing data to prevent the message from being modified in transit. A checksum is a calculated value used to test data for the presence of errors that can occur when data is transmitted.

The following code example shows how to implement **MakeSignature** and how to append the signature to the message so that the receiver can extract it when the message is received.

```
SecBuffer OutSecBuffer[2];
SecBufferDesc OutBufferDesc;

// Set up the buffer descriptors.
OutBufferDesc.ulVersion = 0;
OutBufferDesc.cBuffers = 2;
OutBufferDesc.pBuffers = &OutSecBuffer[0];

OutSecBuffer[0].cbBuffer = MessageLen;
OutSecBuffer[0].BufferType = SECBUFFER_DATA | SECBUFFER_READONLY;
OutSecBuffer[0].pvBuffer = pMessage;

OutSecBuffer[1].cbBuffer = SignatureLen;
OutSecBuffer[1].BufferType = SECBUFFER_EMPTY;
OutSecBuffer[1].pvBuffer = (pMessage + MessageLen);

// Call MakeSignature to get it signed.
status = (*pSecurityInterface->MakeSignature) (&hContext,
                                               0,
                                               &BufferDesc,
                                               ulMessageSeqNum);
```

The sender then uses the buffer descriptor, including the signature, to construct a message to send to the receiver. The receiver takes the message and disassembles it to recreate the buffer descriptor. The receiver then calls the **VerifySignature** function to verify that the message received is correct according to the data in the signature.

The following code example shows how to implement **VerifySignature**.

```
SecBuffer InSecBuffer[2];
SecBufferDesc InBufferDesc;

// Set up the buffer descriptors.
InBufferDesc.ulVersion = 0;
InBufferDesc.cBuffers = 2;
InBufferDesc.pBuffers = &InSecBuffer[0];

InSecBuffer[0].cbBuffer = MessageLen;
InSecBuffer[0].BufferType = SECBUFFER_DATA | SECBUFFER_READONLY;
InSecBuffer[0].pvBuffer = pMessage;
```

```
InSecBuffer[1].cbBuffer = SignatureLen;
InSecBuffer[1].BufferType = SECBUFFER_TOKEN;
InSecBuffer[1].pvBuffer = (pMessage + MessageLen);

// Call VerifySignature to verify the message signature.
status = (*pSecurityInterface->VerifySignature) (&hContext,
                                                 &BufferDesc,
                                                 ulMessageSeqNum,
                                                 &ulQualityProtection);
```

Deleting a Security Context

To delete a security context after the client and the server have finished communicating, the client and the server call the **DeleteSecurityContext** function with their respective context handles. The client should also call the **FreeCredentialsHandle** function when it has finished communicating with any server, or has finished using the additional credentials passed to the **AcquireCredentialsHandle** function. The server should then call **DeleteSecurityContext** when it is ready to shut down, but before unloading the DLL.

Calling the Windows NT LAN Manager Security Support Provider

Windows CE application can interact with Microsoft Windows NT workstations that are running the LAN Manager Security Support Provider service (Windows NT LMSSP). The Windows NT LMSSP is based on the Windows NTLM (Windows NT LAN Manager) authentication protocol. All references to server in this section refer to a Windows-based desktop platform server.

Client Initialization

This section describes how the client of a transport application connects to a Windows-based desktop platform server, using Windows NT LMSSP authentication.

To initialize, the application client calls the **InitSecurityInterface** function. If the client is binding directly to Secur32.dll, it can discard the returned function table; otherwise, it must use the function table to make subsequent calls to the security provider functions. The client must also call the **QuerySecurityPackageInfo** function to get the maximum security buffer size.

To make the connection, the application calls the **AcquireCredentialsHandle** function, using the **SEC_WINNT_AUTH_IDENTITY** structure to specify the credentials. The application must save the credential handle for use when calling the **InitializeSecurityContext** function, but it can discard the other parameters after the call to **AcquireCredentialsHandle**. The following code example shows how to make a connection.

```
SEC_WINNT_AUTH_IDENTITY AdditionalCredentials;
SECURITY_STATUS status;
CredHandle hCredential;
TimeStamp tsExpiry;
BOOL bSupplyCredentials;

// If there are additional credentials stored in lpszUserName,
// lpszDomainName, and lpszPassword, fill them in here.
AdditionalCredentials.Flags = SEC_WINNT_AUTH_IDENTITY_UNICODE;

if (lpszUserName != NULL)
{
  AdditionalCredentials.User = lpszUserName;
  AdditionalCredentials.UserLength = wcslen (lpszUserName);
}

if (lpszDomainName != NULL)
{
  AdditionalCredentials.User = lpszDomainName;
  AdditionalCredentials.UserLength = wcslen (lpszDomainName);
}

if (lpszPassword != NULL)
{
  AdditionalCredentials.User = lpszPassword;
  AdditionalCredentials.UserLength = wcslen (lpszPassword);
}

status = AcquireCredentialsHandle (
             NULL,                    // No principal name
             TEXT("NTLM"),            // Package name
             SECPKG_CRED_OUTBOUND,    // Credential use flag
             NULL,                    // No logon identifier
             bSupplyCredentials ?  &AdditionalCredentials : NULL,
                                      // Package-specific data
             NULL,                    // No GetKey function
             NULL,                    // No GetKey function argument
             &hCredential,            // Receives the new credential
             &tsExpiry);              // Receives the expiration
                                      // time of the credential
```

The credential handle does not expire, so the client can ignore the expiration time for this security package.

Next, the application calls **InitializeSecurityContext** to start setting up the security context. The following code example shows how to start up the security context.

```
SecBufferDesc OutputBufferDescriptor;
SecBuffer OutputSecurityToken;
ULONG ulContextRequirements,
      ulContextAttributes;
SECURITY_STATUS status;
CredHandle hCredential;
CtxtHandle hNewContext;
TimeStamp tsExpiry;

// Build the output buffer descriptor.
OutputBufferDescriptor.cBuffers = 1;
OutputBufferDescriptor.pBuffers = &OutputSecurityToken;
OutputBufferDescriptor.ulVersion = SECBUFFER_VERSION;

OutputSecurityToken.BufferType = SECBUFFER_TOKEN;
OutputSecurityToken.cbBuffer = pPackageInfo->cbMaxToken;
OutputSecurityToken.pvBuffer =
                        LocalAlloc (0, OutputSecurityToken.cbBuffer);

// Insert code here to check for memory allocation failure.
// ...

// Compute context requirements. For message integrity, request
// replay or sequence detection. For message encryption,
// request confidentiality.
ulContextRequirements = ISC_REQ_REPLAY_DETECT;

// Assume that szTargetName is the name of the target server, or NULL.
// Call the InitializeSecurityContext function.
status = InitializeSecurityContext (
             &hCredential,
             NULL,                    // No context handle
             szTargetName,            // Target name, if available
             ulContextRequirements,
             0,                       // Reserved parameter
             SECURITY_NATIVE_DREP,    // Target data representation
             NULL,                    // No input buffer
             0,                       // Reserved parameter
             &hNewContext,            // Receives new context handle
             &OutputBufferDescriptor,// Receives output security token
             &ulContextAttributes,    // Receives context attributes
             &tsExpiry);              // Receives context expiration
                                      // time
```

InitializeSecurityContext returns SEC_I_CONTINUE_NEEDED on success, or an error value on failure. If the function is successful, the application passes the token buffer to the server. The token buffer is stored in the *pvBuffer* member of the **OUTPUTSECURITYTOKEN** structure. The *cbBuffer* member of the structure specifies the buffer length. The token buffer is then sent to the server. The server sends the output back to the client. The client then calls **InitializeSecurityContext** again.

Windows NT LMSSP Server Authentication

When the application makes a second call to the **InitializeSecurityContext** function, the parameters are similar to the first call. The following code example, showing the second call to **InitializeSecurityContext**, assumes that the security buffer returned from the server is in *InputSecurityBuffer* and the length of that buffer is in *InputSecurityBufferSize.*

```
SecBufferDesc OutputBufferDescriptor,
              InputBufferDescriptor;
SecBuffer OutputSecurityToken,
          InputSecurityToken;
ULONG ulContextAttributes;
TimeStamp tsExpiry;

// Build the input buffer descriptor.
InputBufferDescriptor.cBuffers = 1;
InputBufferDescriptor.pBuffers = &InputSecurityToken;
InputBufferDescriptor.ulVersion = SECBUFFER_VERSION;

InputSecurityToken.BufferType = SECBUFFER_TOKEN;
InputSecurityToken.cbBuffer = InputSecurityBufferSize;
InputSecurityToken.pvBuffer = InputSecurityBuffer;

// Build the output buffer descriptor.
OutputBufferDescriptor.cBuffers = 1;
OutputBufferDescriptor.pBuffers = &OutputSecurityToken;
OutputBufferDescriptor.ulVersion = SECBUFFER_VERSION;

OutputSecurityToken.BufferType = SECBUFFER_TOKEN;
OutputSecurityToken.cbBuffer = pPackageInfo->cbMaxToken;
OutputSecurityToken.pvBuffer =
                        LocalAlloc (0, OutputSecurityToken.cbBuffer);

// Insert code here to check for memory allocation failure.
// ...
```

```
// Ignore the pszTargetName and fContextReq parameters on this
// call. This time, instead of passing NULL for phContext, pass
// the context handle received on the first call.
status = InitializeSecurityContext (
            &hCredential,
            &hContext,
            NULL,                    // No target name
            0,                       // No context requirements
            0,                       // Reserved parameter
            SECURITY_NATIVE_DREP,    // Target data representation
            &InputBufferDescriptor, // Input buffer
            0,                       // Reserved parameter
            &hContext,               // Same as the old context
            &OutputBufferDescriptor,// Receives output security token
            &ulContextAttributes,    // Receives context attributes
            &tsExpiry);              // Receives context expiration
                                     // time
```

If the **InitializeSecurityContext** call is successful, it returns SEC_E_OK, and the application transmits the output security buffer and buffer length to the server, as it did after the first call to **InitializeSecurityContext**. If it fails, an error value returns.

When the application has finished setting up the security context, the application can begin using the security context in calls to the **MakeSignature** and **VerifySignature** functions to make and verify message signatures, even though the server has not yet finished authenticating the client.

Using a Security Context

Both the client and the server side of a transport application can use the **MakeSignature** function to generate a signed message to send to the other side. The receiving side then uses the **VerifySignature** function to verify that a signature matches the received message. If the application wants to generate signed messages, the client must have specified the ISC_REQ_REPLAY_DETECT or ISC_REQ_SEQUENCE_DETECT flag in its first call to the **InitializeSecurityContext** function.

When the client calls **MakeSignature** or **VerifySignature**, it uses the *ClientContext* handle that it obtained from its first call to **InitializeSecurityContext**.

The following code example shows how the client side generates a signed message to send to the server. Before calling **MakeSignature**, the client calls the **QueryContextAttributes** function with a **SECPKGCONTEXT_SIZES** structure to determine the length of the buffer needed to hold the message signature. If the *cbMaxSignature* parameter is zero, the security package does not support signing messages; otherwise, this parameter indicates the size of the buffer to allocate to receive the signature.

```
SecPkgContext_Sizes ContextSizes;

status = QueryContextAttributes (&hContext,
                                 SECPKG_ATTR_SIZES,
                                 &ContextSizes);

// Assume the message is in the variable MessageBuffer, and
// its length is in MessageBufferSize. Build up the buffer descriptors
// to pass to the MakeSignature call.
SecBufferDesc InputBufferDescriptor;
SecBuffer InputSecurityToken[2];

// Build the input buffer descriptor.
InputBufferDescriptor.cBuffers = 2;
InputBufferDescriptor.pBuffers = InputSecurityToken;
InputBufferDescriptor.ulVersion = SECBUFFER_VERSION;

// Build a security buffer for the message. If the SECBUFFER_READONLY
// attribute is added, this buffer would not get signed.
InputSecurityToken[0].BufferType = SECBUFFER_DATA;
InputSecurityToken[0].cbBuffer = MessageBufferSize;
InputSecurityToken[0].pvBuffer = MessageBuffer;

// Allocate and build a security buffer for the message signature.
InputSecurityToken[1].BufferType = SECBUFFER_TOKEN;
InputSecurityToken[1].cbBuffer = ContextSizes.cbMaxSignature;
InputSecurityToken[1].pvBuffer =
                        LocalAlloc (0, ContextSizes.cbMaxSignature);

// Insert code here to check for memory allocation failure.
// ...

// Call MakeSignature now. Specify the sequence number;
// Windows NTLM provides one. The quality of service is ignored.
status = MakeSignature (
             &hContext,
             0,                      // No quality of service
             &InputBufferDescriptor, // Input message descriptor
             0);                     // No sequence number
```

MakeSignature returns successfully if the context was set up to enable signing messages and the input buffer descriptor is correctly formatted. If the function is successful, the application sends the message buffer and its size, along with the signature buffer and its size, to the server.

To delete the security contexts after the client and server have finished communicating, both sides can call the **DeleteSecurityContext** function with their respective context handles. The client should also call the **FreeCredentialsHandle** function when it has finished communicating with any server or has finished using the additional credentials passed to the **AcquireCredentialsHandle** function. For more information on writing the server side of the Windows NT LMSSP transport application, see the Microsoft Platform SDK.

SSPI Sample Application

The following is an example of an SSPI application.

```
#include <windows.h>
#include <sspi.h>
#include <issperr.h>
#include <winsock.h>

#define BUFFERLEN     16384

BOOL InitSspi (HINSTANCE);
BOOL AuthConn (PSecurityFunctionTable, PSecPkgInfo, DWORD);

/*****************************************************************************

FUNCTION:
  WinMain

PURPOSE:
  Called by the system as the initial entry point for this Windows
  CE-based application.

*****************************************************************************/
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPTSTR lpCmdLine, int nCmdShow)
{
  HINSTANCE DllHandle;
  TCHAR szError[100];

  // Load the security provider DLL.
  DllHandle = LoadLibrary (TEXT("secur32.dll"));
```

```
  if (!DllHandle)
  {
    wsprintf (szError,
              TEXT("Failed in loading secur32.dll, Error: %x"),
              GetLastError ());
    return 0;
  }

  if (!InitSspi (DllHandle))
  {
    MessageBox (NULL,
                TEXT("Failed in initializing the SSPI."),
                TEXT("Error"),
                MB_OK);
    return 0;
  }

  return 1;
}

/*****************************************************************************

FUNCTION:
  InitSspi

PURPOSE:
  Initializes Security Support Provider Interface.

*****************************************************************************/
BOOL InitSspi (HINSTANCE DllHandle)
{
  DWORD dwIndex,
        dwNumOfPkgs,
        dwPkgToUse;
  TCHAR szError[100];

  INIT_SECURITY_INTERFACE InitSecurityInterface;
  PSecurityFunctionTable pSecurityInterface = NULL;
  PSecPkgInfo pSecurityPackages = NULL;
  SECURITY_STATUS status;
  ULONG ulCapabilities;

  // Get the address of the function InitSecurityInterface.
  InitSecurityInterface = (INIT_SECURITY_INTERFACE) GetProcAddress (
                                          DllHandle,
                                          TEXT("InitSecurityInterfaceW"));
```

```
if (!InitSecurityInterface)
{
  wsprintf (szError,
            TEXT("Failed in getting the function address, Error: %x"),
            GetLastError ());
  return FALSE;
}

// Use InitSecurityInterface to get the function table.
pSecurityInterface = (*InitSecurityInterface)();

if (!pSecurityInterface)
{
  wsprintf (szError,
            TEXT("Failed in getting the function table, Error: %x"),
            GetLastError ());
  return FALSE;
}

if (!(pSecurityInterface->EnumerateSecurityPackages))
{
  wsprintf (szError,
            TEXT("Failed in getting the function table, Error: %x"),
            GetLastError ());
  return FALSE;
}

// Retrieve the security packages supported by the provider.
status = (*pSecurityInterface->EnumerateSecurityPackages)(
                                                    &dwNumOfPkgs,
                                                    &pSecurityPackages);
if (status != SEC_E_OK)
{
  wsprintf (szError,
            TEXT("Failed in retrieving security packages, Error: %x"),
            GetLastError ());
  return FALSE;
}

// Initialize dwPkgToUse.
dwPkgToUse = -1;

// Assume the application needs integrity, privacy, and impersonation
// on messages.
ulCapabilities = SECPKG_FLAG_INTEGRITY | SECPKG_FLAG_PRIVACY |
                 SECPKG_FLAG_IMPERSONATION;
```

```
  // Determine which package should be used.
  for (dwIndex = 0; dwIndex < dwNumOfPkgs; dwIndex++)
  {
    if ((pSecurityPackages[dwIndex].fCapabilities & ulCapabilities) ==
        ulCapabilities)
    {
      dwPkgToUse = dwIndex;
      break;
    }
  }

  if (!AuthConn (pSecurityInterface, pSecurityPackages, dwPkgToUse))
  {
    MessageBox (NULL,
                TEXT("Failed in authenticating a connection."),
                TEXT("Error"),
                MB_OK);
    return FALSE;
  }

  return TRUE;
}

/*****************************************************************************

FUNCTION:
  AuthConn

PURPOSE:
  Authenticates a connection.

*****************************************************************************/
BOOL AuthConn (PSecurityFunctionTable pSecurityInterface,
               PSecPkgInfo pSecurityPackages,
               DWORD dwPkgToUse)
{
  BOOL bReturn = FALSE;           // Return value of the function
  TCHAR szError[100],             // String for the error message
        szTargetName[100];        // Target name
  LPSTR pszOutBuffer = NULL;      // Used in security data xfr
  ULONG ulContextReq,             // Required context attributes
        ulContextAttributes;      // Receives attributes of the context
  TimeStamp tsExpiry;             // Returned credentials' life time
  SECURITY_STATUS status;         // Return codes
  CredHandle hCredential;         // Handle to the credential
  CtxtHandle hNewContext;         // Handle to the security context
  SecBuffer OutSecBuffer;         // Output buffer
  SecBufferDesc OutBufferDesc;    // Output buffer descriptor
```

```
// Check if the pointer to SecurityFunctionTable is valid.
if (!pSecurityInterface)
  goto exit;

// Allocate buffer memory for pszOutBuffer.
if (!(pszOutBuffer = new char[BUFFERLEN]))
  goto exit;

// Acquire an outbound credential handle.
status = (*pSecurityInterface->AcquireCredentialsHandle)(
                                    NULL,
                                    pSecurityPackages[dwPkgToUse].Name,
                                    SECPKG_CRED_OUTBOUND,
                                    NULL,
                                    NULL,
                                    NULL,
                                    NULL,
                                    &hCredential,
                                    &tsExpiry);
if (status != SEC_E_OK)
{
  wsprintf (szError,
            TEXT("Failed in acquiring the credential handle: %x"),
            status);
  goto exit;
}

// Initialize the OutSecBuffer structure.
OutSecBuffer.cbBuffer = BUFFERLEN;
OutSecBuffer.BufferType = SECBUFFER_TOKEN;
OutSecBuffer.pvBuffer = pszOutBuffer;

// Initialize the OutBufferDesc structure.
OutBufferDesc.ulVersion = 0;
OutBufferDesc.cBuffers = 1;
OutBufferDesc.pBuffers = &OutSecBuffer;

ulContextReq = ISC_REQ_MUTUAL_AUTH | ISC_REQ_CONNECTION |
               ISC_REQ_SEQUENCE_DETECT | ISC_REQ_REPLAY_DETECT |
               ISC_REQ_CONFIDENTIALITY | ISC_REQ_ALLOCATE_MEMORY;

// Assign the target (server) name.
// wcscpy (szTargetName, TEXT("..."));

// Get the authentication token from the security package to send to
// the server to request an authenticated token.
```

```
status = (*pSecurityInterface->InitializeSecurityContext)(
                                                &hCredential,
                                                NULL,
                                                szTargetName,
                                                ulContextReq,
                                                0,
                                                SECURITY_NATIVE_DREP,
                                                NULL,
                                                0,
                                                &hNewContext,
                                                &OutBufferDesc,
                                                &ulContextAttributes,
                                                &tsExpiry);

if (status == SEC_I_CONTINUE_NEEDED)
{
  SOCKET Socket = INVALID_SOCKET;          // Server socket

  // Add code here to connect to server. Get the server socket.
  // ...

  // Send hCredential to server
  if (send (Socket, (const char *)OutSecBuffer.pvBuffer,
            OutSecBuffer.cbBuffer, 0) == SOCKET_ERROR)
  {
    wsprintf (szError,
              TEXT("Failed in sending hCredential to the server: %d"),
              WSAGetLastError ());
    goto exit;
  }

  // Add code here to make the second call to the
  // InitializeSecurityContext function
..
  // ...
}
else
{
  if (status != SEC_E_OK)
  {
    wsprintf (szError,
              TEXT("Failed in initiating the outbound security ")
              TEXT("context: %x"),
              status);
    goto exit;
  }
}
```

```
    bReturn = TRUE;

exit:

    if (pszOutBuffer)
      delete[] pszOutBuffer;

    if (pSecurityInterface)
    {
      (*pSecurityInterface->FreeCredentialHandle)(&hCredential);
      (*pSecurityInterface->DeleteSecurityContext)(&hNewContext);
      (*pSecurityInterface->FreeContextBuffer)(&OutBufferDesc);
    }

    return bReturn;
}
```

CHAPTER 9

Cryptography

Cryptography provides a way to distribute files in secret code, or *cipher*, so they can only be read by intended recipients. Cryptography maintains secrecy and ensures data integrity to achieve secure communications in your Windows CE–based application.

The following aspects of cryptography are discussed:

- Encryption and decryption
- Windows CE implementation of the Microsoft Cryptographic API (CAPI)
- How to use CAPI

Encryption and Decryption

Encryption is the process of encoding data into cipher, a form that is unreadable without a decoding key. *Decryption* is the reverse process of converting encoded data to its original unencoded, *plaintext*, form. When a user encodes a file, a user cannot read the file without the proper key to decode it. Adding a *digital signature*, a form of personal authentication, ensures that the original message has not been tampered with.

To encode plaintext, an *encryption key* is used to impose an encryption algorithm onto the data. To decode cipher, a user must possess the appropriate *decryption key*. A decryption key consists of a random string of numbers, from 40 to 2,000 bits in length. The key imposes a decryption algorithm onto the data. This decryption algorithm reverses the encryption algorithm, returning the data to plaintext. The longer the encryption key, the more difficult it is to decode. For a 40-bit encryption key, over one trillion possible decryption keys exist.

There are two primary approaches to encryption: *symmetric* and *public-key*. Symmetric encryption is the most common type of encryption and uses the same key for encoding and decoding data. This key is known as a *session key*. Public-key encryption uses two different keys, a public key and a private key. One key encodes the message and the other decodes it. The public key is widely distributed while the private key is kept secret.

Aside from key length and the encryption approach, other factors and variables impact the success of a cryptographic system. For example, different cipher modes can be used to vary the encryption along with *initialization vectors* and *salt values*. *Cipher modes* define the method in which data is encrypted. The *stream cipher mode* encodes data one bit at a time. The *block cipher mode* encodes data one block at a time. Although it tends to execute more slowly than stream cipher, block cipher is generally more secure. Within block ciphers, there are four encryption modes: electronic codebook (ECB), cipher block chaining (CBC), cipher feedback mode (CFB), and output feedback mode (OFB). For more information on these modes, see Encrypting and Decrypting Data.

Initialization vectors are random numbers used as starting points when encoding data. Usually, initialization vectors have the same number of bits as the block size and do not require encryption. With initialization vectors, two identical plaintext messages can be encoded with the same key and result in two completely different cipher messages. This variation is done by encrypting each plaintext message with a different initialization vector.

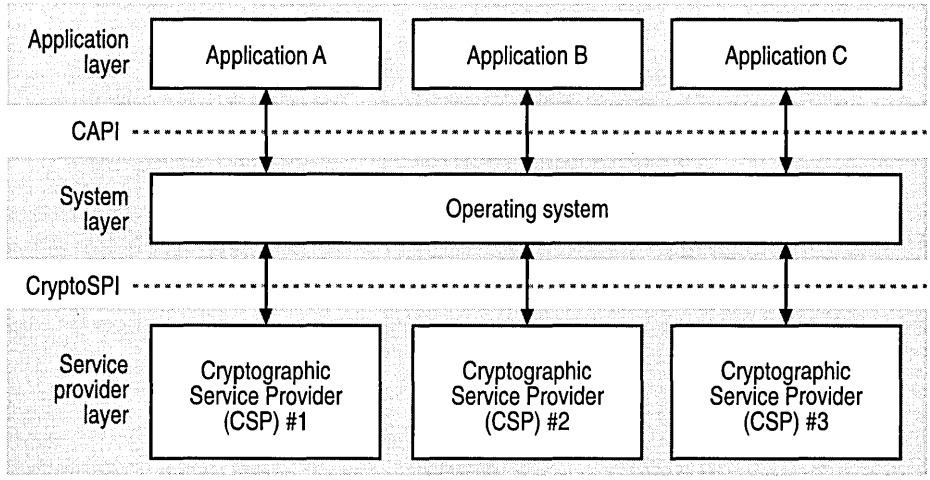
Salt values are most useful when transmitting or storing large numbers of nearly identical packets by using the same encryption key. Typically, two identical packets would encode as two identical cipher packets. However, this would indicate to an eavesdropper that the packets are identical and, thus, the packets could be attacked simultaneously. But if the salt value is changed with every packet sent, a completely different cipher packet is generated, even if the plaintext packets are the same. *Salt values* consist of random numbers and can be transmitted in plaintext form.

In addition to encrypting the data, a user can digitally sign data to enable another user to verify that the data has not been changed since it was signed. The identity of the user that signed the data can also be verified. This digital signature consists of a small amount of binary data, typically less than 256 bytes. A digital signature can be bundled with signed messages or stored separately, depending on the application.

Microsoft Cryptographic System

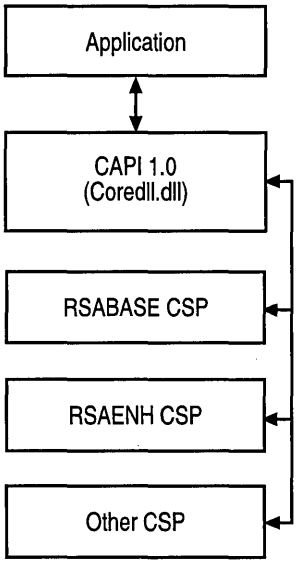
The Microsoft cryptographic system is composed of different components. The three executable portions are the application, the operating system (OS), and the cryptographic service provider (CSP).

Applications communicate with the OS through the cryptographic API (CAPI). The OS communicates with CSPs through the cryptographic service provider interface (CSPI). The following illustration shows these concepts.



All cryptographic operations are performed by independent modules known as *cryptographic service providers* (CSPs). CSPs communicate with applications through Coredll.dll. A CSP is responsible for creating and destroying keys, and using them to perform a variety of cryptographic operations. Each CSP provides a different implementation of the CAPI. Some provide stronger cryptographic algorithms, while others contain hardware components.

The following illustration shows the relationship between applications, Coredll.dll, and the CSPs.



At a minimum, a CSP consists of a dynamic-link library (DLL) and a *signature file*. The signature file ensures that the OS recognizes the CSP. The OS validates this signature periodically to verify that the CSP has not been tampered with.

Each provider has both a name and a type. For example, the name of the CSP currently shipped with Windows CE is Microsoft Base Cryptographic Provider version 1.0, and its type is **PROV_RSA_FULL**. The name of each provider is unique while the provider type is not.

Cryptographic standards are organized into groups known as families. Each family includes a set of data formats and protocols. Even if they use the same algorithm, two families will often use different cipher modes, key lengths, and default modes. In CAPI, each CSP type represents a distinct family.

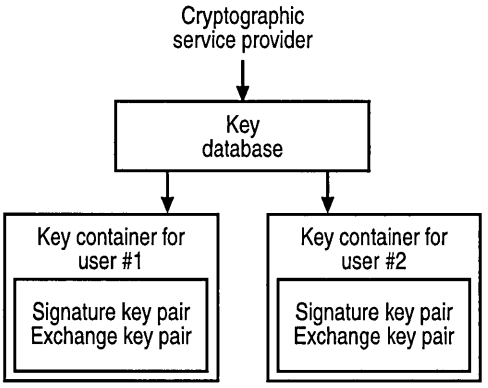
By default, when an application connects to a CSP of a particular type, each CAPI function operates in a way prescribed by the family that corresponds to the CSP type.

The following table shows the items specified by an application's choice of CSP type.

CSP type property	Description
Key exchange algorithm	Specifies one key exchange algorithm. Every CSP of a particular type must implement this algorithm. The only way applications can specify the key exchange algorithm is by selecting a CSP of the appropriate type.
Digital signature algorithm	This is the same as with the key exchange algorithm. Each CSP type specifies one digital signature algorithm.
Key binary large object format	Specifies the format of exported keys. Keys can be exported out of a CSP into a key binary large object format to securely transfer keys between CSPs.
Digital signature format	Prescribes a particular digital signature format. This ensures that a signature produced by a CSP can be verified by any CSP of the same type.
Session key derivation scheme	Specifies the method used to derive session keys
Key length	Specifies the key length
Default modes	Specifies a default mode for various options, such as the block encryption cipher mode or the block encryption padding method

Key Databases

Each CSP has a key database in which it stores its persistent cryptographic keys. Each key database contains one or more *key containers*, each of which contains all the key pairs belonging to a specific user. The following illustration shows the relationship between CSPs, key databases, and key containers.



The CSP stores each key container from session to session, including all of the public and private key pairs that it contains. However, session keys are not preserved from session to session.

Generally, a default key container is created for each user. Default key containers have a default name. An application can create its own key container and key pairs, in which case the key container is given a name by the application.

Key BLOBs

A *key binary large object* (key BLOB) provides a way to store a key outside of the CSP. A key BLOB is used as the medium for securely transferring a key from one provider to another. A key BLOB is fairly secure because it is encrypted with the key exchange public key of the intended recipient. To make it tamperproof, a key is sometimes signed with the key exchange private key of the originating user.

A key BLOB consists of a standard header followed by data that represents the key itself. Key BLOBs exist in three forms: *simple*, *public*, and *private*. A *simple key BLOB*, known as a SIMPLEBLOB, is a session key that has been encoded with the public key exchange key of the destination user. Key exchange keys are used to encode session keys so they can be safely stored and exchanged with other users. This type of key BLOB is used when storing a session key or transmitting a session key to another user. For more information on key exchange, see Exchanging Cryptographic Keys.

A public key BLOB contains the public key portion of a public/private key pair. Unlike simple key BLOBs, these are not encrypted.

A private key BLOB contains one complete public and private key pair. These key BLOBs are used by administrative applications to distribute and transport public and private key pairs. For example, a private key BLOB transports key pairs between a network administrator's computer and a user's computer, or between a user's desktop computer and a laptop computer. These key BLOBs can also be used by advanced applications to store key pairs themselves, rather than relying on the CSP's storage mechanism.

For more information on the formats of these key BLOBs, see the Microsoft Platform SDK.

Microsoft RSA Base Provider

Rsabase.dll, the Microsoft RSA Base Provider included with Windows CE, consists of a software implementation of the **PROV_RSA_FULL** provider type. **PROV_RSA_FULL** supports both digital signatures and data encryption, and is considered to be a general-purpose cryptographic tool. For more information on **PROV_RSA_FULL**, see *Microsoft Cryptographic Service Provider Programmer's Guide*, Microsoft, 1995, and *RSA Laboratories Public-Key Cryptography Standards,* RSA Data Security, November 1993.

Common Encryption Algorithms

The encryption algorithms available to an application depend on the CSP used. Each of the encryption algorithms described here is supplied with the Microsoft RSA Base Provider.

The following table shows several encryption algorithms, along with some performance benchmarks. These figures are for comparison purposes only. Your setup time and encryption speed may vary.

Cipher	Cipher type	Key setup time (microseconds)	Encryption speed (bytes/second)
DES	64-bit block	460	1.1 MB
RC2	64-bit block	40	290 KB
RC4	stream	151	2.4 MB

RC2 and RC4 are variable-key-length ciphers. However, when using CAPI with the Microsoft RSA Base Provider, these key lengths are hard-coded to 40 bits.

Key Length Comparison

When used, the Microsoft Enhanced Cryptographic Provider (Enhanced Provider) provides an application with stronger security than is currently available with the Microsoft Base Cryptographic Provider (Base Provider). This provides users with a greater degree of protection in keeping their sensitive data secure.

The following table shows the default key lengths supported by the Base Provider and the Enhanced Provider for the shown algorithms.

Algorithm	Base Provider	Enhanced Provider
RSA Key Exchange	512-bit	1,024-bit
RSA Signature	512-bit	1,024-bit
RC2	40-bit	128-bit
RC4	40-bit	128-bit
DES	Not supported	56-bit
Triple DES (2-key)	Not supported	112-bit
Triple DES (3-key)	Not supported	168-bit

The Enhanced Provider is backward-compatible with the Base Provider distributed with CAPI version 1.0, with the following exception. For session keys, both CSPs are limited to generating and deriving keys of default key length: 40-bit for the Base Provider, and 128-bit for the Enhanced Provider, which precludes the Enhanced Provider from creating keys with Base Provider–compatible key lengths. However, the Enhanced Provider can import key lengths of any size, up to 128-bits.

Warning If you use the Microsoft RSA Base Provider to create a certification authority, your license to issue certificates is limited to certificates intended for use in the context of your particular application or service.

Using CAPI

Applications can use the Microsoft CAPI for the following tasks:

- Connecting to a CSP
- Generating cryptographic keys
- Exchanging cryptographic keys
- Encrypting and decrypting data
- Creating digital signatures

All data encryption using CAPI is performed with a symmetric algorithm, regardless of which CSP is installed.

Connecting to a CSP

The following table shows functions an application can use to connect to a CSP. These functions also enable applications to choose a specific CSP by name or get one with a needed class of functionality.

Function	Description
CryptAcquireContext	Acquires a handle to the current user's key container within a particular CSP
CryptGetProvParam	Retrieves properties of a CSP
CryptReleaseContext	Releases the handle acquired by **CryptAcquireContext**
CryptSetProvider	Specifies the user default CSP for a particular CSP type
CryptSetProvParam	Specifies properties of a CSP

Each time an application is run, the first CAPI function that an application calls is the **CryptAcquireContext** function. This function returns to the application a handle to a particular CSP. In addition, this handle specifies a particular key container within the CSP. If the CSP has just been installed and no key containers yet exist, **CryptAcquireContext** can also be used to create a new one.

When an application uses **CryptAcquireContext** to obtain a CSP handle, it specifies a CSP type and, optionally, a provider name. If both a type and a name are specified, then the function looks for a CSP with precisely the same type and name, loads it into memory, and returns a handle to the application.

When an application calls **CryptAcquireContext** specifying a CSP type but no provider name, the function tries to find the provider name. It first searches a list of default providers associated with the current user and, if that fails, it searches a list of default CSPs associated with your device.

Once **CryptAcquireContext** has determined the provider name, it searches for the CSP, loads it into memory, and returns a handle to the application.

Generating Cryptographic Keys

The following table shows the functions an application can use to generate cryptographic keys.

Function	Description
CryptDeriveKey	Generates a key derived from a password
CryptGenKey	Generates a random key

Although applications can create unlimited session keys, these keys are not preserved by the CSP from session to session. To preserve a key, export the key out of the CSP and import it into a key BLOB in the application's memory space. For more information on exporting and importing a key, see Exchanging Cryptographic Keys.

Session keys are created using either **CryptGenKey** or **CryptDeriveKey**. When a session key is generated, you must specify the algorithm to use for subsequent encoding and decoding operations. This algorithm must be one of the symmetric algorithms supported by the CSP used.

Because public-key algorithms are slow, it is impractical to use them to encrypt a large amount of data. In practice, symmetric algorithms are used for encoding and decoding large amounts of data, while public-key algorithms are used only to encrypt session keys.

For each user, the CSP usually maintains two public and private key pairs: the key exchange key pair and the digital signature key pair. These keys are maintained from session to session.

There are a number of reasons for having two separate key pairs. For example, some CSPs use one algorithm for key exchange and another for digital signatures. Also, if some data, such as a session key, is both signed and encrypted with the same public key pair, subtle weaknesses can be introduced that make the data vulnerable.

The exchange key and the signature key pairs are created by calling the **CryptGenKey** function and specifying either AT_KEYEXCHANGE or AT_SIGNATURE. The CSP implements these keys in an application-independent manner. Applications are not permitted to know the details about the algorithm used.

Exchanging Cryptographic Keys

This section discusses those situations when you must export keys from the secure environment of the CSP into a key BLOB.

There are two occasions when it is necessary to export keys:

- To save a session key for later use by an application

  For example, if your application has just encoded a database file and you want your application to decode this file at a later time, your application is responsible for storing the encryption key. This is necessary because CSPs do not preserve symmetric keys from session to session.

- To send a key to someone else

  This would be much easier for your application if the respective CSPs could communicate directly, but they cannot. This means that the key has to be exported from your CSP, transmitted by your application to the destination application, and then imported into the destination CSP.

The following table shows functions you can use to create, configure, and destroy cryptographic keys, and to exchange them with other users.

Function	Description
CryptDestroyKey	Destroys a key
CryptExportKey	Exports a key from a CSP into a key BLOB in the application's memory space
CryptGenRandom	Generates random data, usually for salt values
CryptGetKeyParam	Retrieves a key's parameters
CryptGetUserKey	Gets a handle to the key exchange or signature key
CryptImportKey	Imports a key from a key BLOB into a CSP
CryptSetKeyParam	Specifies a key's parameters

Storing Session Keys

To exchange cryptographic keys, it is necessary to first store the session key outside of the CSP because CSPs do not preserve session keys from session to session.

▶ **To store a session key for future use**

1. Create a simple key BLOB using the **CryptExportKey** function.

   This transfers the session key from the CSP to your application's memory space. Specify that your own key exchange public key be used to encrypt the key BLOB.
2. Store the signed key BLOB.
3. Read the key BLOB from storage when you need to use the key.
4. Import the key BLOB into the CSP, using the **CryptImportKey** function.

If you plan to use the session key for encryption at a later time, the key BLOB should be signed with your key exchange key before the key is stored. When you later read the key BLOB, you should validate the signature to make sure that the key BLOB is intact. If these steps are omitted, someone with access to your storage media can create their own session key, encrypt it with your key exchange public key, and substitute it for your key BLOB. You could then unknowingly use their session key to encode files and messages, which the unauthorized user could easily decode.

As an alternative to storing a random session key BLOB, you can use a *derived session key*, which is created from a password using the **CryptDeriveKey** function. In this way, instead of storing a particular derived key, an application can create a derived key as needed by prompting the user for the password.

A stored key BLOB is dependent on the stability of the public and private key pairs stored within the CSP. If these key pairs are lost through a hardware or software incident, for example, you will be unable to decode your key BLOB. Any data that has been encrypted using these keys will also be lost. For this reason, a user should consider using a backup authority when storing long-term archival data.

Using a Backup Authority

A *backup authority* is a trusted application running on a secure computer that provides storage for the session keys of its clients. All session keys stored there are encrypted, in the form of key BLOBs, with the backup authority's public key.

▶ **To store session keys in a backup authority**

1. Encrypt the file as usual.
2. Export the session key used to encrypt the file into a simple key BLOB, specifying that your own key exchange public key be used to encrypt the key BLOB.
3. Store this key BLOB with the encrypted file.
4. Export the session key again, this time specifying that the backup authority's public key be used to encrypt the key BLOB.
5. Send this key BLOB to the backup authority, along with the key's description, serial number, and so on.

If at a later time you lose your key pairs, you can retrieve the session keys from a backup authority, although you will first have to establish your identity with the authority.

Exchanging Public Keys

Exchanging public keys is the first step that two users contemplating encrypted communication need to do. Once this has been done, the users can send encrypted and signed data to each other.

There are two ways to obtain each other's public keys:

- Each user can obtain the other's keys in the form of certificates. This is the most secure way to exchange public keys that does not require user interaction.

- Users can read their public keys to each other over the telephone, use certified mail to send them to each other, or use another tamperproof method. Because your public key is not secret, it does not matter if it is overheard by a third party.

  This method can also be used to validate the public key values that have been exchanged in some other manner.

To exchange public keys, the sender exports his or her public key from the CSP into a public key BLOB, using the **CryptExportKey** function.

When the receiver has received the key BLOB data from the sender, the **CryptImportKey** function is used to import the key BLOB into its own CSP.

Exchanging Session Keys

To send another user an encrypted message, the sender must send the receiver the session key that was used to perform the encryption. There are two ways of doing this:

- The sender and receiver can mutually agree on a session key by exchanging several messages back and forth. The users can then use this session key to send encrypted messages back and forth. Because successfully designing one of these protocols is difficult, consider it only if you are an experienced cryptographer.
- The sender can send an encrypted session key along with the encrypted message. The sender creates a random session key, encrypts it using the receiver's public key, and sends the key BLOB to the receiver along with the message. The receiver then decodes the session key with his or her private key to decode the message.

▶ **To send an encrypted session key**

1. Create a random session key, using the **CryptGenKey** function.
2. Encode the message, using the session key.

   For more information on encoding a message using a session key, see Encrypting and Decrypting Data.
3. Export the session key into a key BLOB with the **CryptExportKey** function.

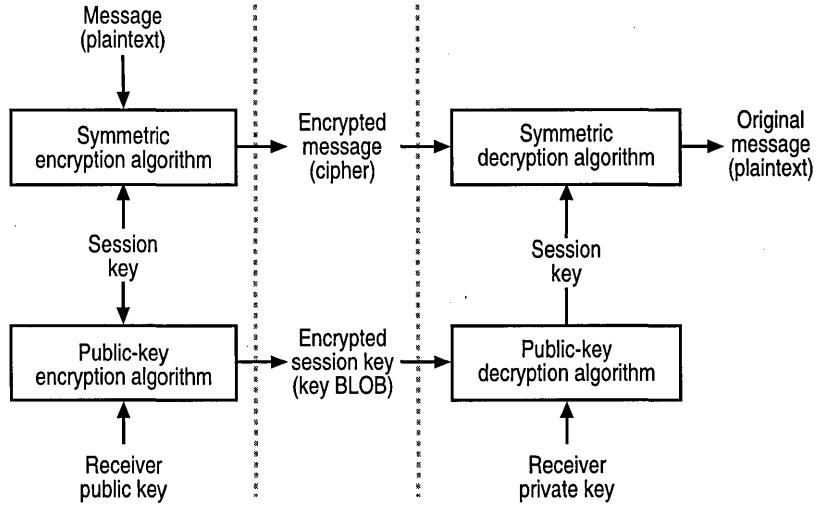
   Specify that the key be encoded with the destination user's key exchange public key, which is the receiver's public key.
4. Send both the encoded message and the encoded key BLOB to the destination user.

5. The receiver then imports the key BLOB into the CSP, using the **CryptImportKey** function.

   This automatically decodes the session key, provided that the destination user's key exchange private key was specified in step three.

6. The receiver can then decode the message, using the session key, following the procedure discussed in Encrypting and Decrypting Data.

The following illustration shows how to send an encoded message, using this procedure.



This approach is vulnerable in at least one way. An unauthorized user can acquire copies of one of more encrypted messages and the encoded keys. Then, at some later time, the eavesdropper can send one of these messages to the receiver and the receiver has no way of knowing that the message did not come directly from the original sender. This risk can be reduced by *timestamping* all messages or by using serial numbers. Timestamping involves attaching the date and time to each message. Using a three-phase key exchange protocol eliminates this problem entirely. For more information on using this protocol, see the Microsoft Platform SDK.

Encrypting and Decrypting Data

An encryption key is needed before invoking encryption and decryption operations. This key is obtained by using the **CryptGenKey CryptDeriveKey**, or **CryptImportKey** functions. The encryption algorithm is specified when the key is created. You can also specify additional encryption parameters, using the **CryptSetKeyParam** function.

For more information on **CryptGenKey** and **CryptDeriveKey**, see Generating Cryptographic Keys. For more information on **CryptImportKey**, see Exchanging Cryptographic Keys.

The following table shows the functions you can use to encode and decode a message.

Function	Description
CryptEncrypt	Encodes a section of plaintext, using the specified encryption key
CryptDecrypt	Decodes a section of cipher, using the specified decryption key

To enable the user to decode the data in the future, the **CryptExportKey** function is used to save the decryption key in a key BLOB that can only be decoded with the user's private key. This function requires the user's key exchange public key for this purpose, which can be obtained by using the **CryptGetUserKey** function. **CryptExportKey** returns a key BLOB that must be stored by the application for use in decoding the file.

To encode a file so that only the current user can access its data, bulk encode the file with a symmetric cipher. The key to this cipher is kept in the key BLOB that can only be decoded with the user's private key. This technique also works for encoding messages for specific recipients.

To encode a message, a session key must first be generated by using the **CryptGenKey** function. Calling this function generates a random key and returns a handle so that the key can encode and decode data. You should specify the encryption algorithm at this point. Because CAPI does not permit applications to use public-key algorithms to encode bulk data, call **CryptGenKey** to specify a symmetric algorithm, such as RC2 or RC4, for your application.

Alternatively, if your application needs to encode the message in such a way that anyone with a specified password can decode the data, the **CryptDeriveKey** function should be used to transform the password into a key suitable for encryption. In this case, **CryptDeriveKey** is called instead of **CryptGenKey** ,and the subsequent **CryptExportKey** calls are not needed.

Once the key is generated, other cryptographic properties of the key can be set with **CryptSetKeyParam**. For example, different sections of the file can be encoded with different salt values, and the cipher mode or initialization vector can be changed. Applications can generate salt values with the **CryptGenRandom** function.

In block ciphers, you can change the method of encryption by setting the block cipher properties with **CryptSetKeyParam**.

The following table shows the ciper modes.

Cipher mode	Description
Electronic codebook (ECB)	Encodes blocks individually. No feedback is used.
Cipher block chaining (CBC)	Encodes blocks, using feedback to ensure uniqueness
Cipher feedback mode (CFB)	Encodes small increments of plaintext at a time, not entire blocks
Output feedback mode (OFB)	Encodes similarly to CFB, but uses a different method for filling shift registers

Electronic codebook (ECB): In this cipher mode, each block is encoded individually and no feedback is used. This means that identical blocks of plaintext encoded with the same key are transformed into identical cipher blocks. If a single bit of the cipher block is garbled, then the entire corresponding plaintext block is also garbled.

Cipher block chaining (CBC): Each plaintext block in this cipher mode is encoded, based on the cipher of the previous block. CBC ensures that even if the plaintext contains many identical blocks, each encodes to a different cipher block. Similarly to EBC, if a single bit of the cipher block is garbled, the corresponding plaintext block is also garbled. Moreover, a bit in the subsequent plaintext block in the same position as the original garbled bit, is garbled. If there are extra or missing bytes in the cipher, the plaintext is garbled from that point on.

Cipher feedback mode (CFB): In this cipher mode, small increments of plaintext can be processed into cipher, instead of processing entire blocks at a time. CFB is useful in some situations. For example, data originating from a keyboard can be encoded at each keystroke without waiting for an entire block to be typed.

This mode uses a shift register that is one block size in length and divided up into sections. For example, if the block size is 64 bits with 8 bits processed at a time, the shift register is divided into eight sections.

CFB follows this process for each encryption cycle:

1. The shift register is filled with the initialization vector.
2. The block in the shift register is encoded.
3. The leftmost 8 bits in the encoded shift register are matched with the next 8 bits of plaintext and sent off as 8 bits of cipher.
4. The shift register shifts 8 bits to the left.
5. The 8 bits of cipher generated in step 2 are placed in the rightmost 8 bits of the shift register.

In CAPI, the number of bits processed at a time is specified by setting the encryption key's KP_MODE_BITS parameter using the **CryptSetKeyParam** function. The default value for this parameter is typically 8 bits.

If 1 bit in the cipher is garbled, 1 bit in the plaintext is garbled, and the shift register is corrupted. This corruption results in the corrupting of subsequent plaintext blocks until the bad bit is shifted out of the shift register.

Output feedback mode (OFB): This cipher mode is identical to CFB, except the shift register is filled differently. If 1 bit in the cipher is garbled, the corresponding bit of plaintext is also garbled. If there are extra or missing bits from the cipher, the plaintext is garbled from that point on.

If the application does not explicitly specify one of these modes, thenCBC is used.

Encode the data in the file with the **CryptEncrypt** function, which takes the previously generated session key and encodes a buffer of data. As the data is encoded, it may be slightly expanded by the encryption algorithm. The application is responsible for remembering the length of the encoded data so the proper length can later be given to the **CryptDecrypt** function.

If your application has certificates or public keys for other users, it can permit other users to decode the file by performing **CryptExportKey** calls for each user to whom it wants to give access. The returned key BLOBs must be stored by the application, as in the previous paragraph.

Once a file or message has been encoded, the following data must be stored by the application:

- Encoded data
- One or more key BLOBs, each containing the session key used to encode the message
- Any salt values specified as the data was encoded
- Any initialization vectors specified as the data was encoded

All parameters that were specified with the **CryptSetKeyParam** function as the message was being encoded must also be specified as the message is decoded. It may be appropriate to store some of these parameters with the encoded message, as well.

The following code example reads data from a text file named Test2.txt, encodes it using the RC2 block cipher, and writes out the encoded data to a file named Test.xxx. A random session key is generated to perform the encryption and is stored to the output file along with the encoded data. This session key is encoded with the user's key exchange public key by the **CryptExportKey** function.

```
#include <windows.h>
#include <stdio.h>
#include <wincrypt.h>

#define BLOCK_SIZE            1000
#define BUFFER_SIZE           1008

BOOL EncryptFile (LPTSTR, LPTSTR, LPTSTR);

/*****************************************************************************

  WinMain

*****************************************************************************/
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPTSTR lpCmdLine, int nCmdShow)
{
  LPTSTR lpszSource  = TEXT("test2.txt");
  LPTSTR lpszDestination = TEXT("test.xxx");
  LPTSTR lpszPassword  = TEXT("password");

  if (!EncryptFile (lpszSource, lpszDestination, lpszPassword))
  {
    wprintf (TEXT("Error encrypting file!\n"));
    return 1;
  }

  return 0;
}

/*****************************************************************************

  EncryptFile

*****************************************************************************/
BOOL EncryptFile (LPTSTR lpszSource, LPTSTR lpszDestination,
                  LPTSTR lpszPassword)
{
  FILE *hSrcFile = NULL,
       *hDestFile = NULL;
```

```
HCRYPTPROV hProv = 0;
HCRYPTHASH hHash = 0;
HCRYPTKEY hKey = 0,
          hXchgKey = 0;

PBYTE pbBuffer = NULL,
      pbKeyBlob = NULL;

BOOL bEOF = 0,
     bReturn = FALSE;

DWORD dwCount,
      dwKeyBlobLen;

// Open the source file.
if ((hSrcFile = _wfopen (lpszSource, TEXT("rb"))) == NULL)
{
  wprintf (TEXT("Error opening Plaintext file!\n"));
  goto exit;
}

// Open the destination file.
if ((hDestFile = _wfopen (lpszDestination, TEXT("wb"))) == NULL)
{
  wprintf (TEXT("Error opening Ciphertext file!\n"));
  goto exit;
}

// Get the handle to the default provider.
if (!CryptAcquireContext (&hProv, NULL, NULL, PROV_RSA_FULL, 0))
{
  wprintf (TEXT("Error %x during CryptAcquireContext!\n"),
           GetLastError ());
  goto exit;
}

if (lpszPassword == NULL)
{
  // Encrypt the file with a random session key.

  // Create a random session key.
  if (!CryptGenKey (hProv, CALG_RC2, CRYPT_EXPORTABLE, &hKey))
  {
    wprintf (TEXT("Error %x during CryptGenKey!\n"),
             GetLastError ());
    goto exit;
  }
```

```
// Get the handle to the key exchange public key.
if (!CryptGetUserKey (hProv, AT_KEYEXCHANGE, &hXchgKey))
{
  wprintf (TEXT("Error %x during CryptGetUserKey!\n"),
           GetLastError ());
  goto exit;
}

// Determine the size of the key BLOB and allocate memory.
if (!CryptExportKey (hKey, hXchgKey, SIMPLEBLOB, 0, NULL,
                     &dwKeyBlobLen))
{
  wprintf (TEXT("Error %x computing blob length!\n"),
           GetLastError ());
  goto exit;
}

if ((pbKeyBlob = malloc (dwKeyBlobLen)) == NULL)
{
  wprintf (TEXT("Out of memory!\n"));
  goto exit;
}

// Export the session key into a simple key BLOB.
if (!CryptExportKey (hKey, hXchgKey, SIMPLEBLOB, 0, pbKeyBlob,
                     &dwKeyBlobLen))
{
  wprintf (TEXT("Error %x during CryptExportKey!\n"),
           GetLastError ());
  goto exit;
}

// Write the size of key BLOB to the destination file.
fwrite (&dwKeyBlobLen, sizeof (DWORD), 1, hDestFile);

if (ferror (hDestFile))
{
  wprintf (TEXT("Error writing header!\n"));
  goto exit;
}

// Write the key BLOB to the destination file.
fwrite (pbKeyBlob, 1, dwKeyBlobLen, hDestFile);
```

```
  if (ferror (hDestFile))
  {
    wprintf (TEXT("Error writing header!\n"));
    goto exit;
  }
}
else
{
  // Encrypt the file with a session key derived from a password.

  // Create a hash object.
  if (!CryptCreateHash (hProv, CALG_MD5, 0, 0, &hHash))
  {
    wprintf (TEXT("Error %x during CryptCreateHash!\n"),
             GetLastError ());
    goto exit;
  }

  // Hash in the password data.
  if (!CryptHashData (hHash, (PBYTE)lpszPassword,
                      wcslen (lpszPassword), 0))
  {
    wprintf (TEXT("Error %x during CryptHashData!\n"),
             GetLastError ());
    goto exit;
  }

  // Derive a session key from the hash object.
  if (!CryptDeriveKey (hProv, CALG_RC2, hHash, 0, &hKey))
  {
    wprintf (TEXT("Error %x during CryptDeriveKey!\n"),
             GetLastError ());
    goto exit;
  }
}

// Allocate memory.
if ((pbBuffer = malloc (BUFFER_SIZE)) == NULL)
{
  wprintf (TEXT("Out of memory!\n"));
  goto exit;
}

// Encrypt the source file and write to the destination file.
do
{
  // Read up to BLOCK_SIZE bytes from the source file.
  dwCount = fread (pbBuffer, 1, BLOCK_SIZE, hSrcFile);
```

```
    if (ferror (hSrcFile))
    {
      wprintf (TEXT("Error reading Plaintext!\n"));
      goto exit;
    }

    bEOF = feof (hSrcFile);

    // Encrypt the data.
    if (!CryptEncrypt (hKey, 0, bEOF, 0, pbBuffer, &dwCount,
                       BUFFER_SIZE))
    {
      wprintf (TEXT("bytes required:%d\n"), dwCount);
      wprintf (TEXT("Error %x during CryptEncrypt!\n"),
               GetLastError ());
      goto exit;
    }

    // Write the data to the destination file.
    fwrite (pbBuffer, 1, dwCount, hDestFile);

    if (ferror (hDestFile))
    {
      wprintf (TEXT("Error writing Ciphertext!\n"));
      goto exit;
    }
  } while (!bEOF);

  bReturn = TRUE;

  wprintf (TEXT("OK\n"));

exit:

   // Close the files.
  if (hSrcFile)
    fclose (hSrcFile);

  if (hDestFile)
    fclose (hDestFile);

  // Free memory.
  if (pbKeyBlob)
    free (pbKeyBlob);

  if (pbBuffer)
    free (pbBuffer);
```

```
    // Destroy the session key.
    if (hKey)
      CryptDestroyKey (hKey);

    // Release the key exchange key handle.
    if (hXchgKey)
      CryptDestroyKey (hXchgKey);

    // Destroy the hash object.
    if (hHash)
      CryptDestroyHash (hHash);

    // Release the provider handle.
    if (hProv)
      CryptReleaseContext (hProv, 0);

    return bReturn;
}
```

If a message was encoded for a particular user, the **CryptGenKey** function was used to create a random session key before the encryption was performed. Before decoding the message, the key BLOB containing the session key must be imported into the CSP with the **CryptImportKey** function. This function uses the user's key exchange private key to decode the key BLOB and ensure that the originating key BLOB was created using the matching key exchange public key.

If the message was encoded so that any password holder can access the data, **CryptImportKey** is not used. Instead, create the decryption session key with the **CryptDeriveKey** function. You also need to supply the function with the password or other access token.

The session key's parameters must be configured in the same way as when the encryption was performed. These parameters can be specified using the **CryptSetKeyParam** function. For example, if the salt value was changed one or more times during the encryption process, it must also be changed during the decryption process in exactly the same manner.

The message is decoded using the **CryptDecrypt** function. If the message is too large to fit comfortably in memory, it can be decoded in sections, through multiple calls to **CryptDecrypt**.

When the decryption is complete, be sure to destroy the session key, using the **CryptDestroyKey** function. In addition to destroying the key, this frees CSP resources.

The following code example decodes the file created by the previous example of encryption. This decryption example uses the RC2 block cipher and writes out the plaintext data to a file named Test2.txt. The session key used to perform the decryption is read from the cipher file.

```
#include <windows.h>
#include <stdio.h>
#include <wincrypt.h>

#define BLOCK_SIZE            1000
#define BUFFER_SIZE           1008

BOOL DecryptFile (LPTSTR, LPTSTR, LPTSTR);

/*****************************************************************************

  WinMain

*****************************************************************************/
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPTSTR lpCmdLine, int nCmdShow)
{
  LPTSTR lpszSource  = TEXT("test.xxx");
  LPTSTR lpszDestination = TEXT("test2.txt");
  LPTSTR lpszPassword  = TEXT("password");

  if (!DecryptFile (lpszSource, lpszDestination, lpszPassword))
  {
    wprintf (TEXT("Error encrypting file!\n"));
    return 1;
  }

  return 0;
}

/*****************************************************************************

  DecryptFile

*****************************************************************************/
BOOL DecryptFile (LPTSTR lpszSource, LPTSTR lpszDestination,
                  LPTSTR lpszPassword)
{
  FILE *hSrcFile = NULL,
       *hDestFile = NULL;
```

```
HCRYPTPROV hProv = 0;
HCRYPTHASH hHash = 0;
HCRYPTKEY hKey = 0;

PBYTE pbBuffer = NULL,
      pbKeyBlob = NULL;

BOOL bEOF = 0,
     bReturn = FALSE;

DWORD dwCount,
      dwKeyBlobLen;

// Open the source file.
if ((hSrcFile = _wfopen (lpszSource, TEXT("rb"))) == NULL)
{
  wprintf (TEXT("Error opening Ciphertext file!\n"));
  goto exit;
}

// Open the destination file.
if ((hDestFile = _wfopen (lpszDestination, TEXT("wb"))) == NULL)
{
  wprintf (TEXT("Error opening Plaintext file!\n"));
  goto exit;
}

// Get the handle to the default provider.
if (!CryptAcquireContext (&hProv, NULL, NULL, PROV_RSA_FULL, 0))
{
  wprintf (TEXT("Error %x during CryptAcquireContext!\n"),
           GetLastError ());
  goto exit;
}

if (lpszPassword == NULL)
{
  // Decrypt the file with the saved session key.

  // Read key BLOB length from the source file and allocate memory.
  fread (&dwKeyBlobLen, sizeof (DWORD), 1, hSrcFile);

  if (ferror (hSrcFile) || feof (hSrcFile))
  {
    wprintf (TEXT("Error reading file header!\n"));
    goto exit;
  }
```

```
    if ((pbKeyBlob = (PBYTE)malloc (dwKeyBlobLen)) == NULL)
    {
      wprintf (TEXT("Out of memory or improperly formatted source ")
               TEXT("file!\n"));
      goto exit;
    }

    // Read the key BLOB from source file.
    fread (pbKeyBlob, 1, dwKeyBlobLen, hSrcFile);

    if (ferror (hSrcFile) || feof (hSrcFile))
    {
      wprintf (TEXT("Error reading file header!\n"));
      goto exit;
    }

    // Import the key BLOB into the CSP.
    if (!CryptImportKey (hProv, pbKeyBlob, dwKeyBlobLen, 0, 0, &hKey))
    {
      wprintf (TEXT("Error %x during CryptImportKey!\n"),
               GetLastError ());
      goto exit;
    }
  }
  else
  {
    // Decrypt the file with a session key derived from a password.

    // Create a hash object.
    if (!CryptCreateHash (hProv, CALG_MD5, 0, 0, &hHash))
    {
      wprintf (TEXT("Error %x during CryptCreateHash!\n"),
               GetLastError ());
      goto exit;
    }

    // Hash in the password data.
    if (!CryptHashData (hHash, (PBYTE)lpszPassword,
                        wcslen (lpszPassword), 0))
    {
      wprintf (TEXT("Error %x during CryptHashData!\n"),
               GetLastError ());
      goto exit;
    }
```

```
    // Derive a session key from the hash object.
    if (!CryptDeriveKey (hProv, CALG_RC2, hHash, 0, &hKey))
    {
      wprintf (TEXT("Error %x during CryptDeriveKey!\n"),
               GetLastError ());
      goto exit;
    }
  }

  // Allocate memory.
  if ((pbBuffer = (PBYTE)malloc (BUFFER_SIZE)) == NULL)
  {
    wprintf (TEXT("Out of memory!\n"));
    goto exit;
  }

  // Decrypt the source file and write to the destination file.
  do
  {
    // Read up to BLOCK_SIZE bytes from the source file.
    dwCount = fread (pbBuffer, 1, BLOCK_SIZE, hSrcFile);

    if (ferror (hSrcFile))
    {
      wprintf (TEXT("Error reading Ciphertext!\n"));
      goto exit;
    }

    bEOF = feof (hSrcFile);

    // Decrypt the data.
    if (!CryptDecrypt (hKey, 0, bEOF, 0, pbBuffer, &dwCount))
    {
      wprintf (TEXT("Error %x during CryptDecrypt!\n"),
               GetLastError ());
      goto exit;
    }

    // Write the data to the destination file.
    fwrite (pbBuffer, 1, dwCount, hDestFile);

    if (ferror (hDestFile))
    {
      wprintf (TEXT("Error writing Plaintext!\n"));
      goto exit;
    }
  } while (!bEOF);
```

```
    bReturn = TRUE;

    wprintf (TEXT("OK\n"));

exit:

    // Close the source files.
    if (hSrcFile)
      fclose (hSrcFile);

    // Close the destination files.
    if (hDestFile)
      fclose (hDestFile);

    // Free memory.
    if (pbKeyBlob)
      free (pbKeyBlob);

    // Free memory.
    if (pbBuffer)
      free (pbBuffer);

    // Destroy the session key.
    if (hKey)
      CryptDestroyKey (hKey);

    // Destroy the hash object.
    if (hHash)
      CryptDestroyHash (hHash);

    // Release the provider handle.
    if (hProv)
      CryptReleaseContext (hProv, 0);

    return bReturn;
}
```

Encrypting and Decrypting Simultaneously

When encrypting or decrypting data simultaneously with the same key, the same physical session key must not be used for both operations. This is because every session key contains internal state data that becomes jumbled if it is used for more than one operation at a time. A simple solution to this problem is to make a copy of the session key so that the original key can be used for one operation and the copy used for the other.

Copying a session key is done by exporting the key with **CryptExportKey** and then using **CryptImportKey** to import it back in. When the key is imported, the CSP gives the imported key its own section of internal memory, as if it were not related to the original key.

The following code example shows how a copy of a session key can be obtained.

```
HCRYPTPROV hProv;           // Handle to a CSP
HCRYPTKEY hKey;             // Handle to a session key
HCRYPTKEY hCopyKey = 0,
          hPubKey = 0;

BYTE pbBlob[256];
DWORD dwBlobLen;

// Get a handle to your own key exchange public key.
CryptGetUserKey (hProv, AT_KEYEXCHANGE, &hPubKey);

// Export the session key into a key BLOB.
dwBlobLen = 256;
CryptExportKey (hKey, hPubKey, SIMPLEBLOB, 0, pbBlob, &dwBlobLen);

// Import the session key back into the CSP. This is stored separately
// from the original session key.
CryptImportKey (hProv, pbBlob, dwBlobLen, 0, 0, &hCopyKey);
```

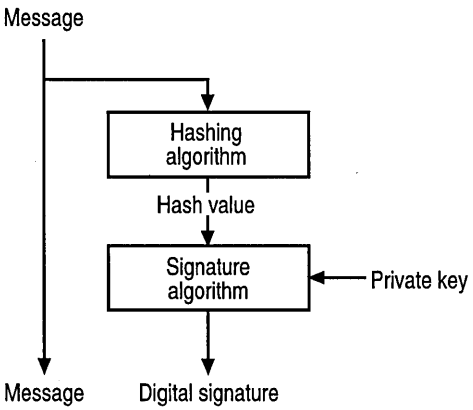
This technique should not be used with stream ciphers because stream cipher keys should never be used more than once. Instead, use separate keys to transmit and receive data.

Creating Digital Signatures

The following table shows the functions applications can use to compute secure digests of data and to create and verify digital signatures.

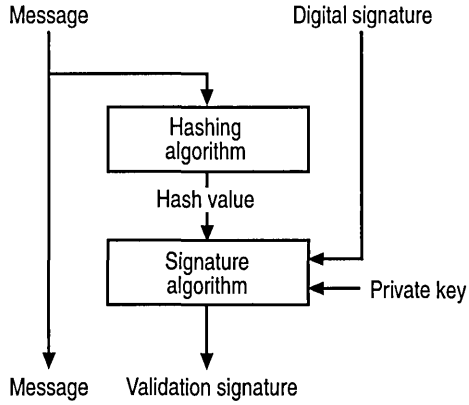
Function	Description
CryptCreateHash	Creates an empty hash object
CryptDestroyHash	Destroys a hash object
CryptGetHashParam	Retrieves a hash object parameter
CryptHashData	Hashes a block of data, adding it to the specified hash object
CryptHashSessionKey	Hashes a session key, adding it to the specified hash object
CryptSetHashParam	Sets a hash object parameter
CryptSignHash	Signs the specified hash object
CryptVerifySignature	Verifies a digital signature, given a handle to the hash object that was signed

To create a digital signature from a message, create a *hash value*, also known as a *message digest*, from the message. Then, use the signer's private key to sign the hash value. The following illustration shows the process for creating a digital signature.



To verify a digital signature, both the message and the signature are required. First, a hash value must be created from the message in the same way as it was done when the signature was created. This hash value is then verified against the signature, using the public key of the signer. If the hash value and the signature match, you can be confident that the message is the one originally signed and that it has not been tampered with.

The following illustration shows the process of verifying a digital signature.



A hash value consists of a small amount of binary data, typically 160 bits. It is produced using a hashing algorithm.

All hash values share the following properties, regardless of the algorithm used:

- A hash value is of a fixed length, regardless of the size of the message.
- Every pair of nonidentical messages translates into a different hash value, even if the two messages differ only by a single bit. Using today's technology, it is not feasible to discover a pair of messages that translate to the same hash value without breaking the hashing algorithm.
- All hashing algorithms are fully deterministic. That is, each time a particular message is hashed using the same algorithm, the same hash value is produced.
- All hashing algorithms are one-way. Given a hash value, it is not possible to recover the original message. In fact, none of the properties of the original message can be determined with the hash value alone.

Signing and Verifying Messages

To sign data, a hash object must first be created using the **CryptCreateHash** function. This object accumulates the data to be signed. Next, the data is added to the hash object with the **CryptHashData** function.

After the last block of data is added to the hash, the **CryptSignHash** function is used to sign the hash. A description of the data can also be added to the hash object at this point. Once the digital signature data has been obtained, the hash object should be destroyed by using the **CryptDestroyHash** function.

Hashes can be signed with either the signature private key or the key exchange private key. The signature key should be used when the user who owns the signature key is signing some of his or her data. The key exchange key should be used when signing data that does not directly belong to the user. The classic example of this is when the exchange key is used to sign session keys during a key exchange protocol.

To verify a signature, applications must first create a hash object, using **CryptCreateHash**. This object accumulates the data to be verified. The data is then added to the hash object with **CryptHashData**.

After the last block of data is added to the hash, the **CryptVerifySignature** function is used to verify the signature. The signature data, a handle to the hash object, and the description string must all be supplied to **CryptVerifySignature**. A handle to the key pair that was used to sign the data must also be specified.

Once the signature has been verified, or has failed the verification, the hash object should be destroyed by using **CryptDestroyHash**.

To obtain the hash value, a hash object must first be created using **CryptCreateHash**. This object accumulates the data to be verified. The data is then added to the hash object with **CryptHashData**.

After the last block of data is added to the hash, the **CryptGetHashParam** function is used to obtain the hash value.

Once the hash value has been obtained, the hash object should be destroyed with **CryptDestroyHash**.

Because CAPI handles the actual method of doing the signature, applications do not need to be aware of how the signature is applied.

Hashing and Digital Signature Algorithms

This section lists several algorithms used to compute hashes and digital signatures. Each of these algorithms is supported by the Microsoft RSA Base Provider.

The MD2, MD4, and MD5 hashing algorithms were all developed by RSA Data Security, Inc. These algorithms were developed in sequential order. The later algorithms are generally more secure than the earlier ones. All three algorithms generate 128-bit hash values.

The secure hashing algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and by the National Security Agency (NSA). This algorithm was developed for use with Digital Signature Algorithm (DSA) or Digital Signature Standard (DSS). This algorithm generates a 160-bit hash value.

Message Authentication Codes are similar to hash values, but are computed using a session key. Because of this, you must possess the session key to recompute the hash value to verify that the base data has not changed.

The Message Authentication Codes implemented by the Microsoft RSA Base Provider are of the most common sort. That is, they are block cipher Message Authentication Codes. This method encodes the base data with a block cipher and then uses the last encoded block as the hash value. The encryption algorithm used to build the Message Authentication Code is the one that was specified when the session key was created.

Warning The same session key should not be used for both message encryption and Message Authentication Codes generation. Doing so greatly increases the risk of your messages being decoded.

Administrating CAPI

This section discusses how multiple CSPs can be installed on a computer and the default providers specified. The structure of the registry is also mentioned.

New providers are installed by running their setup applications. This copies the CSP files to the appropriate directories and makes all needed changes to the registry.

Overview of the CAPI Registry

CAPI uses the registry to store a database of the CSPs that have been installed on the Windows CE–based device. Both the Windows CE–based device default providers and the user-installed providers are recorded here.

Warning This section is included for informational purposes only. Details of the CAPI registry use may change at any time. Under no circumstances should an application read from or alter the registry directly.

Entries under the **HKEY_LOCAL_MACHINE\...\Provider** key contain data about all of the CSPs that have been installed on the computer. These entries are created by the setup application used to install a new CSP. These entries are organized under subkeys whose names indicate the provider name. For example:

```
[HKEY_LOCAL_MACHINE\Comm\Security\Crypto\Defaults\Provider\Microsoft
Base Cryptographic Provider v1.0]
    Image Path = REG_SZ:rsabase.dll
    Signature = REG_BINARY:digital signature
    Type = REG_DWORD:0x1

[HKEY_LOCAL_MACHINE\Comm\Security\Crypto\Defaults\Provider\XYZ Provider]
    Image Path = REG_SZ:johncsp.dll
    Signature = REG_BINARY:digital signature
    Type = REG_DWORD:0x2a
```

Entries under the **HKEY_LOCAL_MACHINE\...\Provider Types** key contain the name of the machine default CSP for each provider type. These entries are also created by the setup application used to install a new CSP. These entries are organized under subkeys whose names, appearing in decimal format, indicate the provider type. For example:

```
[HKEY_LOCAL_MACHINE\Comm\Security\Crypto\Defaults\Provider Types\Type
001]
    Name = REG_SZ:Microsoft Base Cryptographic Provider v1.0

[HKEY_LOCAL_MACHINE\Comm\Security\Crypto\Defaults\Provider Types\Type
042]
    Name = REG_SZ:XYZ Provider
```

Entries under the **HKEY_CURRENT_USER\...\Providers** key contain the name of the current user default CSP for each provider type. These entries are created or modified by the **CryptSetProvider** function. These entries are organized under subkeys whose names indicate the provider type. For example:

```
[HKEY_CURRENT_USER\Comm\Security\Crypto\Providers\Type 001]
    Name = REG_SZ:Microsoft Base Cryptographic Provider v1.0
```

Writing a CSP

Once you have decided which cryptographic algorithms and data formats are to be included in your CSP and obtained implementations for each of them, putting together a CSP is comparatively straightforward.

▶ **To create a CSP**

1. Create a DLL that exports all of the CSPI functions.

   If your CSP has hardware components, this might also involve writing a smart-card device driver and/or the embedded code that runs on the card.
2. Write a setup program for the CSP that creates the appropriate registry entries.
3. Test the CSP. This can only be done using the CSP Developer's Kit and the Microsoft Windows CE Platform Builder. Testing the CSP involves the following substeps:
   1. Sign the CSP with the Sign.exe utility, producing a debug signature file.
   2. Install the CSP, using the setup program mentioned in previously.
   3. Run test code that makes calls to the CSP by way of CAPI.
4. Have the CSP signed by Microsoft. This enables the CSP to be used with the released versions of Windows CE and Windows-based desktop platforms. This procedure is described in Getting CSPs Signed.
5. Test the CSP again. This is the same as step 3, except that the original signature and the release version of Windows CE are used.

Writing a CSP Setup Application

At a minimum, a CSP Setup application must copy the CSP DLL to the \Windows\ directory and create the appropriate registry entries.

Registering the CSP

The following registry entries register the CSP with the OS.

```
[HKEY_LOCAL_MACHINE\Comm\Security\Crypto\Defaults\Provider\CSP name]
    Image Path = REG_SZ: CSP DLL name
    Signature = REG_BINARY: digital signature
    Type = REG_DWORD: CSP Type
```

The ***CSP name*** entry must be the textual name of the CSP. If the CSP has been signed by Microsoft, this name must exactly match the CSP name that was specified in the Export Compliance Certificate (ECC).

The ***Image Path*** entry must be set to the name of the CSP DLL. A fully qualified path, such as "\windows\rsabase.dll" can also be specified here.

The ***Signature*** entry must contain a digital signature for the CSP DLL. This signature can either be created with the Sign.exe utility, the debug signature, or obtained from Microsoft.

Setting the Machine Default CSP

One machine default CSP can be specified for each CSP type. This entry is used when an application calls the **CryptAcquireContext** function with only a CSP type specified, and no user default CSP registry entry exists.

The typical CSP Setup application installs its CSP as the machine default. The following registry entry sets the machine default CSP.

```
[HKEY_LOCAL_MACHINE\Comm\Security\Crypto\Defaults\ Provider Types\Type
CSP type]
    Name = REG_SZ:CSP name
```

The ***CSP type*** portion of the key name must be in decimal format, and exactly three digits in length. For example, if the CSP is of type 25, the key name would be **Type 025**.

The ***Name*** entry must be set to the textual name of the CSP. This must exactly match the ***CSP name*** registry key discussed in the previous section.

Setting the User Default CSP

One user default CSP can be specified for each CSP type. This entry is used when an application calls the **CryptAcquireContext** function with only a CSP type specified.

The user defaults are stored in the registry's **HKEY_CURRENT_USER** key.

The user default CSP is to be set by way of the **CryptSetProvider** function, which internally sets the following registry entry:

```
[HKEY_LOCAL_MACHINE\Comm\Security\Crypto\Defaults\Providers\Type CSP
type]
    Name = REG_SZ:CSP name
```

The ***CSP type*** portion of the key name must be in decimal format and exactly three digits in length.

The ***Name*** entry must be set to the textual name of the CSP. This must exactly match the ***CSP name*** registry key discussed earlier.

Testing the CSP

Your CSP DLL must be signed each time that it is built, and the signature placed appropriately in the registry. It is a good idea to incorporate this procedure into your Make file, so that steps are not forgotten.

The Sign.exe utility is used to sign CSP DLLs. Given a DLL file, it produces a signature file, whose contents can be placed into the registry as discussed in the previous section. Sign.exe takes three arguments, as shown:

```
sign {s|v} <filename> <signature file>
```

The first argument must be "s" if a signature file is to be generated, and "v" if an existing signature file is to be verified against the DLL file. The second argument must be the fully qualified file name of the DLL file, and the third argument the fully qualified file name of the signature file.

If the CSP DLL file is called Myxcsp.dll, the following command can be used to generate a signature file for it (called Myxcsp.sig).

```
sign s myxcsp.dll myxcsp.sig
```

Getting CSPs Signed

Every CSP must be digitally signed by Microsoft to be recognized by the operating system. The primary purpose of the digital signature is the protection of the system and its users; the operating system validates this signature periodically to ensure that the CSP has not been tampered with. A secondary effect of the digital signature is that it separates applicable export controls on the CSP from the host operating system and applications, thus allowing broader distribution of encryption-enabled products than would be possible under other circumstances.

Generally, U.S. export law limits the export outside the United States or Canada of products that host strong encryption or an open cryptographic interface. The digital signature requirement effectively prevents arbitrary use of CAPI and enables export of the host operating system and CAPI-enabled applications. By removing encryption services from host systems and applications, CAPI places the burden of U.S. encryption export restrictions on the CSP vendor, who is subject to those controls regardless.

Questions and comments about the CSP signing mechanism, signing procedures and CAPI licensing policy can be directed to cspsign@microsoft.com.

CSP vendors may wish to consult the U.S. Commerce Department, Bureau of Export Administration, Office of Exporter Services for assistance in the classification and/or export licensing of CSPs for export from the United States.

CHAPTER 10

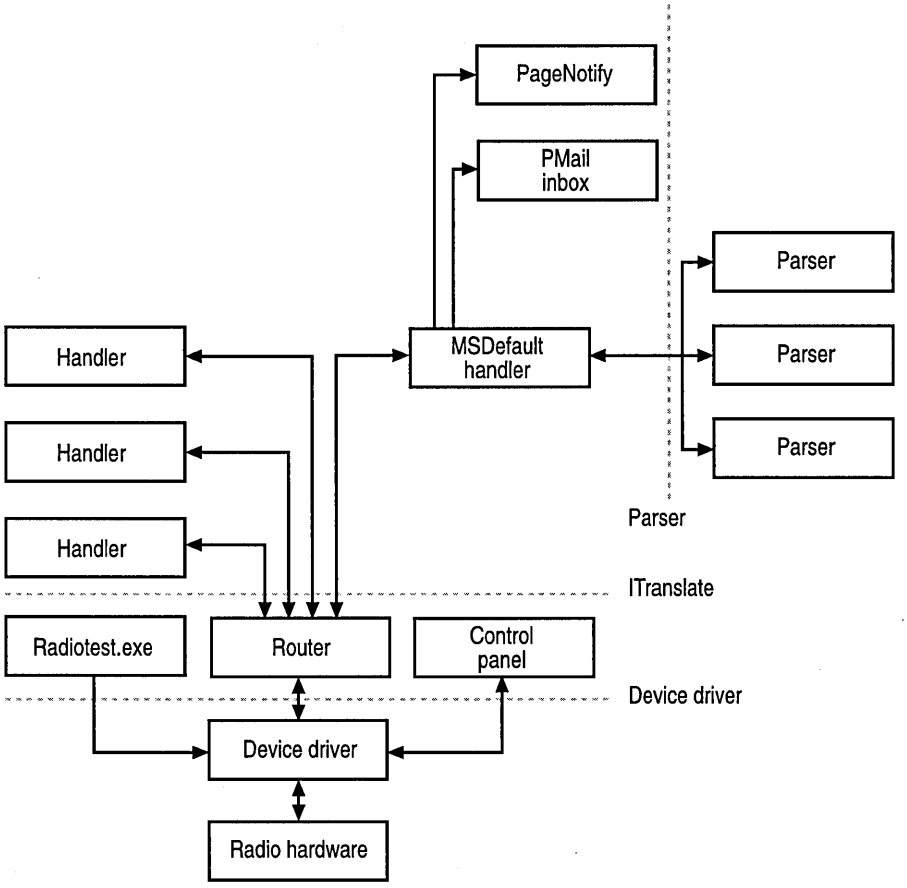
Wireless Services

Wireless Services for Windows CE provides support for receiving e-mail messages, pager messages, and other radio services on a Windows CE–based device. It enables you to create DLLs to customize the application and provides utility programs for testing custom DLLs.

Wireless Services enables multiple service inputs to a Windows CE–based device, using radio hardware, typically a PC Card radio device. The radio hardware may be limited to receiving signals or transmitting a simple acknowledgement, or it may be a fully bidirectional device.

Processing Messages

Messages are passed among different hardware and software areas. The following illustration shows paths a message can take.



Radio hardware is the first component of a Windows CE–based device to interact with a message. After the radio hardware receives a message, the message is sent through several OS levels before it reaches an application in the following way:

1. The radio hardware receives a message.

   The hardware receives all messages on monitored frequencies, whether intended for the device or not.

2. The hardware verifies that the message is for the device by reading the message address.

   If the message is not intended for the device, the message is discarded.

3. The hardware passes the message to the device driver.

   The hardware turns on the Windows CE–based device, if necessary.
4. The device driver sends the message to the router.
5. The router communicates with the device and reads the message address to determine which handler to invoke.
6. The router loads the handler and passes it the message.
7. The handler processes the message.

   The message can be processed by the default handler, or by a custom handler. You can write a handler to perform decryption, store the message data in a database, or update a calendar based on the message content.

By default, messages pass through the **MSDefault** message handler to PMail, a Windows CE–based mail application that synchronizes to Microsoft® Outlook® 97, a desktop information manager.

Message Handlers

Messages are routed to the **MSDefault** message handler. A *message handler* is a COM object that implements the **ITranslate** interface in an in-process COM object. When a message is received, the router looks in the registry to determine which handler to invoke. If no registry entry is found, the default handler, **MSDefault**, is invoked.

MSDefault Message Handler

The primary purpose of **MSDefault** is to convert paging messages into an e-mail format that can be stored in the PMail message store. Pager messages typically consist of alphanumeric text, but a wireless service provider may use a proprietary format to include additional header information. **MSDefault** performs the following actions:

1. **MSDefault** attempts to parse and reformat the message into a format understood by the Windows CE Messaging API (MAPI), and then passes the message to the PMail message store.
2. If appropriate, **MSDefault** executes the PageNotify application, Pagenotif.exe. This application notifies the user when a pager message is received on the pager radio address by chiming or displaying a dialog box.
3. **MSDefault** can be set to limit the number of messages stored in a PMail folder. For example old stock quotations can be discarded as new quotations are received.

Each wireless service provider defines its own message format. **MSDefault** converts these different formats into a standard MAPI format by using parser routines stored in a DLL. These routines must be supplied by the wireless service provider.

The parser routine that **MSDefault** calls to reformat a message is chosen based upon the device and the address on which the message was received. The process of identifying a parser is hierarchical:

1. **MSDefault** looks for a parser for the device and address.
2. It looks for a parser for the device only.
3. If no parser is found, it looks for a user-defined parser or uses its default parser.

All messages received on a specified device and address must be able to be parsed by the parser found by this process. For more information on parsers, see Registering a Parser.

The default parser sets the **From** header field to the string "PAGER", sets the **Type** header field to "PAGE", and sets the **Subject** field to the first 80 characters of the body.

When **MSDefault** receives a message from the router, it searches the registry to find the parser DLL associated with the device and message address. It then loads the DLL, which must contain the **ParseEmailHeaders** function.

Internationalization and Unicode Support

All driver-generated fields in the message header are in Unicode. Strings that the driver receives, however, are passed in the same format as received. If you have set the appropriate registry keys, the message body is converted to Unicode by **MSDefault** using carrier-supplied DLLs. The key is **HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control** and its values are CONVTRDLL and CONVTRPARAM. You may set CONVTRDLL to the name of a DLL that performs the conversion, or to the string "BUILTIN". If you use "BUILTIN", the default Unicode conversion routine in **MSDefault** is used. You may set CONVTRPARAM to A2U to convert from ASCII to Unicode, or to U2U to indicate that no conversion is required.

You can set the subkeys so that conversions are done differently on certain devices or addresses. For example, by setting the key **HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control\Device1**, you can specify which DLL and conversion is used on messages arriving on Device 1.

The following registry key example shows how to convert incoming messages on Addr1 of Device 1 from ASCII to Unicode, using the built-in conversion routine.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control\Device1\Addr1
"ConvtrDll"="BuiltIn";
"ConvtrParam"="A2U";
```

The Radiotest test utility, Radiotest.exe, also supports Unicode messages. For more information, see Sending Test Messages with Radiotest.

Writing a Parser Routine

When you write a parser routine for **MSDefault**, use standard MAPI function calls to set the necessary e-mail message headers. At a minimum, set the **Subject**, **Type**, and **From** fields. You must include the Wis.h header file to obtain the interface definition and link to Wisuuid.lib to obtain the necessary universally unique identifier (UUID). The following code example shows the default parser that **MSDefault** uses when no parser is defined for a device or address.

```
#include <wis.h>
#include <msgstore.h>

LPWSTR ParseEmailHeaders (MailMsg *pMsg, LPCWSTR pText)
{
#define MAX_FROM_TEXT_LEN       80   // Maximum number of characters in
                                     // the From field in PMail
#define MAX_SUBJECT_TEXT_LEN    80   // Maximum number of characters in
                                     // the Subject field in PMail
    TCHAR   szFromText[MAX_FROM_TEXT_LEN + 1];
    TCHAR   szSubjectText[MAX_SUBJECT_TEXT_LEN + 1];
    DWORD   ii, iNum, iIndece;

    ULONG       ulcch;
    LPWSTR      pwszKeyWord;
    ULONG       ulKeyWordsFound;
    MAILHDRS    mhEMail[] = {{L"Fr:", 3, -1},
                             {L"|Re:", 4, -1},
                             {L"|Msg:", 5, -1},
                             {NULL, 0, -1}};
    MAILHDRS    mhWebMail[] = {{L"Msg:", 4, 0},
                               {NULL, 0, 0}};

    // Search for e-mail signatures.
    // The three are Fr:, Re:, and Msg:.
    iNum = 0;
    ulKeyWordsFound = 0;
    while (TRUE)
    {
```

```
        pwszKeyWord = mhEMail[iNum].pwszHeader;
        ulcch = mhEMail[iNum].cchwszHeader;
        DEBUGCHK(pMsg->dwMsgLen >= ulcch);
        for (ii = 0; ii < pMsg->dwMsgLen - ulcch; ii++)
        {
            if (!memcmp (pwszKeyWord, (LPWSTR)&pText[ii], ulcch))
            {
                mhEMail[iNum].ulPosFound = ii;
                ulKeyWordsFound++;
                break;
            }
        }
        iNum++;
        if (!mhEMail[iNum].cchwszHeader)
            break;
    }

    if (ulKeyWordsFound >= 1)
    {
        // Assume it is an e-mail-type message and process.

        if ( (mhEMail[0].ulPosFound != -1) &&
            (mhEMail[2].ulPosFound != -1) )
        {
            if (mhEMail[1].ulPosFound == -1)
            {
                // The delimiters Fr: and Msg: were found.
                iIndece = 2;
            } else {
                // All three delimeters were found.
                iIndece = 1;
            }
            if (mhEMail[iIndece].ulPosFound
                    > (mhEMail[0].ulPosFound + mhEMail[0].cchwszHeader))
            {
                ULONG   ulFrPos = mhEMail[0].ulPosFound
                                + mhEMail[0].cchwszHeader;
                ulcch = mhEMail[iIndece].ulPosFound
                      - mhEMail[0].ulPosFound - mhEMail[0].cchwszHeader;
                if (pText[ulFrPos] == '\"')
                {
                    ulFrPos++;
                    ulcch--;
                }
                if (ulcch)
                {
```

```
                    DEBUGCHK(!(ulcch < 0 || ulcch > MAX_FROM_TEXT_LEN));
                    if (ulcch > MAX_FROM_TEXT_LEN)
                        ulcch = MAX_FROM_TEXT_LEN ;
                    _tcsncpy(szFromText, &pText[ulFrPos], ulcch);
                    szFromText[ulcch] = (TCHAR)'\0';
                    MailSetField(pMsg, L"From", szFromText);
                    goto SetSubject;
                }
            }
            // $REVIEW must be in a localized file.
            MailSetField(pMsg, L"From", L"Unknown Sender Name");

SetSubject:
            if ( (mhEMail[1].ulPosFound != -1) &&
                 (mhEMail[2].ulPosFound
                  > (mhEMail[1].ulPosFound + mhEMail[1].cchwszHeader)) )
            {
                ulcch = mhEMail[2].ulPosFound
                      - mhEMail[1].ulPosFound - mhEMail[1].cchwszHeader;
                DEBUGCHK(!(ulcch < 0 || ulcch > MAX_SUBJECT_TEXT_LEN));
                if (ulcch > MAX_SUBJECT_TEXT_LEN)
                    ulcch = MAX_SUBJECT_TEXT_LEN;
                _tcsncpy(szSubjectText, &pText[mhEMail[1].ulPosFound
                                + mhEMail[1].cchwszHeader], ulcch);
                szSubjectText[ulcch] = (TCHAR)'\0';
                MailSetField(pMsg, L"Subject", szSubjectText);
            }
            else
            {
                // $REVIEW must be in a localized file.
                MailSetField(pMsg, L"Subject", L"N/A");
            }

            pText = pText + mhEMail[2].ulPosFound
                                        + mhEMail[2].cchwszHeader;
        }
        else
        {
            // Only one field was found,
            // or only Fr: and Re:
            // or only Re: and Msg:
            // $REVIEW must be in a localized file.
            MailSetField(pMsg, L"From", L"Scrambled EMail message");
            // $REVIEW must be in a localized file.
            MailSetField(pMsg, L"Subject", L"N/A");
        }
        return (LPWSTR)pText;
    }
```

```
    //  Search for the WebMail signature.
    // "Msg:" should be at the start of the string.
    iNum = 0;
    ulKeyWordsFound = 0;
    ii = 0;
    pwszKeyWord = mhWebMail[iNum].pwszHeader;
    ulcch = mhWebMail[iNum].cchwszHeader;
    DEBUGCHK(pMsg->dwMsgLen >= ulcch);
    if (   (ulcch <= pMsg->dwMsgLen)
        && (!memcmp (pwszKeyWord, (LPWSTR)&pText[0], ulcch)) )
    {
        MailSetField(pMsg, L"Type",L"Page" );
        pText = pText + 4;
        return (LPWSTR)pText;
    }

    // Numeric pages that become alpha pages, due to the
    // configuration of the CUE Receiver, will
    // fall through to here. Change the type.
    MailSetField(pMsg, L"Type",L"Page" );
    return (LPWSTR)pText;

}
```

Registering a Parser

Because each wireless service provider determines the format for e-mail messages sent by it, parsing is done by functions stored in DLLs. **MSDefault** selects the DLL to invoke, based on the device and address on which the message was received. **MSDefault** uses the following criteria to determine which DLL to use:

- If a DLL is specified for a device address, the device DLL is used.
- If no DLL is specified for an address but one is specified for the device, the device DLL is used.
- If no DLL is specified for either the address or the device but one is specified as the default, the default parser is used.
- If no DLL is defined as the default, a system default parser is used.

The registry entries are stored under the key **HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\EmailParser**. For example, consider a Windows CE–based device that has two radio devices, Device 1 and Device 2. Device 1 has three addresses, Tstaddr1, Tstaddr2, and Tstaddr3.

The Windows CE–based device has the following registry entries.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control
"EmailParser"="empdef.dll"
HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control\Device1
"EmailParser"="empdev.dll"
HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control\Device1\Tstaddr1
"EmailParser"="empcue.dll"
HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control\Device1\Tstaddr2
"EmailParser"="emp2.dll"
```

No parser has been defined for Tstaddr3 or Device 2. In this example, the following DLLs would be selected for messages:

- If a message is received on any address of Device 2, Empdef.dll is used to parse the message.
- If a message is received on Tstaddr2 of Device 1, Emp2.dll is used to parse the message.
- If a message is received on Tstaddr3 of Device 1, Empdev.dll is used to parse the message.

In this example, because a default parser has been defined, the system default parser is never called for any message.

Replacing PageNotify

By default, PageNotify, Pagenotif.exe, is defined in the registry as the application that **MSDefault** invokes when a pager message arrives. PageNotify has two modes, one to define its notification behavior when a message arrives on the Personal Page address and one to define its notification behavior for all other addresses. The following key and values register Addr1 as the Personal Page address and register Pagenotif.exe as the notification application.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Paging
; PersonalPageAddress is the Address Tag on which pages arrive.
"PersonalPageAddress"="Addr1"
; NotificationApp is the application MSDefault runs when a page arrives.
"NotificationApp"="PageNotif.exe"
```

You can replace PageNotify with a custom application to notify a user of the arrival of a page. For example, if a pager service features priorities, such as urgent and normal, you might want to create a custom notification application that plays a different sound depending on the message priority. To do this, change the registry key to reference a different notification application. **MSDefault** executes the notification application with the following command line:

```
<notificationapp.exe> Message_ObjectID FolderID
```

Limiting the Number of Messages

The following registry key example shows how to limit the number of messages saved in a given folder by **MSDefault**.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control\Device1\Addr1
"MaxMsgs"=3
HKEY_LOCAL_MACHINE\Software\Microsoft\WIS\Control\Device1\Addr2
"MaxMsgs"=6
```

The folder used for messages that arrive on Addr1 holds three messages. The folder used for messages that arrive on Addr2 holds six messages. As more messages arrive, the oldest messages are deleted.

Writing a Message Handler

In addition to the **IUnknown** interface, which must be implemented by all COM objects, message handlers must implement the **ITranslate** interface in an in-process COM object. The following table shows the **ITranslate** methods.

Method	Description
ProcessMSG(LPMSGINFO)	This method must accept a pointer to an **MSGINFO** structure from the router. It must return one of the following values: ▪ TRANS_S_STOP, if the handler completes successfully. ▪ TRANS_E_STOP, if the handler fails.
GetLastError(pszBuffer, dwSize)	This method takes a pointer to a buffer into which to copy the last error message, and a **DWORD** giving the maximum length of the string that can be copied into the buffer. The length includes one byte for the null terminator. This method returns S_OK.
GetInfo(LPMODULE_INFO)	This method takes a pointer to a **MODULE_INFO** structure, which it fills with handler data. It returns an HRESULT.

The following code example shows the **ITranslate** interface members of a message handler that writes a printable copy of the received message to a file. The file name to which it writes the message is the first string of the message.

```
#include <windows.h>
#include <string.h>
#include <TCHAR.H>

#include <wis.h>

#include "main.h"
#include "trans.h"
#include "resource.h"

STDMETHODIMP CTrans::GetLastError (LPWSTR pszError, DWORD dwSize)
{
    if (pszError && dwSize)  {
        --dwSize;
        if (m_dwLastError >= TRANS_LAST_ERRCODE)  {
            m_dwLastError = TRANS_LAST_ERRCODE;
        }
    }
    wcsncpy(pszError, TRANS_ERROR_STRINGS[m_dwLastError], dwSize);
    pszError[dwSize] = 0;
    return (S_OK);
}

STDMETHODIMP CTrans::GetInfo(LPMODULE_INFO lpInfo)
{
    int i;
    // Read the resources into the buffers.
    if ((i = LoadStringW(g_hInst, IDS_TYPE, m_szType, MAX_INFOSTRING)))
    {
        m_szType[i] = 0;
        if ((i = LoadStringW(g_hInst, IDS_FRIENDLY_NAME,
                             m_szFriendlyName, MAX_INFOSTRING)))
        {
            m_szFriendlyName[i] = 0;
            if ((i = LoadStringW(g_hInst, IDS_DISCRIPTION,
                                 m_szDiscription, MAX_DISSTRING)))
            {
                m_szDescription[i] = 0;
                if ((i = LoadStringW(g_hInst, IDS_VERSION,
                                     m_szVersion, MAX_DISSTRING)))
                {
                    m_szVersion[i] = 0;
                    if ((i = LoadStringW(g_hInst, IDS_MANUFACTURER,
                                     m_szManufacturer, MAX_DISSTRING)))
                    {
```

```
                        m_szManufacturer[i] = 0;
                        *lpInfo = m_ti;
                        return(S_OK);
                    }
                }
            }
        }
    }
    return(E_FAIL);
}


STDMETHODIMP
CTrans::ProcessMSG ( LPMSGINFO pMSG )
{
    WCHAR   szDummy[] = L"\\testmsgs.txt";
    BOOL bOperationComplete = FALSE;
    HRESULT hr =  TRANS_E_CONTINUE;
    HANDLE htmp;

    // Validate the file name.
    if (pMSG->pszFileName == NULL || (lstrlenW(pMSG->pszFileName) == 0))
    {   // The method should have received a valid file name.
        m_dwLastError = TRANS_INVALID_IN_FILENAME;
        return(TRANS_E_STOP);
    }
    // Ensure that the input and output file handles are NULL.
    if (  m_hInput != INVALID_HANDLE_VALUE
       || m_hOutput != INVALID_HANDLE_VALUE)
    {
        m_dwLastError = TRANS_FILES_ALREADY_OPEN;
        return(hr);
    }

    // Open the input file.
    htmp = CreateFileW(pMSG->pszFileName, GENERIC_READ, FILE_SHARE_READ,
                       NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                       NULL );
    if (htmp == INVALID_HANDLE_VALUE)
    {
        // The input file does not exist or cannot be opened.
        m_dwLastError = TRANS_CANT_OPEN_IN_FILE;
        return(hr);
    }
    else
    {
        m_hInput = htmp;
    }
```

```
    // Open the output file.
    htmp = CreateFileW(szDummy, GENERIC_WRITE, FILE_SHARE_READ,
                       NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL,
                       NULL );
    if (htmp == INVALID_HANDLE_VALUE)
    {
        // The output file cannot be opened or created.
        // Close the input file.
        CloseHandle(m_hInput);
        m_hInput = INVALID_HANDLE_VALUE;
        m_dwLastError = TRANS_CANT_OPEN_OUT_FILE;
        return(hr);
    }
    else
    {
        m_hOutput = htmp;
    }


    // Both the input and output files are ready.
    if (pMSG->pRL->Dir == DIR_RECEIVE)
    {
        // Perform Untranslation on the message.
        // TODO -- Replace this If statement with your call.
        if (Untranslate(pMSG))
        {
            bOperationComplete = TRUE;
        }
    }
    else // The other valid value is DIR_TRANSMIT.
    {
        // Perform translation on the message.
        // TODO -- Replace this If statement with your call.
        if (Translate(pMSG))
        {
            bOperationComplete = TRUE;
        }
    }

    // Close the files.
    CloseHandle(m_hInput);
    CloseHandle(m_hOutput);
    m_hInput = INVALID_HANDLE_VALUE;
    m_hOutput = INVALID_HANDLE_VALUE;
    if (bOperationComplete)  {
        hr = TRANS_S_STOP;
    }
    return (hr);
}
```

```
// Dump the MSGINFO structure and the data into a text file.
BOOL DumpToFile(HANDLE hOutFile, LPTSTR Str, ...)
{
    DWORD dwBytesToWrite, dwBytesWritten;
    va_list val;
    WCHAR byBuf[BUFF_SIZE];

    // Create the output string and send it to the
    // debugger's output window, if requested.
    va_start(val,Str);
    wvsprintf(byBuf, Str, val);
    va_end(val);
    dwBytesToWrite = lstrlen(byBuf) * sizeof(WCHAR);
    if (!WriteFile(hOutFile, byBuf, dwBytesToWrite,
                                &dwBytesWritten, NULL)
        || (dwBytesToWrite != dwBytesWritten))
    {
        return FALSE;
    }
    return TRUE;
}

BOOL
CTrans::Translate( LPMSGINFO pMSG )
{
    BYTE byBuf[BUFF_SIZE];
    DWORD dwBytesWritten;
    DWORD dwBytesRead;
    SYSTEMTIME stLocalTime;
    BOOL fRetValue = TRUE;

#if defined(DEBUG) && defined(DEBUG_BREAK)
        DebugBreak();
#endif

    // Append the message to the end of the output file.
    // First, move the file pointer to the end of the output file.
    if (SetFilePointer(m_hOutput, 0, NULL, FILE_END) == 0xFFFFFFFF)  {
        m_dwLastError = TRANS_FILE_SEEK_ERROR;
        return FALSE;
    }
```

```
// Write the message separator to the output file.
GetLocalTime(&stLocalTime);
fRetValue &= DumpToFile(m_hOutput,
                    L"==0--==--== Msg Log %d/%d/%d %d:%d.%d\r\n",
                    stLocalTime.wYear,
                    stLocalTime.wMonth,
                    stLocalTime.wDay,
                    stLocalTime.wHour,
                    stLocalTime.wMinute,
                    stLocalTime.wSecond);

// Write out MSGINFO structure members.
fRetValue &= DumpToFile(m_hOutput, L"Size            =%d\r\n",
                            pMSG->cbSize);
fRetValue &= DumpToFile(m_hOutput, L"FileName        =%s\r\n",
                            pMSG->pszFileName);
fRetValue &= DumpToFile(m_hOutput, L"ErrFileName     =%s\r\n",
                            pMSG->pszErrFileName);
fRetValue &= DumpToFile(m_hOutput, L"ResponseFileName=%s\r\n",
                            pMSG->pszResponseFileName);
fRetValue &= DumpToFile(m_hOutput, L"OEMFileName     =%s\r\n",
                            pMSG->pszOEMFileName);
fRetValue &= DumpToFile(m_hOutput, L"FolderName      =%s\r\n",
                            pMSG->pszFolderName);
fRetValue &= DumpToFile(m_hOutput, L"Source          =%d\r\n",
                            pMSG->Source);
fRetValue &= DumpToFile(m_hOutput, L"Device          =%d\r\n",
                            pMSG->Device);
fRetValue &= DumpToFile(m_hOutput,
                        L"Systemtime      =%d/%d/%d %d:%d.%d\r\n",
                             pMSG->DateTime.wYear,
                             pMSG->DateTime.wMonth,
                             pMSG->DateTime.wDay,
                             pMSG->DateTime.wHour,
                             pMSG->DateTime.wMinute,
                             pMSG->DateTime.wSecond);
fRetValue &= DumpToFile(m_hOutput, L"MsgType         =%d\r\n",
                            pMSG->pXtraMsgInfo->MsgType);
fRetValue &= DumpToFile(m_hOutput, L"MsgPriority     =%d\r\n",
                            pMSG->pXtraMsgInfo->MsgPriority);
fRetValue &= DumpToFile(m_hOutput, L"MsgFlags        =0x%X\r\n",
                            pMSG->pXtraMsgInfo->MsgFlags);
fRetValue &= DumpToFile(m_hOutput, L"NumParts        =%d\r\n",
                            pMSG->pXtraMsgInfo->NumParts);
fRetValue &= DumpToFile(m_hOutput, L"ErrorFlags      =0x%X\r\n",
                            pMSG->pXtraMsgInfo->wErrorFlags);
fRetValue &= DumpToFile(m_hOutput, L"MsgSeqNum       =%d\r\n",
                           pMSG->pXtraMsgInfo->wMsgSequenceNumber);
fRetValue &= DumpToFile(m_hOutput, L"==1-------==\r\n");
```

```
    while ( ReadFile(m_hInput, byBuf, BUFF_SIZE, &dwBytesRead, NULL)
            && dwBytesRead )
    {
        if (!WriteFile(m_hOutput, byBuf, dwBytesRead,
                        &dwBytesWritten, NULL)
             || (dwBytesRead != dwBytesWritten))
        {
            fRetValue = FALSE;
            break;
        }
    }
    fRetValue &= DumpToFile(m_hOutput, L"\r\n==*-------==\r\n");
    return fRetValue;
}

// Make the untranslate function the same as the translate function.
BOOL CTrans::Untranslate( LPMSGINFO pMSG )
{
    return Translate(pMSG);
}
```

Installing a Message Handler

Message handlers must be registered before they can be used. The following registry key example shows the four registry keys that you must set in order to install and register a message handler.

```
[HKEY_CLASSES_ROOT\CLSID\{14EF11D0-3EC7-11d2-90BF-0000F80272E4}]
Default="stock translator"
[HKEY_CLASSES_ROOT\CLSID\{14EF11D0-3EC7-11d2-90BF-
0000F80272E4}\InprocServer32]
Default="\\windows\\stock.dll"
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WIS\Routers\{14EF11D0-3EC7-11d2-
90BF-0000F80272E4}]
"Type"="Translator"
Default=hex:60,2C,2A,2A,20,53,74,6F,63,6B,73,20,2A,2A
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WIS\Control\Device1\Stocks]
"PredefinedRouter"=hex:21,60,2C,2A,2A,3,74,6F,63,6B,73,20,2A,2A
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WIS\Control\Device1\News]
"MaxMsgs"=dword:10
```

The first two keys are COM object registry entries.

The third key registers the handler as a translator and sets the tag to be used by the router to refer to the handler. The first hexadecimal value is the constant 0x60, the second hexadecimal value is 0x20 plus the number of bytes for the tag name, and the rest of the values make up a unique ASCII name. Because the tag in this example is 12 bytes long, the second value is 0x2c.

The fourth key defines how the router routes messages through the handler. Because the first hexadecimal value is 0x20 plus the number of handlers, which is always 1, this value is 0x21. The rest of the values are the tag from the third key.

This example shows a fifth, optional, registry entry that places a limit on the number of messages held in a PMail folder for messages that arrive on this address.

This information is provided in case you need to search the registry for a problem. Wireless Services provides two functions, **RegisterHandler** and **UnregisterHandler**, that simplify the process of registering and deregistering message handlers. **RegisterHandler** enables you to register a message handler in any of the following ways:

- As the handler for messages received by a specific radio device and address
- As the default handler for messages received by a radio device, but that do not have an address-level handler
- As the overall default handler for any messages that do not have message handlers specified at either the radio device or the device-and-address levels

To use these functions in your program, include the Riohlp.h header file.

▶ To register a message handler

1. Specify the device for which the handler is to be registered.
2. Specify the address for which the handler is to be registered.

   If this value is NULL, the handler is assigned as the default handler for the device.
3. Specify a string containing the name of the DLL in which the handler is stored.
4. Call **RegisterHandler**, and check the return value for success.

If you want a message handler to stop handling messages received for an address or device, use **UnregisterHandler**. You can deregister a message handler for a specific device and address or for all device addresses.

▶ **To deregister a message handler**

1. Specify the device for which the handler is to be deregistered.
2. Specify the address for which the handler is to be deregistered.

   If this value is NULL, the handler is deregistered for all addresses on this device.
3. Specify the handler class identifier (CLSID).
4. Call **UnregisterHandler**, and check the return value for success.

Programming Considerations for Message Handlers

While the handler is executing, the router pauses while waiting, and messages can transmit to the radio device. Because a radio device has little memory, it relies on being able to transfer incoming data to a Windows CE–based device before its buffer is full. The code must be written in such a way that the handler returns as quickly as possible. Because the router does not invoke message handlers in separate threads, handlers must not block.

A message handler should not attempt to interact with a user. Instead, it can store a message in a queue, handle the message in real time, or execute an application program in a separate thread.

A message handler should perform all data conversions, including those required for internationalization, such as converting from ASCII to Unicode characters. This ensures that any application to which the message handler passes data does not have to convert between different formats.

Stock Quotation Sample Application

The source code for the StockDB sample application is included with the SDK. The following table shows the code samples used in StockDB. Each code sample is contained in a directory of the same name.

Application section	Description
Trans	A message handler that parses the messages for stock quotations and stores the quotations in a database
Sdump	A viewer application that enables a user to view the accumulated quotations in chart form or as individual quotations by way of a graphical user interface (GUI)
Stockdb	A library of helper functions used by the message handler and the viewer

The router invokes the message handler when messages come in on a specified address and device.

StockDB demonstrates the separation of user interface and message handler, the use of the database to transfer data from the message to the application, and the general handler design.

Testing a Wireless Application

The following table shows the test utility applications included in Wireless Services.

Application	Description
Genpgm.exe	Creates radio address data files from text files you write
Dopgm.exe	Installs the radio address data files created by Genpgm.exe
Radiotest.exe	Sends test messages through the device driver from text files in which you have written the messages
Hwinfotest.exe	Changes values that the device driver reports so that you can test applications that monitor signal strength

Creating Radio Address Data

Genpgm.exe creates the radio address data file that tells the radio card which addresses to use. Genpgm.exe is used with Dopgm.exe, which loads the radio address data file into the radio card.

▶ **To program your radio hardware**

1. Create a file on your system that contains the programming information that you want to use.
3. Run Genpgm.exe, and pass it the name of your file through the /PGM= switch.

   Specify the output file name with the /OUT= switch.
4. Copy the output file to the Windows CE directory on your Windows CE–based device using the Windows CE Services utilities.
5. Run Dopgm.exe on your Windows CE–based device, and pass it the name of your .bin file through the /PGM= switch.

When you restart your machine, repeat steps 4 and 5.

When you run Genpgm.exe, it creates the programming data file in an encrypted format using the entry identification (EID) of the radio hardware as the key. If you do not want the file to be encrypted, use the switch "/KTYPE=N" to indicate encryption should not be performed.

The programming data file follows a specific format. It consists of a series of programming and deprogramming clauses for keys, addresses, and carriers. Any type of block can be repeated to program multiple keys, addresses, or carriers. Comments can be interspersed; a comment begins with a semicolon and extends to the end of the line. Each clause contains a set of parameters and values. Values consist of either strings or byte values. The following table shows the formats of each of these.

Type	Format	Examples
Byte value	Decimal integer from 0 through 255, which must not have a leading 0	0 1 60 219 255
	Octal integer from 0 through 0377, which must begin with 0	0 01 074 0333 0377
	Hexadecimal integer from 0x0 through 0xff, which must begin with 0x	0x0 0x1 0x3c 0xff
	Single-byte character, which must be enclosed in single quotation marks	'c' 'A' ';' '#' ' '
String	Text within quotation marks	"hello, world", "etaoin shrdlu"
	Comma-separated list of byte values	0x68, 101, 0154, 0x6c, 'o'

The following example of a definition file for radio address data shows its syntax. Any type of block can be repeated to program multiple keys, addresses, or data structures about wireless service providers.

```
PROGRAM_KEY
KeyNumber=<ByteValue>
AlgCode=<ByteValue>
KeyTag=<String>
KeyValue=<String>
END
```

```
PROGRAM_ADDRESS
AddressNumber=<ByteValue>
Status=<ByteValue>          ;ENABLE=0x01, PRIORITY=0x02, AC_ONLY=0x04
ExpirationDate=CC,YY,MM,DD
AddressTag=<String>
KeyTag=<String>
AddressName=<String>
AddrDescription=<String>
AddressInfo=<String>
END

PROGRAM_CARRIER
RxFrequency=ByteValue       ; Reception frequency
UserId=<String>             ; User PIN or equivalent
CarrierName=<String>
CarrierDescription=<String>
END

; Use either KeyNumber or KeyTag.
UNPROGRAM_KEY
KeyNumber=<ByteValue>
KeyTag=<String>
END

; Use either AddressNumber or AddressTag.
; If there are any keys associated with the address,
; and if they are not referenced by another address, they are deleted.
UNPROGRAM_ADDRESS
AddressNumber=<ByteValue>
AddressTag=<String>
END

; Use the CarrierName parameter to select which carrier to delete.
UNPROGRAM_CARRIER
CarrierName=<String>
END
```

The following example of a radio address definition file shows an input file for Genpgm.exe that, when used as input for Genpgm.exe, sets up a key and two addresses.

```
PROGRAM_KEY
KeyNumber=0x02
AlgCode=0x22
KeyTag="Key234"
KeyValue=0x20,0x21,0x22,0x23
END
```

```
PROGRAM_ADDRESS
AddressNumber=0x0
Status=0x01              ; ENABLE=0x01, PRIORITY=0x02, AC_ONLY=0x04
; The address will expire on August 12, 1999. August is a single digit,
; not a double digit, 8 not 08, because the zero would be interpreted
; as making it octal.
ExpirationDate=19,99,8,12
AddressTag="Addr1"
KeyTag="Key1"
AddressName="Test Addr1"
AddrDescription="Address used for personal messaging"
AddressInfo="AddrInfo"
END
```

If you save this example of a radio address definition file in a file named Card.dat, then run Genpgm.exe, the file Card.bin is created. It contains the address programming. The following example of a command line shows how to do this.

```
genpgm.exe /I=card.dat
```

Sending Test Messages with Radiotest

Radiotest.exe enables you to send a message through a device driver to emulate a Windows CE–based device that receives a message from a radio card. You can type the message on the command line or send it from a file. If you do not type any command-line switches or messages, Radiotest.exe brings up a GUI with which you can interact. This is useful on machines without a full-size keyboard.

The following table shows the switches used when you run Radiotest.exe.

Switch	Description
/anum	Specifies the address number the message is sent to.
/atag	Specifies the address tag the message is sent to.
/mtype	Specifies the message type (1=numeric, 2=alpha).
/mpri	Specifies the **MsgPriority** field.
/errflags	Specifies the value for the **wErrFlags** field.
/mseq	Specifies the value for the **wMsgSequenceNumber** field.
/mflags	Specifies the value for the **MsgFlags** field.
/delay	Specifies the number of seconds to delay before sending the message.
/count	Specifies the number of times to repeat sending the message.
/log	Specifies the file name of the activity log file.
/msg	Specifies the file name from which to read the message.
/cnvt	Specifies to convert the message from Unicode to ASCII if /cnvt=1. Specifies not to convert if omitted, or if /cnvt=0.

You can specify either **/atag** or **/anum**, but not both. If you specify neither, an address of zero is assumed. If you specify **/anum** and you do not program the address into the device, the tag Addr*XX* is used instead. For example, if you send a message to a nonexistent /anum=4, the message is sent to an address with a tag Addr04.

If **/delay** and **/count** are both specified, the delay value is used before sending each copy of the message, including the first copy.

If no log file is given and an error occurs, a message box pops up to alert the user of the error.

If a file name is specified for a message body, the message is read and sent exactly as it appears in the file.

The following command line examples further explain the switches:

```
radiotest /anum=3 /log=rtst.log Hello, world!
```

This sends a message containing the text "Hello, world!" to address number 3, logging any errors in the file Rtst.log.

```
radiotest /atag=Addr1 /delay=2 /count=5 /msg=testmsg1.txt
```

This sends the contents of the file Testmsg1.txt to the address with the tag Addr1 five times, with a two-second delay before each message is sent.

```
radiotest Test Message
```

This sends a message containing the text "Test Message" to address number 0, which is the default. If an error occurs, a dialog box alerts the user.

Testing Hardware Feedback with Hwinfotest

Hwinfotest.exe enables you to set the values that a device driver reports for a radio receiver signal level. The application is included as sample source code so that you can add other switches to it to suit your testing needs.

To execute Hwinfotest.exe, pass it the **/rxsignallevel** switch.

The following command line example sets the receiver signal level that the device driver reports in the **RADIO_HARDWARE_INFO** structure to the value 5.

```
hwinfotest /rxsignallevel=5
```

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W

Microsoft® Windows® CE Communications Guide

Your official guide to communications and connectivity in Windows CE—straight from the source.

Enable your devices to talk to desktop PCs, the Internet, or other systems by tapping the built-in communication interfaces in Windows CE. From basic serial and infrared communications to RAS and the Windows networking (WinINet) API, you get definitive information to understand your full set of communication and connectivity options—including how to easily add drivers and protocols to extend application functionality for specific platforms. You'll also delve into Windows CE security features, ranging from password authentication to sophisticated data encryption.

Get the definitive guide to programming the Windows CE API.

Programming Microsoft Windows CE
ISBN: 1-57231-856-2

Part No. 097-0002196

mspress.microsoft.com

Microsoft® Press