Powered by Microsoft® Windows® CE

MICROSOFT PROFESSIONAL EDITIONS

Microsoft® Press

The ultimate reference and toolkit for Windows CE

Microsoft® Windows® CE Programmer's Guide

Microsoft®

Windows® CE Programmer's 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. 097-0002194

Contents

Part 2 Connection Services

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) functionality 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 Reference	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 Programmer's Guide*, contains the following:

- **Introduction to Windows CE**

  This chapter describes the four primary modules of the Windows CE operating system: the kernel, the file system, the Graphics, Windowing, and Events Subsystem (GWES), and the communications interface. It also discusses what you should consider as you develop an application for Windows CE.

- **Core Services**

  The chapters in this part describe how Windows CE manages threads, memory, and resources. They describe the Windows CE communications interface and information processing, as well as how to integrate engines into a Windows CE–based application.

- **Connection Services**

  The chapters in this part describe how Windows CE establishes a serial connection with a Windows-based desktop computer to transfer files, debug remotely, and synchronize databases. Additionally, they discuss the connection between a Windows CE–based device and a desktop computer, and how to install applications.

- **Writing Applications for a Global Market**

  The chapters in this part describe how to create internationalized applications. They discuss general programming and design considerations, Unicode, national language support (NLS), and creating an Input Method Editor (IME).

- **Glossary**

  The glossary defines terms that are used in the Windows CE documentation.

About the Code Examples Included in this Guide

The *Microsoft® Windows® CE Programmer's Guide* includes code examples developed with Microsoft® Visual C++® version 6.0 and the Microsoft® Windows® CE Toolkit for Visual C++® version 6.0. The code included in each example is ported for an H/PC and for an H/PC running Microsoft® Windows® CE, Handheld PC Professional Edition software; however, the programming concepts presented in each example applies to all Windows CE–based platforms.

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

Introduction to Windows CE

Microsoft® Windows® CE is a compact, highly efficient, scalable operating system (OS) that is designed for a variety of embedded systems and products. Its multithreaded, multitasking, fully preemptive OS environment is designed specifically for hardware with limited resources. Its modular design enables embedded systems developers and application developers to customize it for a variety of products, such as consumer electronic devices, specialized industrial controllers, and embedded communications devices.

To ensure maximum flexibility, Windows CE is portable to many 32-bit processors, including the following chip sets:

AMD X5	MIPS R4101
ARM 720T	Motorola MPC823
ARM SA-1100	NEC VR4111 (16-bit support)
Hitachi SH4 (16-bit support)	NEC VR4300
IBM PPC 403GC	PPC 821
MIPS 4102	QED5230
MIPS R3910	SH3
MIPS R3912	x86

For the most up-to-date information on supported chip sets, see the Windows CE Web site at http://www.microsoft.com/windowsce/embedded/partner/semiconductor.asp.

Windows CE supports various hardware peripherals, devices, and networking systems. These include keyboards, mouse devices, touch panels, serial ports, Ethernet connections, modems, universal serial bus(USB) devices, audio devices, parallel ports, printer devices, and storage devices, such as PC Cards.

Additionally, Windows CE supports more than 1,000 common Microsoft® Win32® APIs and several additional programming interfaces that you can use to develop applications. These interfaces include:

- Component Object Model (COM)
- Microsoft Foundation Classes (MFC)
- Microsoft® ActiveX® controls
- Microsoft Active Template Library (ATL)

Furthermore, Windows CE supports the following technologies:

- Real-time processing for managing time-critical responses
- A variety of serial and network communication technologies, including USB support
- Mobile Channels, which provides Web services for Windows CE users
- Automation and other methods of interprocess communication

For hardware that is intended to perform as an adjunct to a desktop computer, Windows CE provides the following tools to allow a user to manage and transfer data between a desktop computer and an attached Windows CE–based device:

- A connection manager for establishing and maintaining a connection
- A data synchronization interface to allow synchronization of shared data
- File filters for importing and exporting files
- A remote application programming interface (RAPI) for enabling a client on a desktop computer to request services, such as file manipulation, from a server on an attached Windows CE–based device
- Application installation and management services for installing and uninstalling Windows CE–based applications from an attached desktop computer or other sources

In short, Windows CE is streamlined and flexible enough to use in a variety of small embedded systems, yet powerful enough to use in the newest generation of high-performing industrial and consumer devices.

Operating System Architecture

Windows CE is built from a number of discrete modules, each providing specific functionality. Several of these modules are divided into components. Components enable Windows CE to become very compact (less than 200 KB of ROM), using only the minimum ROM, RAM, and other hardware resources that are required to run a device.

Windows CE contains four modules that provide the most critical features of the operating system: the kernel; the object store; the Graphics, Windowing, and Events Subsystem (GWES); and communications. Windows CE also contains additional, optional modules that support such tasks as managing installable device drivers and supporting COM.

Kernel

The kernel is the core of the OS, and is represented by the **Coredll** module. It provides the base operating system functionality that must be present on all devices. The kernel is responsible for memory management, process management, and certain required file management functions. It manages virtual memory, scheduling, multitasking, multithreading, and exception handling.

Most components of the **Coredll** module are required for any configuration of Windows CE. There are some optional kernel components, however, that are needed only when you include such operating system features as telephony, multimedia, and graphics device interface (GDI) graphics. For more information on the kernel, see the *Microsoft® Windows® CE Programmer's Guide.*

Object Store

The **Filesys** module supports the Windows CE object store API functions. The following table shows the types of persistent storage that the object store supports.

Type of storage	Description
File system	Contains application and data files
System registry	Stores the system configuration and any other information that an application must access quickly
Windows CE database	Provides structured storage

The object store offers an alternative to storing user data and application data in files or in the registry. These various object store components can be selected or omitted during the operating system build process to include only those features that are required. For more information on the object store, see the *Microsoft® Windows® CE Programmer's Guide.*

GWES

GWES is the graphical user interface between a user, your application, and the OS. GWES handles user input by translating keystrokes, stylus movements, and control selections into messages that convey information to applications and the OS. GWES handles output to the user by creating and managing the windows, graphics, and text that are displayed on display devices and printers.

Central to GWES is the window. All applications need windows in order to receive messages from the OS, even those applications created for devices that lack graphical displays. GWES provides controls, menus, dialog boxes, and resources for devices that require a graphical display. It also provides the GDI, which controls the display of text and graphics. For more information on controls, GDI, windows, and messaging, see the *Microsoft® Windows® CE User Interface Services Guide.*

Communications

The communications component provides support for the following communications hardware and data protocols:

- Serial I/O support
- Remote Access Service (RAS)
- Transmission Control Protocol/Internet Protocol (TCP/IP)
- Local Area Network (LAN)
- Telephony API (TAPI)
- Wireless Services for Windows CE

For more information on the communications component, see the *Microsoft® Windows® CE Communication Services Guide.*

Optional Components

In addition to the primary modules just described, other operating system modules are available. These include modules and components in the following categories:

- Device manager and installable device drivers
- Multimedia (sound) support module
- COM support module
- Windows CE Shell module

Each module or component provided in Windows CE supports a group of related API functions that are available to you.

Windows CE–Based Products

Microsoft provides an entire line of Windows CE–based products ranging from tools used to develop Windows CE–based applications and device drivers to tools used to create customized versions of the OS. Additonally, Microsoft has partnered with several third-party vendors to create devices that are powered by Windows CE. These devices—the Handheld PC (H/PC), the Palm-size PC, and the Auto PC—are mobile devices that communicate with desktop computers, networks, the Internet, and each other.

For each of these devices, Microsoft provides a software development kit (SDK) to assist you in creating applications that run on the device. Each SDK contains programming libraries, header files, and source code for sample programs, and documentation that describes how to use the libraries. Each SDK also includes programming guidelines and an API reference, as well as a device driver kit. You can download these SDKs from the Windows CE Web site at http://www.microsoft.com/windowsce.

The following table shows the available SDKs.

SDK	Description
Microsoft® Windows® CE Platform SDK, Handheld PC Edition, version 2.0	A set of libraries that enable you to develop applications for an H/PC. Online documentation provides information on managing memory, communicating with other devices, designing a user interface, and programming with Microsoft® Pocket Excel and Microsoft® Pocket Word. This SDK includes a comprehensive reference and an H/PC emulator.
Microsoft® Windows® CE Platform SDK, Palm-size PC Edition	A set of libraries that enable you to develop applications for a Palm-size PC. Online documentation provides information on managing and accessing files, connecting to other devices, creating navigation and input controls, designing a user interface, and implementing sample code. This SDK includes a comprehensive reference and a Palm-size PC emulator.
Preview Release of the Microsoft® Windows® CE Platform SDK, Auto PC Edition	A set of libraries that enable you to develop applications for an Auto PC. Online documentation provides information on using system services, working with controls, implementing speech, interacting with a vehicle computer, and using position and navigation information.

Operating System Development

For the original equipment manufacturer (OEM) interested in creating a custom version of Windows CE, Microsoft offers the Microsoft® Windows CE Platform Builder. Platform Builder is a development tool that combines the Windows CE operating system with an integrated development environment (IDE) and a rich set of embedded development tools, including cross-compilers, assemblers, kernel debuggers, and remote debugging tools.

Platform Builder enables you to select Windows CE components to include in your custom OS. Although your OS can contain any combination of components, Microsoft provides eight standard configurations from which to choose. The following table shows these configurations.

Configuration	Description
Minkern	Builds a minimal version of Windows CE featuring only the core OS.
Mininput	Builds a minimal version of Windows CE featuring user input and native-driver support.
Mincomm	Builds a minimal version of Windows CE featuring serial communications and networking.
Mingdi	Builds a minimal version of Windows CE featuring graphics device interface (GDI) support.
Minwmgr	Builds a minimal version of Windows CE featuring window management.
Minshell	Builds a nearly complete version of Windows CE featuring the Task Manager and the Command Processor.
Maxall	Builds a fully configured version of Windows CE featuring communication applications.
Command processor	Builds a shell similar to Command.com in Windows 95 or Cmd.exe in Microsoft Windows NT®. This processor relies on a console driver, Condev.dll, to display text in a window.

Platform Builder includes programming guidelines, an API reference, and a device driver kit to help you create custom drivers and applications that run on your particular version of Windows CE. Platform Builder is available through standard retail channels. No additional tools or products are required to build a Windows CE–based OS.

Application Development

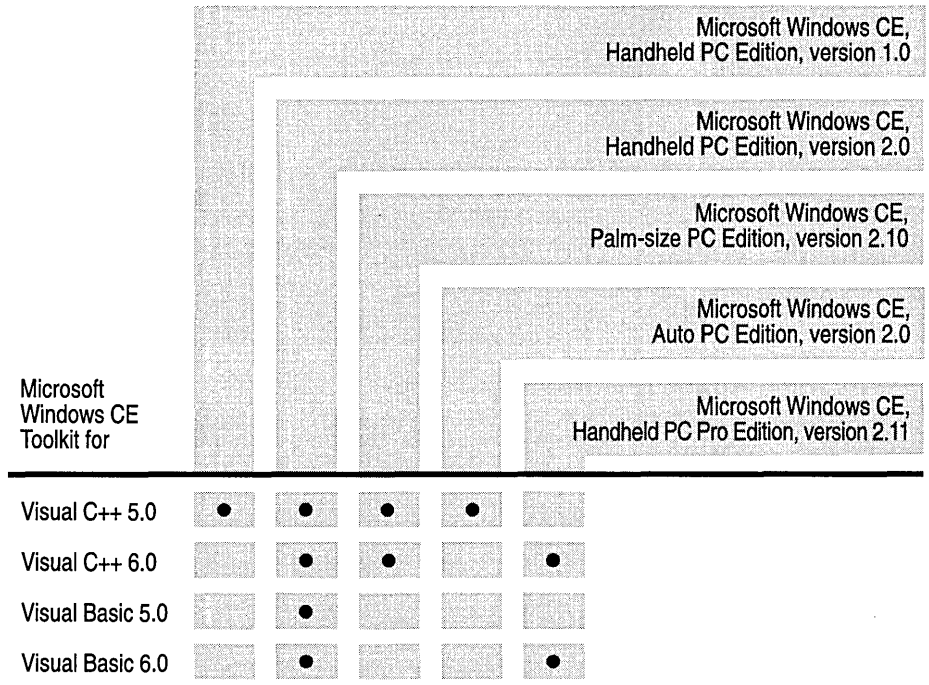
Microsoft provides several toolkits to assist you in developing Windows CE–based applications. These toolkits include include the Microsoft® Windows® CE Toolkit for Visual C++® 6.0 and the Microsoft® Windows® CE Toolkit for Visual Basic® 6.0. The toolkits are add-ins to the Microsoft® Visual C++® and Microsoft® Visual Basic® development systems, which means that they use the same IDE used to develop desktop applications. Microsoft packages the toolkits along with emulators to enable you to develop applications on a desktop computer.

Windows CE–based devices run different versions of the Windows CE OS and therefore support different toolkits. For example, if you want to use Visual Basic to create an application for an H/PC running Handheld PC Pro Edition software, you need the following products:

- Microsoft Visual Basic 6.0 development system
- Microsoft Windows CE Toolkit for Visual Basic 6.0
- Microsoft Windows CE Platform SDK, Handheld PC Edition, version 2.0

Both the development system and the toolkit are available through standard retail channels; the SDK is distributed on the Windows CE Web site.

The following table shows the toolkits that are available for each platform.



Windows CE–based devices span the home entertainment, vertical device, and PC companion markets. In the home entertainment market, products that run Windows CE include the Sega Dreamcast system, Internet set-top boxes, and Web telephones. In the vertical device market, embedded systems developers provide custom-built computers designed for special tasks, such as package and mail tracking devices, point-of-sale terminals, and navigation devices. In the PC companion market, products that run Windows CE include the H/PC, the Palm-size PC, and the Auto PC.

Each device category supports a different set of APIs. Within each device category, what is supported depends on the version of the OS that the device is built on and what modules and components are included. In addition, each device category contains a unique shell with its supporting APIs. Therefore, a Windows CE–based platform can contain APIs that are not included in the core Windows CE OS.

Additionally, Windows CE differs based on how it is ported to a device. While all H/PCs of a particular version may have the same set of functions, the functions available on a Palm-size PC differ from those on an H/PC. In addition, OEMs have the option of removing optional sections of the OS, so configuration of the OS running on a specific device can vary significantly.

Device Driver Development

Windows CE supports a wide range of device drivers that you can customize for various Windows CE–based platforms. Windows CE provides several models for device driver development, including models from other operating systems. Because of this diversity of device driver models, Windows CE accommodates almost all device types, including native devices and peripheral devices.

To assist you in creating custom device drivers, Platform Builder and all platform-specific SDKs include a device driver kit, programming guidelines, an API reference, and code samples that you can use to create device drivers for Windows CE.

PART 1

Core Services

This part contains the following chapters:

- **Working with Processes and Threads**
- **Managing Program Memory**
- **Accessing the Object Store, Database, and Registry**
- **Integrating Engines into an Application**

CHAPTER 2

Working with Processes and Threads

All Windows CE–based applications consist of a *process* and one or more *threads*. A process is a single instance of a running application. A thread is the basic unit to which the Windows CE operating system (OS) allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

Processes enable users to open and work in several applications at the same time. For example, a user can edit a file in a word processing application while another application is recalculating a spreadsheet. Threads enable an application to perform more than one task at a time even though applications cannot execute more than one thread at a time; however, Windows CE supports *preemptive multitasking*, which creates the effect of simultaneously executing multiple threads. When a process has more than one thread running, the OS rapidly switches from one thread to another so that the threads appear to run simultaneously.

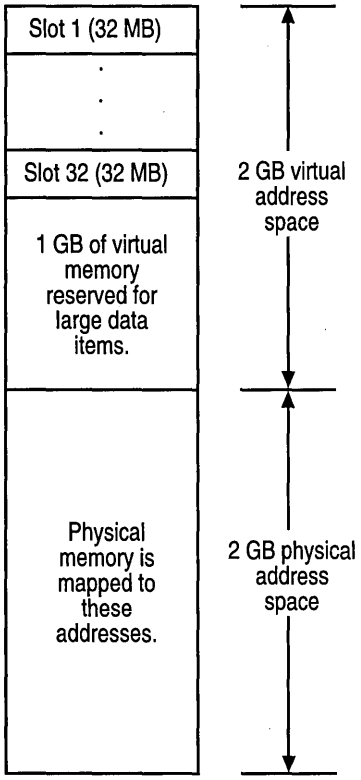
Processes

Windows CE can run up to 32 processes at one time. Each process starts with a single thread—often called the "primary thread"—which provides the resources that are required to run an application. Windows CE creates a primary thread when the loader calls the **WinMain** function. A process can also create additional threads when needed. The number of additional threads a process can create is limited only by the amount of random access memory (RAM) available on the device.

When the Windows CE OS initializes, it creates a single 4-gigabyte (GB) virtual address space. The address space is divided into 33 "slots," and each slot is 32 megabytes (MBs). The address space is shared by all processes. When a process initializes, Windows CE selects an open slot for the process in the system's address space. Slot zero is reserved for the currently running process.

In addition to assigning a slot for each process, Windows CE creates a stack for the thread and a heap for the process. Each stack has an initial size of at least 1 kilobyte (KB), which is committed on demand. The amount of stack space that is reserved by the system is specified in the (/STACK) option for the linker. Because the stack size is CPU-dependent, on some devices the system allocates 4 KB for each stack. The maximum number of threads is dependent upon the amount of available memory. You can allocate additional memory, outside of the 32 MB that is assigned to each slot, by using memory-mapped files or by calling the **VirtualAlloc** function.

The following illustration shows how memory is allocated in the Windows CE address space.



When a process initializes, the OS stores in the slot that is assigned to the process all of the dynamic-link libraries (DLLs), the stack, the heap, the application code, and the data section for each process. DLLs are loaded at the top of the slot, followed by the stack, the heap, and the executable file (.exe). The bottom 64 KB is always left free. DLLs are controlled by the loader, which loads all DLLs at the same address for each process.

Creating a Process

To start a process from within another process, call the **CreateProcess** function, which loads a new application into memory and creates a new process with at least one new thread.

The following code example shows the **CreateProcess** function prototype.

```
BOOL CreateProcess(LPCTSTR lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles, DWORD dwCreationFlags, LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory, LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation );
```

Because Windows CE does not support security or current directories and does not handle inheritance, the majority of the parameters must be set to NULL or 0. The following code example shows how the function prototype would look when all nonsupported features are taken into consideration.

```
BOOL CreateProcess(LPCTSTR lpApplicationName,
LPTSTR lpCommandLine, NULL, NULL, FALSE,
DWORD dwCreationFlags, NULL, NULL, NULL,
LPPROCESS_INFORMATION lpProcessInformation );
```

The first parameter, *lpApplicationName,* must contain a pointer to the name of the application to start. Windows CE does not support passing NULL for *lpApplicationName* and looks for the application in the following directories, in the following order:

1. The path that is specified in *lpApplicationName*, if one is listed.
2. An OEM-specified search path.
3. The Windows directory (\Windows).
4. The root directory in the object store (\).

The *lpCommandLine* parameter specifies the command line to pass to the new process. The command line must be passed as a Unicode string. The *dwCreationFlags* parameter specifies the initial state of the process after loading.

The following table describes all of the supported flags.

Flag	Description
0	Creates a standard process.
CREATE_SUSPENDED	Creates a process with a suspended primary thread.
DEBUG_PROCESS	Creates a process to be debugged by the calling process.
DEBUG_ONLY_THIS_PROCESS	Creates a process to be debugged by the calling process, but does not debug any child processes that are launched by the process being debugged. This flag must be used in conjunction with DEBUG_PROCESS.
CREATE_NEW_CONSOLE	Creates a new console.

The last parameter used by **CreateProcess** is *lpProcessInformation*. This parameter points to the **PROCESS_INFORMATION** structure, which contains data about the new process. The parameter can also be set to NULL.

If the process cannot run, **CreateProcess** returns FALSE. For more information about the failure, call the **GetLastError** function.

Terminating a Process

The most common way to terminate a process is to have it return from a **WinMain** function call. You can also terminate a process by having the primary thread of the process call the **ExitThread** function. A Windows CE process automatically terminates if its primary thread is terminated, even if there are other active threads in existence for the process. **ExitThread** returns the exit code of the process. You can determine the exit code of a process by calling the **GetExitCodeProcess** function. Specify the handle to the process, which you can obtain by calling the **CreateProcess** or **OpenProcess** function; the function returns the exit code. If the process is still running, the function returns the STILL_ACTIVE termination status.

There are also other, less common, ways of terminating a process:

- Use interprocess synchronization to instruct the process to terminate itself.

  For more information on using interprocess communication to terminate a process, see "Interprocess Synchronization."
- If the process has a message queue, send a WM_CLOSE message to the main window of the process. An application might not close if it does not receive this message and might display a message box.
- Use the **TerminateProcess** function, which does not notify any attached DLLs that the process is terminating. This method should be used as a last resort.

Note A process immediately terminates if a related secondary thread generates an unhandled exception. This is a change in behavior from Windows CE version 2.10 or earlier.

Threads

A thread describes a path of execution within a process. Every time the OS creates a new process, it also creates at least one thread. The purpose of creating a thread is to make use of as much of the CPU's time as possible. For example, in many applications, it is useful to create a separate thread to handle printing tasks so that the user can continue to use the application while it is printing.

Each thread shares all of the process's resources, including its address space. Additionally, each thread has a stack, where the linker sets the stack size for all of the threads that are created in a process (/STACK). A thread also contains the state of the CPU registers, known as the "context," and an entry in the execution list of the system scheduler. You can use the **GetThreadContext** function to retrieve the context of the specified thread and the **SetThreadContext** function to set the context of the specified thread.

Each thread in a process operates independently. Unless you make the threads visible to each other, they execute individually and are unaware of the other threads in a process. Threads sharing common resources, however, must coordinate their work by using a method of synchronization.

An application starts when the system scheduler gives one of its threads execution control. The system scheduler determines which threads should run and when they should run. Threads of lower priority may have to wait while higher priority threads complete their tasks.

Threads can be in one of the following states: running, suspended, sleeping, blocked, or terminated. When all threads are in the blocked state, Windows CE enters the suspended mode, which stops the CPU from executing instructions and consuming power. To conserve power, be sure to use synchronization objects to block threads that are waiting, instead of creating a thread that polls for status, such as the **PeekMessage** function.

Creating and Terminating a Thread

To create a thread, call the **CreateThread** function. The following code example shows the **CreateThread** function prototype.

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId );
```

Because Windows CE does not support the *lpThreadAttributes* and *dwStackSize* parameters, you must set them to NULL or 0. The following table describes the remaining **CreateThread** parameters.

Parameter	Description
lpStartAddress	Points to the start of the thread routine
lpParameter	Specifies an application-defined value that is passed to the thread routine
dwCreationFlags	Set to 0 or CREATE_SUSPENDED
lpThreadId	Points to a **DWORD** that receives the new thread's identifier

If **CreateThread** is successful, it returns the handle to the new thread and the thread identifier. You can also retrieve the thread identifier by calling the **GetCurrentThreadId** function from within the thread. In Windows CE, the value returned in **GetCurrentThreadId** is the actual thread handle. You can also retrieve a handle to the thread by calling the **GetCurrentThread** function. This function returns a pseudo-handle to the thread that is valid only while in the thread. If you specify CREATE_SUSPENDED in the *dwCreationFlags* parameter, the thread is created in a suspended state and must be resumed with a call to the **ResumeThread** function.

You can terminate a thread by calling **ExitThread**, which frees the resources that are used by a thread when they are no longer needed. Calling **ExitThread** for an application's primary thread causes the application to terminate.

Scheduling a Thread

Windows CE uses a priority-based time-slice algorithm to schedule the execution of threads. Because Windows CE does not have priority classes, the process in which the thread runs does not influence thread priorities. All the priorities can be used in the same process. A thread can have one of the eight priorities. The following table describes these priorities.

Priority	Description
THREAD_PRIORITY_TIME_CRITICAL	Indicates 3 points above normal priority
THREAD_PRIORITY_HIGHEST	Indicates 2 points above normal priority
THREAD_PRIORITY_ABOVE_NORMAL	Indicates 1 point above normal priority
THREAD_PRIORITY_NORMAL	Indicates normal priority
THREAD_PRIORITY_BELOW_NORMAL	Indicates 1 point below normal priority
THREAD_PRIORITY_LOWEST	Indicates 2 points below normal priority
THREAD_PRIORITY_ABOVE_IDLE	Indicates 3 points below normal priority
THREAD_PRIORITY_IDLE	Indicates 4 points below normal priority

Threads with a higher priority run first. Threads with the same priority run in a round-robin fashion—when a thread has stopped running, all other threads of the same priority run before the original thread can continue. Threads at a lower priority do not run until all threads with a higher priority have either finished or have been blocked. If one thread is running and a thread of higher priority is unblocked, the lower-priority thread is immediately suspended and the higher-priority thread is scheduled.

Threads run for a specific slice of time—called a quantum—which has a default value of 25 milliseconds. An OEM can specify a different quantum. If, after the quantum has elapsed, the thread has not relinquished its time slice and is not time-critical, it is suspended and another thread is scheduled to run. Threads having a priority level of THREAD_PRIORITY_TIME_CRITICAL cannot be preempted except by an interrupt service routine (ISR).

For the most part, thread priorities are fixed and do not change. However, there is one exception, called *priority inversion*. If a low-priority thread is using a resource that a high-priority thread is waiting to use, the kernel temporarily boosts the priority of the low-priority thread until it releases the resource that is required by the higher-priority thread.

To query the priority level of a thread, call the **GetThreadPriority** function. To change the priority level of a thread, call the **SetThreadPriority** function.

Suspending a Thread

You can suspend a thread at any time by using the **SuspendThread** function. You can also perform additional tasks on a thread. The following table describes these tasks.

To	Call
Resume running a thread	**ResumeThread**
Suspend a thread for a specified number of milliseconds	**Sleep**
Profile the performance of a thread and return how much time the thread has run	**GetThreadTimes**

When suspending a thread more than once, you must match multiple calls to **SuspendThread** with the same number of calls to **ResumeThread**.

Using Thread Local Storage

Thread local storage (TLS) is the method by which each thread in a multithreaded process allocates a location in which to store thread-specific data. There are several situations in which you may want a thread to access unique data. One example might include a spreadsheet application that creates a new instance of the same thread each time the user opens a new spreadsheet. The DLL that provides the functions for various spreadsheet operations can use TLS to save data about the current state of each spreadsheet.

TLS uses a TLS array to save thread-specific data. When a process is created, Windows CE allocates a 64-slot array for each running process. When a DLL attaches to a process, the DLL calls the **TlsAlloc** function, which looks through the array to find a free slot. The function then marks the slot "in use" and returns an index value to the newly assigned slot. If no slots are available, the function returns –1. Individual threads cannot call **TlsAlloc**. Only a process or DLL can call the function and it must do so before creating the threads that will use the TLS slot.

Once a slot has been assigned, each thread can access its unique data by calling the **TlsSetValue** function to store data in the TLS slot, or the **TlsGetValue** function to retrieve data from the slot.

The following table describes the TLS functions that are supported by Windows CE.

Function	Description
TlsAlloc	Allocates a TLS index. The index is available to any thread in the process for storing and retrieving thread-specific values. You must store this index in global memory, where all threads can retrieve its value.
TlsFree	Releases the TLS index, making it available for reuse.
TlsGetValue	Retrieves the value that is pointed to by the TLS index.
TlsSetValue	Stores a value in the slot that is pointed to by the TLS index.

Synchronizing Processes and Threads

In an OS where several threads run concurrently, it is important to be able to synchronize the activities of various threads. Windows CE provides several synchronization objects that enable you to synchronize a thread's actions with those of another thread. These objects include: critical sections, mutexes, and events. Additionally, you can use interlocked functions to synchronize a thread.

Regardless of the synchronization method that is used, a thread synchronizes itself with another thread by releasing a synchronization object and then entering a wait state. The synchronization object tells the OS what special event has to occur before the thread can resume execution. When the event occurs, the thread is again eligible to be scheduled for CPU time. Once it is scheduled, the thread continues executing. The thread has now synchronized its execution with the occurrence of the event.

Critical Section Objects

When multiple threads have shared access to the same data, the threads can interfere with one another. A critical section object protects a section of code from being accessed by more than one thread. A critical section is limited, however, to only one process or DLL and cannot be shared with other processes.

Critical sections work by having a thread call the **EnterCriticalSection** function to indicate that it has entered a critical section of code. If another thread calls **EnterCriticalSection** and references the same critical section object, it is blocked until the first thread calls the **LeaveCriticalSection** function. A critical section can protect more than one section of code as long as each section of code is protected by the same critical section object.

To use a critical section, you must first declare a **CRITICAL_SECTION** structure. Because other critical section functions require a pointer to this structure, be sure to allocate it within the scope of all functions that are using the critical section. Then, create a handle to the critical section object by calling the **InitializeCriticalSection** function.

To request ownership of a critical section, call **EnterCriticalSection**; to release ownership, call **LeaveCriticalSection**. When you are finished with a critical section, call the **DeleteCriticalSection** function to release the system resources that were allocated when you initialized the critical section.

The following code example shows the prototype for the critical section functions. Notice that they all require a pointer to the **CRITICAL_SECTION** structure.

```
void InitializeCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
void EnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
void LeaveCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
void DeleteCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
```

The following code example shows how a thread initializes, enters, and leaves a critical section. This example uses the **try-finally** structured exception-handling syntax to ensure that the thread calls **LeaveCriticalSection** to release the critical section object.

```
void CriticalSectionExample (void)
{
  CRITICAL_SECTION csMyCriticalSection;

  InitializeCriticalSection (&csMyCriticalSection);

  __try
  {
    EnterCriticalSection (&csMyCriticalSection);

    // Your code to access the shared resource goes here.
  }
  __finally
  {
    // Release ownership of the critical section
    LeaveCriticalSection (&csMyCriticalSection);
  }
} // End of CriticalSectionExample code
```

Mutex Objects

A *mutex object* is a synchronization object whose state is set to signaled when it is not owned by any thread and non-signaled when it is owned. Its name comes from its usefulness in coordinating mutually exclusive access to a shared resource. Only one thread at a time can own a mutex object.

A thread calls the **CreateMutex** function to create a mutex object. The creating thread can request immediate ownership of the mutex object as well as specify a name for the mutex object. Threads in other processes can open a handle to an existing mutex object by specifying the object's name in a call to **CreateMutex**. If the mutex object already exists, **GetLastError** returns ERROR_ALREADY_EXISTS. For more information about names for mutex objects and event objects, see "Interprocess Synchronization."

Any thread with a handle to a mutex object can use a wait function to request ownership of the mutex object. If the mutex object is owned by another thread, the wait function blocks the requesting thread until the owning thread releases the mutex object by calling the **ReleaseMutex** function. The return value of the wait function indicates whether the function returned for some reason other than that the state of the mutex is set to signaled. For more information about wait functions, see "Wait Functions."

Once a thread owns a mutex object, it can specify the same mutex object in repeated calls to one of the wait functions without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex object that it already owns. To release its ownership under such circumstances, the thread must call **ReleaseMutex** once for each time that the mutex object satisfied the conditions of a wait function.

If a thread terminates without releasing its ownership of a mutex object, the mutex object is considered to be abandoned. A waiting thread can acquire ownership of an abandoned mutex object, but the wait function's return value indicates that the mutex object is abandoned. To be safe, you should assume that an abandoned mutex object indicates that an error has occurred and that any shared resource being protected by the mutex object is in an undefined state. If the thread proceeds as though the mutex object had not been abandoned, the object's abandoned flag is cleared when the thread releases its ownership. This restores typical behavior, if a handle to the mutex object is subsequently specified in a wait function.

The following code example shows how to call **CreateMutex** to create a named mutex object.

```
void NamedMutexExample (void)
{
  HANDLE  hMutex;
  TCHAR szMsg[100];

  hMutex = CreateMutex (
              NULL,                        // No security descriptor
              FALSE,                       // Mutex object not owned
              TEXT("NameOfMutexObject")); // Object name

  if (NULL == hMutex)
  {
    // Your code to deal with the error goes here.

    // Here is one example of what might be done.
    wsprintf (szMsg, TEXT("CreateMutex error: %d."), GetLastError ());
    MessageBox (NULL, szMsg, TEXT("Error"), MB_OK);
  }
  else
  {
    // Not an error -- deal with success
    if ( ERROR_ALREADY_EXISTS == GetLastError () )
      MessageBox (NULL, TEXT("CreateMutex opened existing mutex."),
                  TEXT("Results"), MB_OK);
    else
      MessageBox (NULL, TEXT("CreateMutex created new mutex."),
                  TEXT("Results"), MB_OK);
  }
} // End of NamedMutexExample code
```

The following code example opens a handle of an existing mutex object. Additionally, it uses the **try-finally** structured exception-handling syntax to ensure that the thread properly releases the mutex object. To prevent the mutex object from being abandoned inadvertently, the **finally** block of code executes no matter how the **try** block terminates, unless the **try** block includes a call to the **TerminateThread** function.

```
BOOL WriteToDatabase (HANDLE hMutex)
{
  DWORD dwWaitResult;

  dwWaitResult = WaitForSingleObject (hMutex,    // Handle of mutex
object
                                      5000L);    // Five-second time-out
  switch (dwWaitResult)
```

```
  {
    case WAIT_OBJECT_0:
      __try
      {
        // Your code to write to the database goes here.
      }
      __finally
      {
        // Your code to clean up the database operations goes here.

        if (! ReleaseMutex (hMutex))
        {
          // Your code to deal with the error goes here.
        }
      }
      break;

    // Cannot get mutex object ownership due to time-out
    case WAIT_TIMEOUT:
      return FALSE;

    // Got ownership of an abandoned mutex object
    case WAIT_ABANDONED:
      return FALSE;
  }

  return TRUE;
} // End of WriteToDatabase example code
```

Event Objects

Windows CE uses event objects to notify a thread when to perform its task or to indicate that a particular event has occurred. For example, a thread that writes to a buffer resets the event object to signaled when it has finished writing. By using an event object to notify the thread that its task is finished, the thread can immediately start performing other tasks.

To create an event object, call the **CreateEvent** function. The following code example shows the **CreateEvent** function prototype.

```
HANDLE CreateEvent (NULL, BOOL bManualReset, BOOL bInitialState,
                    LPTSTR lpName);
```

Use the *lpName* parameter to name the event object or to leave the object unnamed. The *bManualReset* parameter enables you to specify whether the event object automatically resets itself from a signaled state to a nonsignaled state or whether it will require a manual reset. Use the *bInitialState* parameter to specify whether the event object is created in the signaled or nonsignaled state.

Named events can be shared with other processes. Threads in other processes can open a handle to an existing event object by specifying its name in a call to **CreateEvent**. All named synchronization objects are stored in the same queue. To determine whether a **CreateEvent** function created a new object or opened an existing object, you can call the **GetLastError** function immediately after calling **CreateEvent**. If **GetLastError** returns ERROR_ALREADY_EXISTS, the call opened an existing event. In other words, if the name specified in a call to **CreateEvent** matches the name of an existing event object, the function returns the handle of the existing object. When you use this technique for event objects, none of the calling processes should request immediate ownership of the event, because it is uncertain which process actually gets ownership.

To signal an event, use the **SetEvent** or **PulseEvent** function. **SetEvent** does not automatically reset the event object to a nonsignaled state. **PulseEvent** signals the event, and then it resets the event. For manual events, **PulseEvent** unblocks all threads waiting on the event. For automatic events, **PulseEvent** unblocks only one thread.

If an event object can reset itself, you need only use **SetEvent** to signal. The event object is then automatically reset to the nonsignaled state when one thread is unblocked after waiting on that event. To manually reset an event, use the **ResetEvent** function.

To close an event object, call the **CloseHandle** function. If the event object is named, Windows CE maintains a use count on the object, and you must make one call to **CloseHandle** for each call to **CreateEvent**.

A single thread can specify different event objects in several simultaneous overlapped operations. If this is the case, use one of the multiple-object wait functions to wait for the state of any one of the event objects to be signaled. You can also use event objects in a number of situations to notify a waiting thread of the occurrence of an event. For example, communications devices use an event object to signal their completion.

The following code example shows an application that uses event objects to prevent several threads from reading from a shared memory buffer while a master thread is writing to that buffer. The master thread uses the **CreateEvent** function to create a manual-reset event object. The master thread sets the event object to nonsignaled when it is writing to the buffer, and it then resets the object to signaled when it has finished writing. The master thread then creates several reader threads and an auto-reset event object for each thread. Each reader thread sets its event object to signaled when it is not reading from the buffer.

```
VOID ReadThreadFunction (LPVOID lpParam);  // Forward declaration

#define NUMTHREADS 4

HANDLE  hGlobalWriteEvent;
HANDLE  hReadEvents[NUMTHREADS];

void CreateEventsAndThreads (void)
{
  HANDLE hThread;   // Newly created thread handle
  DWORD dwThreadID; // Newly created thread ID value
  int i;            // For-loop counter

  hGlobalWriteEvent = CreateEvent (NULL,  // No security attributes
                                   TRUE,  // Manual-reset event
                                   TRUE,  // Initial state is signaled
                                   TEXT("WriteEvent"));
                                          // Object name
  if (hGlobalWriteEvent == NULL)
  {
    // Your code to deal with the error goes here.
  }

  for (i = 0; i < NUMTHREADS; i++)
  {
    hReadEvents[i] = CreateEvent (NULL,   // No security attributes
                                  FALSE,  // Auto-reset event
                                  TRUE,   // Initial state is signaled
                                  NULL);  // Object not named
    if (hReadEvents[i] == NULL)
    {
      // Your code to deal with the error goes here.
    }
```

```
    hThread = CreateThread (
                NULL,                   // No security attributes
                                        // in Windows CE
                0,                      // Must be 0 under Windows CE
                (LPTHREAD_START_ROUTINE) ReadThreadFunction
                &hReadEvents[i],        // Pass new thread's event handle
                0,                      // Thread runs immediately
                &dwThreadID);           // Returned ID value (ignored)

    if (hThread == NULL)
    {
      // Your code to deal with the error goes here.
    }
  }
} // End of CreateEventsAndThreads example code
```

In the following code example, before the master thread writes to the shared buffer, it uses **ResetEvent** to set the state of *hGlobalWriteEvent*, which is an application-defined global variable, to nonsignaled. This blocks the reader threads from starting a read operation. The master thread then uses the **WaitForSingleObject** function to wait for all reader threads to finish any current read operations. When the loop of calls to **WaitForSingleObject** is completed, the master thread can safely write to the buffer. After it has finished writing, it sets *hGlobalWriteEvent* and all the reader-thread events to signaled, which enables the reader threads to resume their read operations. The example does not use the **WaitForMultipleObjects** function because in Windows CE the *fWaitAll* flag must be set to FALSE. For more information about wait functions, see "Wait Functions."

```
int WriteToBuffer (void)
{
  DWORD dwWaitResult;
  int i;

  // Block all read threads from starting any new operations.
  if (!ResetEvent (hGlobalWriteEvent))
  {
    // Your code to deal with the error goes here.
  }

  // Wait for all of the read events to be signaled (threads done).
  for (i = 0; i < NUMTHREADS; i++)
  {
    dwWaitResult = WaitForSingleObject (hReadEvents[i], INFINITE);
    if (WAIT_OBJECT_0 != dwWaitResult)
    {
      // Your code to deal with the error goes here.
    }
  }
```

```
  // Now it is okay to write to the shared buffer, so that code
  // goes here.


  // Put all the events back to signaled (so read threads can run).
  for(i = 0; i < NUMTHREADS; i++)
  {
    if (!SetEvent (hReadEvents[i]))
    {
      // Your code to deal with the error goes here.
    }
  }

  if (!SetEvent (hGlobalWriteEvent))
  {
    // Your code to deal with the error goes here.
  }

  return 1;
} // End of WriteToBuffer example code
```

In the following code example, before starting a read operation, each reader thread uses the **WaitForSingleObject** function to wait for the application-defined global variable, *hGlobalWriteEvent*, and then uses it again to wait for its own read event to be signaled. When the loop of calls to **WaitForSingleObject** completes, the reader thread's auto-reset event has been reset to nonsignaled. This blocks the master thread from writing to the buffer until the reader thread uses **SetEvent** to set the event's state back to signaled.

```
VOID ReadThreadFunction (LPVOID lpParam)
{
  DWORD dwWaitResult;
  HANDLE hEvent;
  BOOL bStayInLoop;

  hEvent = * (HANDLE *) lpParam;   // This thread's read event

  bStayInLoop = TRUE;

  while (bStayInLoop)
  {
    // First, wait for the master thread's event.
    WaitForSingleObject (hGlobalWriteEvent, INFINITE);
    if (WAIT_OBJECT_0 != dwWaitResult)
    {
      // Your code to deal with the error goes here.
    }
```

```
    // Now, wait for this thread's event object.
    WaitForSingleObject (hEvent, INFINITE);
    if (WAIT_OBJECT_0 != dwWaitResult)
    {
      // Your code to deal with the error goes here.
    }

    // Now it is okay to read the shared buffer -- the code to do
    // that goes here.  When it is time for the thread to quit, set
    // the bStayInLoop variable to FALSE.

    // Now, signal that this thread is done with the buffer.
    if (!SetEvent (hEvent))
    {
      // Your code to deal with the error goes here.
    }
  }
} // End of ReadThreadFunction example code
```

Wait Functions

Windows CE supports single-object and multiple-object wait functions that block or unblock a thread based on the signaled or nonsignaled state of the object. The single object function is **WaitForSingleObject**. The multiple object functions are **WaitForMultipleObjects** and **MsgWaitForMultipleObjects**.

The **WaitForSingleObject** function enables a thread wait on a single object. The object can be a synchronization object, such as an event or mutex, or the object can be a handle to a process and thread. Handles are signaled when their processes or threads terminate. Thus, a process can monitor another process or thread and perform some action based on the status of the process or thread.

The **WaitForMultipleObjects** and **MsgWaitForMultipleObjects** functions enable the calling thread to specify an array containing one or more synchronization object handles. These functions return when one of the following events occurs:

- The state of any of the specified objects is set to signaled or the states of all objects have been set to signaled.
- The time-out interval elapses. You can set the time-out interval to INFINITE to specify that the wait will not time out.

Windows CE does not support waiting for all objects to be signaled.

The following code example shows how to use **CreateEvent** to create two event objects. It then uses the two created objects as parameters in the function call to **WaitForMultipleObjects**. **WaitForMultipleObjects** does not return until one of the objects is set to signaled.

```
int EventsExample (void)
{
  HANDLE hEvents[2];
  DWORD dwEvent;
  int i;

  for (i = 0; i < 2; i++)
  {
    hEvents[i] = CreateEvent (NULL,   // No security attributes
                              FALSE,  // Auto-reset event object
                              FALSE,  // Initial state is nonsignaled
                              NULL);  // Unnamed object

    if (hEvents[i] == NULL)
    {
      // Your error handling code goes here.
      MessageBox (NULL, TEXT("CreateEvent error!"),
                  TEXT("Error"), MB_OK);

      // You can use GetLastError to obtain more information.

      return 0;
    }
  }

  // Put code that uses the events here; that is, startup
  // of threads, which will set the events when completed.
  // . . .

  dwEvent = WaitForMultipleObjects (
                      2,            // Number of objects in an array
                      hEvents,      // Array of objects
                      FALSE,        // Wait for any (required
                                    // under Windows CE)
                      INFINITE);    // Indefinite wait

  switch (dwEvent)
  {
    case WAIT_OBJECT_0 + 0:         // First event was signaled
    case WAIT_OBJECT_0 + 1:         // Second event was signaled
      break;
```

```
    default:
      // Your error handling code goes here.
      MessageBox (NULL, TEXT("Wait error!"),
                  TEXT("Error"), MB_OK);

      // You can use GetLastError to obtain more information.

      return 0;
  }

  return 1;
} // End of EventsExample code
```

MsgWaitForMultipleObjects is similar to **WaitForMultipleObjects**, except that it enables you to specify input event objects in the object handle array. You select the type of input event to wait for in the *dwWakeMask* parameter. **MsgWaitForMultipleObjects** does not return if there is unread input of the specified type in the queue. It returns only when new input arrives.

For example, a thread can use **MsgWaitForMultipleObjects** with its *dwWakeMask* parameter set to QS_KEY. This blocks its execution until the state of a specified object has been set to signaled and there is keyboard input available in the thread's input queue. The thread can use the **GetMessage** or **PeekMessage** function to retrieve the input.

The following code example shows how to use **MsgWaitForMultipleObjects** in a message loop. The loop continues until a WM_QUIT message appears in the queue. The *dwWakeMask* parameter is set to QS_ALLINPUT so that all messages are checked.

```
int MessageLoop (
      HANDLE* lphObjects,   // Handles that need to be waited on
      int     iObjCount )   // Number of handles to wait on
{
  while (TRUE)
  {
    // Block-local variables
    DWORD dwResult;
    MSG msgCurrent;

    while (PeekMessage (&msgCurrent, NULL, 0, 0, PM_REMOVE))
    {
      if (WM_QUIT == msgCurrent.message)
        return 1;

      DispatchMessage (&msgCurrent);
    }
```

```
    dwResult = MsgWaitForMultipleObjects (iObjCount, lphObjects,
                                FALSE, INFINITE, QS_ALLINPUT);

    if (dwResult == DWORD(WAIT_OBJECT_0 + iObjCount))
    {
      // Some input was received -- go around loop again
      continue;
    }
    else
    {
      // Check for errors and handle them here. If not an error,
      // write code which processes the result here.  Be sure to
      // create code that provides some way to get out of the loop!

      // The index for the signaled is
      // (dwResult - WAIT_OBJECT_0).
    }
  }
} // End of MessageLoop example code
```

Note Be careful when using the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all the windows in the system. If you have a thread that uses a wait function with no time-out interval, the system will deadlock. One example of code that indirectly creates a window is the Component Object Model (COM) **CoInitialize** function. If you have a thread that creates windows, use **MsgWaitForMultipleObjects** rather than the other wait functions.

Interlocked Functions

Interlocked functions synchronize access to a variable that is shared by multiple threads. Their purpose is to prevent a thread from being preempted when it is in the middle of incrementing or checking a variable. The threads of different processes can use these functions as long as their variables share memory. Windows CE supports three interlocked functions: **InterlockedIncrement**, **InterlockedDecrement**, and **InterlockedExchange**.

The following table describes the tasks that you can perform with each function.

To	Call
Increment a shared variable and check the resulting value	**InterlockedIncrement**
Decrement a shared variable and check the resulting value	**InterlockedDecrement**
Exchange the values of specified variables	**InterlockedExchange**
Exchange the values of specified variables only if one of the variables is equal to a specified value	**InterlockedTestExchange**

Interprocess Synchronization

All processes protect against the casual exchange of data; however, occasionally two processes may need to communicate with each other. One method that enables one process to communicate with another is called interprocess synchronization.

Because multiple processes can have handles to the same event or mutex object, these objects can be used to accomplish interprocess synchronization. The process that creates an object can use the handle returned by the **CreateEvent** or **CreateMutex** function. Other processes can open a handle to the object by using its name in another call to the **CreateEvent** or **CreateMutex** function.

Named objects provide a way for processes to share object handles. The name specified by the creating process is limited to the number of characters that are defined by MAX_PATH. It can include any character except the backslash (\) path-separator character. Once a process has created a named event or mutex object, other processes can use the name to call the appropriate function, either **CreateEvent** or **CreateMutex**, to open a handle to the object. Name comparison is case-sensitive.

The names of event and mutex objects share the same name space. If you specify a name that is in use by an object of another type when you create an object, the function succeeds, but **GetLastError** returns ERROR_ALREADY_EXISTS. To avoid this error, use unique names and be sure to check function return values for duplicate-name errors.

The following code example shows how to use object names by creating and opening named objects. The first process uses **CreateMutex** to create the mutex object. Note that the function succeeds even if there is an existing object with the same name.

```
HANDLE MakeMyMutex (void)
{
  HANDLE hMutex;

  hMutex = CreateMutex (
            NULL,                              // No security attributes
            FALSE,                             // Initially not owned
            TEXT("MutexToProtectDatabase"));  // Name of mutex object

  if (NULL == hMutex)
  {
    // Your code to deal with the error goes here.
  }

  return hMutex;
} // End of MakeMyMutex example code
```

Synchronization and Device I/O

Windows CE enables you to synchronously perform the **WriteFile**, **ReadFile**, and **WaitCommEvent** functions.

Windows CE does not support the overlapped I/O features of Windows NT. The *lpOverlapped* parameter to **ReadFile** or **WriteFile** must be NULL. Therefore, Windows CE cannot signal the event that is passed in when the I/O operation is completed. However, Windows CE does support simultaneous synchronous or asynchronous calls to **ReadFile** or **WriteFile** made by separate threads that are overlapped in time; this is not supported in Windows NT.

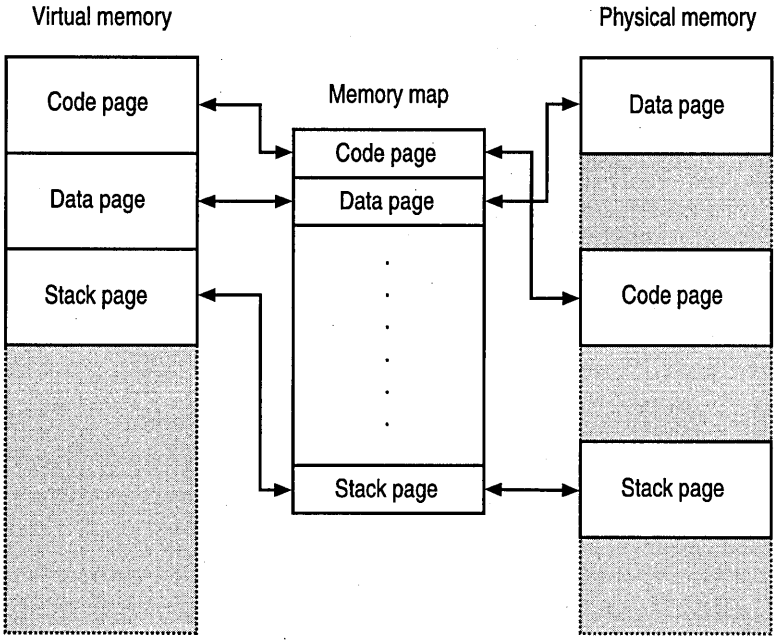
CHAPTER 3

Managing Program Memory

One of the main sections of random access memory (RAM) in a Windows CE–based device is called the *program memory*. With program memory, an application dynamically allocates memory for strings, structures, and other data. These allocations exist for the lifetime of the application and are deallocated while or after the application completes its task. As such, program memory fills the same function that RAM does in a desktop computer.

Windows CE uses a virtual memory system to manage and allocate program memory. Virtual memory is a style of memory management that separates how an application requests memory from the way that Windows CE provides memory. The application requests a block of virtual memory, and Windows CE maps that virtual memory address to a physical address. Subsequent memory allocations are not necessarily mapped to subsequent physical memory locations. Rather, Windows CE maps to whatever physical memory is available, regardless of the location of the memory on the device.

The following illustration shows how virtual memory maps to physical memory.



Virtual memory keeps the application from having to manage memory allocation. Because the application uses virtual memory addresses instead of physical memory addresses, the application sees contiguous memory. Windows CE has a total of 4 gigabytes (GB) of virtual address space. Of the 4-GB virtual address space, Windows CE reserves 32 slots of 32 megabytes (MB) to run processes. Each process receives one 32-MB slot for dynamic memory requirements, including process code segments, loaded dynamic-link libraries (DLLs), thread stacks, and created heaps. The rest of the virtual address space is reserved for the Windows CE operating system (OS). Windows CE uses these memory addresses for tasks such as memory mapped files and large virtual allocations, neither of which come out of the process slots.

Windows CE allocates virtual memory in pages. A page is a unit of memory that is either 1,024 or 4,096 bytes long. Windows CE marks each page in memory as free, reserved, or committed:

- A free page can be allocated to any application at any time.
- A reserved page is held for an application, but has not yet been mapped to physical memory.
- A committed page has been allocated to physical memory and is currently in use by an application.

Windows CE allocates pages along 64-kilobyte (KB) regions. Any function that reserves virtual memory pages automatically rounds up to the nearest 64-KB region. You can allocate memory more efficiently if you reserve virtual memory blocks in 64-KB regions. After you reserve the region, you can then go back and commit pages within the region, as needed.

Before allocating memory, use the **GetSystemInfo** and **GlobalMemoryStatus** functions to return information about your Windows CE–based device. **GetSystemInfo** returns information such as which microprocessor a device has, the memory page size, and the virtual addresses that are available to your application. **GlobalMemoryStatus** returns general information about memory allocation on the device. You can also use the **GetStoreInformation** function to determine how much memory is allocated to the system volume.

You can use the virtual memory application programming interface (API) to directly allocate virtual memory. The functions in the virtual memory API include **LocalAlloc**, **LocalFree**, **LocalReAlloc**, **VirtualFree**, **VirtualProtect**, and **VirtualQuery**. In addition, Windows CE uses the virtual memory in other memory allocations. Both the heap and the stack indirectly use virtual memory in their API sets. The fourth type of memory, the static data block, does not have a direct virtual memory address; rather, you place information in the static data block before compiling your application.

Using Virtual Memory

The virtual memory API directly allocates virtual memory. Windows CE also uses the virtual memory API to allocate memory for the heap and the stack. Use the virtual memory API when you need to reserve and allocate large blocks of memory for an application. The advantage of using virtual memory is that virtual memory does not fragment: Windows CE always allocates an integral number of pages. However, because Windows CE manages virtual memory in 64-KB units, you must ensure that you use all of the memory efficiently. Typically, an application wastes half of a virtual memory page per allocation. Also, Windows CE requires a slight overhead for managing the memory mapping. Finally, you must remember the reserved and committed status of each page. Therefore, if you do not think that you can use most of the 64-KB unit in your virtual memory allocation, use a heap instead.

▶ **To allocate and deallocate virtual memory**

1. Call the **VirtualAlloc** function to reserve and commit virtual memory.

   You can also use **VirtualAlloc** to reserve virtual memory and then have Windows CE map the memory address directly to physical memory when the pages are actually used by the application.

2. Use the virtual memory that is allocated to your application.

3. If necessary, call the **VirtualQuery** function to determine the read/write status of a virtual memory page.
4. If necessary, call the **VirtualProtect** function to alter the access rights on committed pages.
5. Call the **VirtualFree** function to decommit or free virtual memory and return the memory to Windows CE.

   Decomitting a page unmaps the page from physical memory but keeps the virtual addresses reserved for the application.

The following code example shows how to simultaneously reserve and commit virtual memory.

```
#define THIRTY_K  (30 * 1024) // Definition example

LPVOID lpMemory = NULL;       // Pointer to a region of memory
LPVOID lpPage = NULL;         // Pointer to a page of memory

lpMemory = VirtualAlloc (NULL, THIRTY_K,
                         MEM_RESERVE | MEM_COMMIT,
                         PAGE_READWRITE);

if (NULL == lpMemory)
{
  // Your error-handling code goes here. You can use the
  // GetLastError function to obtain more information.
}

// Your code that uses the just-allocated memory goes here.
// For example:     LPBYTE MyPointer = (LPBYTE)lpMemory;
//                  memset (MyPointer, 0, THIRTY_K);
//                  return MyPointer;
// . . .

// When you are finished with the memory, you must release it.
if (!VirtualFree (lpMemory, 0, MEM_RELEASE))
{
  // Your error-handling code goes here. You can use the
  // GetLastError function to obtain more information.
}

// Remember to clear the pointer to prevent re-use.
lpMemory = NULL;
```

The following code example shows how to reserve a bundle of memory and then commit that memory, as needed.

```
lpMemory = VirtualAlloc (NULL, THIRTY_K,
                         MEM_RESERVE, PAGE_READWRITE);

if (NULL == lpMemory)
{
  // Your error-handling code goes here. You can use the
  // GetLastError function to obtain more information.
}

// Determine the page size on this system.
SYSTEM_INFO siSystemInfo;

GetSystemInfo (&siSystemInfo);

// Commit the first page of the just-reserved block.
// (To commit more than one page, multiply the dwPageSize
// parameter used below by the number of pages to commit.)
lpPage = VirtualAlloc (lpMemory,        // Base for this commit
                       siSystemInfo.dwPageSize,
                                        // number of bytes
                                        // to commit
                       MEM_COMMIT,      // Selects the operation
                       PAGE_READWRITE); // Selects permissions

if (NULL == lpPage)
{
  // Your error-handling code goes here. You can use the
  // GetLastError function to obtain more information.
}

// To commit additional pages, add the appropriate
// offset to 'lpMemory' and call VirtualAlloc again.
// For example, to commit the tenth page:
// lpPage = VirtualAlloc ((LPBYTE)lpMemory + 9 * dwPageSize,
// and so on.
// . . .
```

The following code example shows how to use the **VirtualQuery** function to examine the characteristics of a page of virtual memory that you previously allocated.

```
DWORD dwResult;
MEMORY_BASIC_INFORMATION mbiMemory;

// Clear the results structure.
memset (&mbiMemory, 0, sizeof(MEMORY_BASIC_INFORMATION));

dwResult = VirtualQuery (lpPage,        // Page to examine
                         &mbiMemory,    // Structure for results
                         sizeof(MEMORY_BASIC_INFORMATION));

if (sizeof(MEMORY_BASIC_INFORMATION) != dwResult)
{
  // Your error-handling code goes here.
}

// Here, write code to use the contents of the mbiMemory
// structure that was filled in by the above call.
// For example: if (mbiMemory.Protect != PAGE_READWRITE)
// and so on.
// . . .
```

The following code example shows how to use **VirtualProtect** to change the characteristics of a page of virtual memory that you previously allocated.

```
DWORD dwOldProtect;
DWORD dwSize = 2048;  // Or, set this to 'siSystemInfo.dwPageSize'
                      // (See the above code for how to get this.)

if (!VirtualProtect (lpPage,        // Beginning of the set of
                                    // pages to change
                     dwSize,        // Length, in bytes, of the
                                    // set of pages to change
                     PAGE_READONLY, // What to change it to
                     &dwOldProtect  // Place to store the old setting
                     ))
{
  // Your error-handling code goes here. You can use the
  // GetLastError function to obtain more information.
}
```

The following code example shows how to decommit a page by using **VirtualFree**. Presumably, you could recommit the page later with different attributes.

```
if (!VirtualFree (lpPage, 0, MEM_DECOMMIT))
{
  // Your error-handling code goes here. You can use the
  // GetLastError function to obtain more information.
}
```

Using the Local Heap

A *heap* is a region of reserved virtual memory space that Windows CE manages for your application. The original heap is called the local heap. Unlike **VirtualAlloc**, you can allocate memory on a heap in 4-byte or 8-byte units, depending on your CPU type. In addition to being more efficient for small sizes, the heap can be used to avoid having to deal with the different sizes of memory pages that different microprocessors support. The heap also has no set limit. However, any heap allocations greater than 192 KB are fulfilled with a call to **VirtualAlloc**. If you attempt to allocate more memory than Windows CE originally set aside for a heap, the system attempts to find unreserved memory. Like virtual memory mapping, this memory might not be physically located next to the original heap. Also, because Windows CE allocates memory in fixed blocks, the heap might become fragmented over time.

Use the local heap when you need to allocate specific sizes of memory on a regular basis. Because Windows CE reclaims a memory page only if that page is totally free, be sure that you deallocate memory blocks correctly when you are using the heap. This becomes an issue with applications that run for a long time, such as applications that are designed for the Palm-size PC device category.

▶ **To allocate and deallocate memory using the local heap**

1. Call the **LocalAlloc** function with the size of the memory block passed in the *uBytes* parameter.

   LocalAlloc returns a handle to the virtual memory block that is allocated to your application. Windows CE also maps the virtual memory block to a physical memory block at this time.

2. Use the memory allocated to your application.

3. If necessary, call the **LocalSize** and **LocalReAlloc** functions to reallocate the local heap memory.

   If you need to increase the size of a block, call **LocalSize** to determine that the block contains enough space. Then, call **LocalReAlloc** to either add memory to the top of the allocation or to move the block to a larger area.

4. Call the **LocalFree** function to return the memory to Windows CE.

Note The **HeapAlloc** function can also allocate memory outside of the local heap by using the handle that is returned by the **GetProcessHeap** function.

The following code example shows how to allocate a set number of bytes from the local heap.

```
LPBYTE pbMyPointer = NULL;
HLOCAL hlHeapMemory = NULL;
HLOCAL hlHeapMem2 = NULL;
UINT uiSize;

hlHeapMemory = LocalAlloc (LPTR,    // FIXED and ZEROINIT
                           42);     // Number of bytes

if (hlHeapMemory == NULL)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}

// The return value is actually the address of the memory.
pbMyPointer = (LPBYTE)hlHeapMemory;

// Your code to use the memory goes here.
// SomeFunctionCall (pbMyPointer, and so on)
// . . .
```

The following code example shows how to check the size of a piece of the local heap by using **LocalSize**.

```
uiSize = LocalSize (hlHeapMemory);
if (uiSize == 0)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}
```

The following code example shows how to change the size of a piece of local heap by using **LocalReAlloc**.

```
hlHeapMem2 = LocalReAlloc (hlHeapMemory,  // Existing handle
                           100,           // New size, in bytes
                           0);            // Options (none)

if (hlHeapMem2 == NULL)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}

// The return value is actually the address of the memory.
pbMyPointer = (LPBYTE)hlHeapMem2;
```

The following code example shows how to free memory from the local heap.

```
hlHeapMem2 = LocalFree (hlHeapMem2);

// The pointer returned should be NULL.
if (hlHeapMem2 != NULL)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}
```

Using a Separate Heap

Instead of adding more size to the local heap, you might need to create a series of separate heaps. For example, a word processor might have a separate heap associated with each text file. Because you can delete separate heaps, fragmentation is not as much of an issue.

▶ **To allocate and deallocate memory on a separate heap**

1. Call the **HeapCreate** function.

   HeapCreate returns a handle. Use this handle to reference which heap you are using in the following steps.

2. Call **HeapAlloc** to allocate memory from the heap to your application.

   Windows CE does not place any limits on the size of a separate heap. As long as you have virtual and physical memory addresses available, you can allocate memory from the separate heap.

3. Use the memory that is allocated to your application.

4. If necessary, call **HeapSize** and **HeapReAlloc** to reallocate the separate heap memory.

   If you need to increase the size of a block, call **HeapSize** to determine if the block contains enough space. Then, call **HeapReAlloc** to either add memory to the top of the allocation or to move the block to a larger area.

5. Once you are done, call the **HeapFree** function to deallocate the memory back to Windows CE.

   Like the local heap, Windows CE deallocates a page only if all of the blocks within that page are also deallocated. This may result in fragmentation within the separate heap.

6. Once you are done with the heap, call the **HeapDestroy** function to return all of the heap memory to Windows CE.

   You do not have to call **HeapFree** on all blocks within the separate heap before you call **HeapDestroy**.

The following code example shows how to create a new heap.

```
LPBYTE pbMyPointer = NULL;
HANDLE hMyNewHeap = NULL;
LPVOID lpHeapMemory = NULL;
LPVOID lpHeapMem2 = NULL;
UINT uiSize;
BOOL bResult;

hMyNewHeap = HeapCreate (0,     // *do* serialize
                         35000, // 35,000 bytes wanted
                         0);    // Unsupported in Windows CE
```

The following code example shows how to allocate memory from the heap even when the heap is very small, by using **HeapAlloc**.

```
lpHeapMemory = HeapAlloc (hMyNewHeap,       // Specify which heap
                          HEAP_ZERO_MEMORY, // Zero the memory
                          42);              // Number of bytes
if (lpHeapMemory == NULL)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}

// The return value is actually the address of the memory.
pbMyPointer = (LPBYTE)lpHeapMemory;

// Your code to use the memory goes here.
// SomeFunctionCall (pbMyPointer, and so on)
// . . .
```

The following code example shows how to check the size of part of the heap by using **HeapSize**.

```
uiSize = HeapSize (hMyNewHeap,     // Specify the heap
                   0,              // No flags
                   lpHeapMemory); // Specify which memory
if (uiSize == 0)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}
```

The following code example shows how to change the size of part of the heap by using **HeapReAlloc**.

```
lpHeapMem2 = HeapReAlloc (hMyNewHeap,   // Specify which heap
                          0,            // Options (none)
                          lpHeapMemory, // Existing memory
                          100);         // New size, in bytes
if (lpHeapMem2 == NULL)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}

// The return value is actually the address of the memory.
pbMyPointer = (LPBYTE)lpHeapMem2;
```

The following code example shows how to free memory from the heap by using **HeapFree**.

```
bResult = HeapFree (hMyNewHeap,       // Specify which heap
                    0,                // Options (none)
                    lpHeapMem2);      // Specify what to free
if (!bResult)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}

// Always zero the caller's pointer after freeing memory.
lpHeapMem2 = NULL;
```

The following code example shows how to destroy the heap by using **HeapDestroy**.

```
bResult = HeapDestroy (hMyNewHeap);

if (!bResult)
{
  // Your error-handling code goes here. You can use
  // the GetLastError function to obtain more information.
}
```

Using the Stack

The stack is the storage area for variables that are referenced in a function. Windows CE allocates memory for a variable from the stack and deallocates the memory after the function call is complete. When a thread or process begins, Windows CE allocates one page of memory to the stack for that thread. Each thread has a stack, and each stack contains up to 60 KB of data: Windows CE reserves 58 KB for the stack, and reserves the final 2 KB for stack overflow control.

Note Exceeding the 60-KB limit for a stack causes a system-access violation that shuts down your application.

By default, you call on the stack each time that you declare a variable. You can also allocate memory from the stack by using the **HeapAlloc** function.

Using the Static Data Block

The static data block is a block of memory that Windows CE loads with an application. This block contains strings, buffers, and other static values that the application references throughout the life of the application. Windows CE allocates two sections for static data: one for read/write data and one for read-only data. Because Windows CE allocates the static data blocks one page at a time, you can often find unused memory between the end of the static data and the end of the memory page. Use a utility such as DumpBin.exe, or the Remote Memory viewer, to determine the size of your application's static data block. Use this information to arrange your declared data as efficiently as possible.

One way to use the unused memory in the static data blocks is to assign the unused memory to a buffer. Use this buffer in your application instead of creating a buffer dynamically. When assigning extra memory to a buffer, be sure to leave at least 64 KB free for the loader. Windows CE allocates another page for the loader if it cannot find enough space in the current static data page.

Another technique is to place a note in your code to remind you to look at your memory use each time that you add more data. For example, Windows CE might assign an entire memory page to your application when you only need room for a single variable. By modifying your application, you can save this page and thus lower your memory requirements.

You can also reduce the size of your data blocks by declaring data in the R/W sections as "const." Windows CE then moves all "const" data to the read-only section. Windows CE places all constant data items in the read-only data block instead of the R/W data block. However, if declaring all of your constant data forces Windows CE to allocate another page, you can move some of that data into the R/W block.

Identifying Low-Memory Situations

No matter how efficiently you allocate memory and how efficiently your application uses RAM, your device might run low on memory. At a programming level, a low-memory situation can manifest itself to the application in the following ways:

- A call to **VirtualAlloc** returns a 0, indicating that the function failed to allocate memory.
- Either **LocalAlloc** or **HeapAlloc** returns a 0, indicating that an attempt to increase the size of a heap has failed.
- Windows CE returns a stack fault error to your application, indicating that a stack allocation has failed.

Note Low-memory situations may be caused indirectly. For example, a call to the **CreateWindow** function can cause these types of memory failures.

To avoid the problems associated with low memory, Windows CE constantly monitors the amount of memory that is available and tries to prevent low-memory situations from occurring. It does this in several ways:

- When an application attempts to allocate memory, Windows CE filters the request. Filtering prevents a single application from using all of the available memory with one large allocation by lowering the maximum allocation limit.
- When Windows CE enters a low-memory situation, it lowers the memory limit further.

To recognize a low-memory situation, monitor the return values for your memory allocation functions. The platform you program on may also have additional functions to help you deal with low-memory situations. For more information, see the platform-specific documentation.

Sharing Memory-Mapped Objects Between Processes

You can use a memory-mapped object to share data between processes. However, do not create an unnamed object and pass a memory pointer to the different processes: one process can close the unnamed object without informing the other process. To avoid this memory error, Windows CE supports naming the unnamed object. Instead of passing a pointer to the object, you can now pass the name of the object. The other process then accesses the object through its name. Accessing the object through the name informs Windows CE which processes have access to the object. Windows CE then deletes the object only when both processes have closed the object.

▶ **To share data between processes by using memory-mapping**

1. Call the **CreateFileMapping** function to create the memory object, using the *lpName* parameter to pass in a name for the memory-mapped object.
2. Pass the name of the memory-mapped object to the process that you want to communicate with.
3. Call **CreateFileMapping** in the second process, using the name of the object that you passed with the first object.

   The name of the memory-mapped object is global. When the second process calls **CreateFileMapping**, Windows CE passes back the handle to the original object.
4. Use the **MapViewOfFile** function in either process to gain access to the memory-mapped object.

CHAPTER 4

Accessing the Object Store, Database, and Registry

A Windows CE–based device has two types of memory: read-only memory (ROM) and random access memory (RAM). ROM typically stores the Windows CE operating system (OS) and any bundled applications that are included on your device. While some ROM-based applications must be loaded into RAM in order to work properly, many applications are designated as execute in place (XIP). While an XIP application can run from ROM, any data that the user creates with the application must be stored in RAM.

Using the File System

The Windows CE file system supports files that are stored in RAM, ROM, and installed file systems. Like Windows-based platforms, Windows CE employs handle-based file access. The **CreateFile** function returns a handle that references the created or opened file. The read, write, and information functions all use that handle to determine which file to act on. The read and write functions also use a file pointer to determine where in the file they read and write.

Windows CE uses a variety of techniques to simplify memory management and to reduce memory overhead. First, Windows CE does not use the current directory concept. Instead, all references to an object are given in the full path name. Further, Windows CE automatically compresses all files in the object store; therefore, no file has a flag to indicate compression. Of course, a flag does exist that distinguishes between a file in ROM and a file in RAM.

One of the sections of RAM on a Windows CE–based device is called the *object store.* The object store is where the directory, databases, registry, and file objects in volatile memory exist. The object store can be up to 16 megabytes (MB) in size. On Windows CE version 2.0 and earlier, a file can be up to 4 MB in size. Directories, databases, the registry, and files on Windows CE 2.10 and later are not limited to the 4-MB size restriction. Windows CE stores each database in the object store in a separate database volume. A database volume is a file that contains all of the data that is necessary for a database. Like the object store, a database volume can be up to 16 MB. All of the data in the object store is *transactioned,* which protects against data loss. If a Windows CE–based device loses power during a data transaction, Windows CE reverts all partial operations to the last known good state.

In addition to the object store, a user can install a file system such as the file allocation table (FAT) file system. An installed file system can provide access to a PC Card or to other external storage devices. You can divide an external storage device into multiple volumes, each of which is mounted separately. Each mounted volume is visible to the user as a folder in the root directory of the installed file system. Like the object store, a mounted volume is limited to 16 MB, and is transactioned to avoid data loss. While you can back up data to an external storage device, the working registry and RAM file system can only exist in the object store.

Note For programming purposes, Windows CE considers the object store as a special type of volume.

Using an Object Identifier

Windows CE assigns each object that is created within a volume a unique Windows CE object identifier. Each Windows CE object identifier is unique within a given volume, but not across multiple volumes. Windows CE also gives all volumes a Windows CE globally unique identifier (CEGUID). The object store also has a CEGUID, which is predefined. The most common use for a Windows CE object identifier is to access object data, such as a database record, and to obtain object data. Use the CEGUID in conjunction with the Windows CE object identifier to uniquely reference each record and object.

The following table shows where to obtain the Windows CE object identifier for the various types of objects in the object store.

Object type	Where to obtain the Windows CE object identifier
Directory or file	The *dwOID* member of the **WIN32_FIND_DATA** structure, which is returned by the **FindFirstFile** and **FindNextFile** functions. Also in the *dwOID* member of the **BY_HANDLE_FILE_INFORMATION** structure, which is returned by the **GetFileInformationByHandle** function.
Database	The return value of the **CeCreateDatabaseEx** or **CeFindNextDatabaseEx** function
Database record	The return value of the **CeSeekDatabase**, **CeReadRecordPropsEx**, or **CeWriteRecordProps** function
Mounted database volume	The **CeMountDBVol** and **CeEnumDBVolume** functions return the CEGUID of the mounted database volume.

Use the **CeOidGetInfoEx** function to return object data associated with the Windows CE object identifier. This function returns object data in a **CEOIDINFO** structure. The **CEOIDINFO** *wObjType* member contains a flag indicating the object type, such as OBJTYPE_DATABASE for a database object, and also identifies which object structure to use to access the data. **CEOIDINFO** also contains a member that returns data on either a file, directory, database, or database record. These values correspond to the type of object that is indicated by the *wObjType* member. For example, using **CeOidGetInfoEx** on a mounted database volume returns the database name, type identifier, number of records, database size, and sort order in a **CEDBASEINFO** structure, as well as the OBJTYPE_DATABASE value.

The following code example shows how to retrieve data from the object store or mounted volume by using the Windows CE object identifier.

```
void UsingCeOidGetInfo (void)
{
  CEOID CeOid;            // Object identifier
  PCEGUID pceguid;        // GUID of the mounted volume
  CEOIDINFO CeObjectInfo; // Structure into which to retrieve
                          // object data
  TCHAR szMsg[200];       // String to use with object data

  // Assign to pceguid the CEGUID of the mounted volume
  // on which the object resides.
  // ...
```

```
// Assign to CeOid the OID of the object about which
// you want to get data.
// ...

if (CeOidGetInfoEx (pceguid, CeOid, &CeObjectInfo))
{
  switch (CeObjectInfo.wObjType)
  {
    case OBJTYPE_FILE:
      wsprintf (szMsg, TEXT("Object is a file: %s"),
                CeObjectInfo.infFile.szFileName);
      break;

    case OBJTYPE_DIRECTORY:
      wsprintf (szMsg, TEXT("Object is a directory: %s"),
                CeObjectInfo.infDirectory.szDirName);
      break;

    case OBJTYPE_DATABASE:
      wsprintf (szMsg, TEXT("Object is a database: %s"),
                CeObjectInfo.infDatabase.szDbaseName);
      break;

    case OBJTYPE_RECORD:
      wsprintf (szMsg, TEXT("Object is a record"));
      break;

    case OBJTYPE_INVALID:
      wsprintf (szMsg,
                TEXT("The object store does not contain a valid ")
                TEXT("object that has this object identifier."));
      break;

    default:
      wsprintf (szMsg, TEXT("Object is of unknown OBJTYPE"));
      break;
  }
}
else
{
  // Your error-handling code goes here.

  if (GetLastError () == ERROR_INVALID_HANDLE)
    wsprintf (szMsg, TEXT("Invalid Object ID"));
  else
    wsprintf (szMsg, TEXT("Unknown error"));
}
} // End of UsingCeOidGetInfo code
```

Determining Available Disk Space

Memory is usually at a premium on a Windows CE–based device. Use the **GetDiskFreeSpaceEx** function to determine organizational information about a file system partition, including the number of bytes per sector, the number of sectors per cluster, the number of free clusters, and the total number of clusters. **GetDiskFreeSpaceEx** works on any file system, including installed file systems. You can also use the **GetStoreInformation** function to determine the current size of the object store, and how much free memory is available in the object store.

Creating and Opening a File or Directory

Call the **CreateFile** function to create a new file or to open an existing file. When you call **CreateFile**, Windows CE searches for a file in the directory that you specify. Depending on the parameters that you pass into **CreateFile**, Windows CE either opens the file, creates a new file, or truncates the old file. When you open a file, you set the read/write access for that file. Further, you also determine if Windows CE can share the file. If you choose not to share the file, Windows CE enables another call to **CreateFile** on the file only after your application closes the file. Once the file is opened or created, Windows CE assigns a unique file handle to the file. An application can use the file handle in functions that read from, write to, or describe the file. Note that files cannot be created with the overlapped attribute set.

Call the **CreateDirectory** function, with the full path of the new directory in the *lpPathName* parameter, to create a directory. The *lpSecurityAttributes* parameter is a holdover from Windows NT, and it should be set to NULL.

The following code example shows how to use **CreateFile** to open an existing file for reading.

```
void OpenFileExample (void)
{
  HANDLE hFile;

  hFile = CreateFile (TEXT("\MYFILE.TXT"),    // Open MYFILE.TXT
                      GENERIC_READ,           // Open for reading
                      FILE_SHARE_READ,        // Share for reading
                      NULL,                   // No security
                      OPEN_EXISTING,          // Existing file only
                      FILE_ATTRIBUTE_NORMAL,  // Normal file
                      NULL);                  // No template file
```

```
    if (hFile == INVALID_HANDLE_VALUE)
    {
      // Your error-handling code goes here.
      return;
    }
} // End of OpenFileExample code
```

Reading from and Writing to a File

Once you open or create a file with **CreateFile**, use the returned file handle to gain access to the file. Windows CE maintains a file pointer to read and write data between a file and a buffer. When you open a file for the first time, Windows CE places the file pointer at the beginning of the file. Windows CE advances the file pointer after reading or writing each byte between the buffer and the file.

Accessing the data buffer while Windows CE is performing a read or write operation with that buffer might lead to data corruption. Therefore, be sure that you do not modify a data buffer currently in use by a read or write operation. If you use multiple threads or semaphores, be sure that one thread does not access the data buffer while another thread performs a read/write operation.

Reading from a File

Use the **ReadFile** function to read data from a file. **ReadFile** uses the returned handle from **CreateFile** in the *hFile* parameter to identify the file from which to read. **ReadFile** begins reading at the location that is pointed to by the file pointer, and it continues reading up to the number of bytes that is specified in the *nNumberOfBytesToRead* parameter or to the end of the file. If **ReadFile** finds the end of the file, it does not return an error value. Instead, **Readfile** returns as much data as it reads up until the end of the file. Therefore, be sure to check the value returned in *lpNumberOfBytesRead* against the value passed to the function in *nNumberOfBytesToRead*. **ReadFile** returns the read data through the buffer that is pointed to in the *lpBuffer* parameter. During the copying process, **Readfile** does not perform any formatting or parsing; **ReadFile** reads the data exactly as the data exists in the file.

Note Windows CE does not support simultaneous read/write operations. Further, **ReadFile** does not support asynchronous read operations nor does it support read operations through a socket.

Writing to a File

Use the **WriteFile** function to place data into a file. Like **ReadFile**, **WriteFile** uses the handle that is returned from **CreateFile** in the *hFile* parameter to identify which file to write to. **WriteFile** then copies a specified number of bytes from the buffer that is pointed to by the *lpBuffer* parameter into the specified file. **CreateFile** begins placing the buffered data at the location within the file that is pointed to by the file pointer. Like **ReadFile**, **WriteFile** does not perform any formatting on the data; **WriteFile** writes the data exactly as the data exists in the buffer.

Note Windows CE does not support simultaneous read/write operations. Further, **WriteFile** does not support asynchronous write operations.

When writing to a file, use the **FlushFileBuffers** function to be sure that all data is properly written from the data buffer to the file. This situation is common when writing to a file on an installed file system, such as the FAT file system.

If you want to truncate the file when you close it, call the **SetEndOfFile** function. **SetEndOfFile** truncates the file at the current location of the pointer, and then it closes the file.

Setting the File Pointer in a File

Use the **SetFilePointer** function to move the file pointer a specified number of bytes. Like **ReadFile** and **WriteFile**, **SetFilePointer** uses the handle that is returned by **CreateFile** in the *hFile* parameter to identify the file in which to move the pointer. The number of bytes that are described by the *lDistanceToMove* and *lpDistanceToMoveHigh* parameters can be relative to the beginning of the file or relative to the current position of the pointer. Set which process you prefer by using the *dwMoveMethod* parameter. If you pass in a positive number of bytes, **SetFilePointer** moves the pointer toward the end of the file. If you pass in a negative number, **SetFilePointer** moves the file pointer toward the beginning of the file.

Read/Write Example

The following code example shows how to append one file to the end of another file. The example uses the **CreateFile** function to open two files: One.txt for reading and Two.txt for writing. Then the example uses **ReadFile** and **WriteFile** to append the contents of One.txt to the end of Two.txt by reading and writing 4-kilobyte (KB) blocks.

```
void AppendExample (void)
{
  HANDLE hFile, hAppend;
  DWORD dwBytesRead, dwBytesWritten, dwPos;
  char buff[4096];
  TCHAR szMsg[1000];

  // Open the existing file.

  hFile = CreateFile (TEXT("\ONE.TXT"),       // Open One.txt.
                      GENERIC_READ,           // Open for reading
                      0,                      // Do not share
                      NULL,                   // No security
                      OPEN_EXISTING,          // Existing file only
                      FILE_ATTRIBUTE_NORMAL,  // Normal file
                      NULL);                  // No template file

  if (hFile == INVALID_HANDLE_VALUE)
  {
    // Your error-handling code goes here.
    wsprintf (szMsg, TEXT("Could not open ONE.TXT"));
    return;
  }

  // Open the existing file, or if the file does not exist,
  // create a new file.

  hAppend = CreateFile (TEXT("\TWO.TXT"),       // Open Two.txt.
                        GENERIC_WRITE,          // Open for writing
                        0,                      // Do not share
                        NULL,                   // No security
                        OPEN_ALWAYS,            // Open or create
                        FILE_ATTRIBUTE_NORMAL,  // Normal file
                        NULL);                  // No template file

  if (hAppend == INVALID_HANDLE_VALUE)
  {
    wsprintf (szMsg, TEXT("Could not open TWO.TXT"));
    CloseHandle (hFile);            // Close the first file.
    return;
  }
```

```
    // Append the first file to the end of the second file.

    do
    {
      if (ReadFile (hFile, buff, 4096, &dwBytesRead, NULL))
      {
        dwPos = SetFilePointer (hAppend, 0, NULL, FILE_END);
        WriteFile (hAppend, buff, dwBytesRead,
                   &dwBytesWritten, NULL);
      }
    }
    while (dwBytesRead == 4096);

    // Close both files.

    CloseHandle (hFile);
    CloseHandle (hAppend);

    return;
} // End of AppendExample code
```

Reading and Writing File Attributes

Windows CE provides a variety of functions to identify and modify the status of a file. You can set the parameters of a file when you first create the file with **CreateFile**. After you create the file, you can view the file attributes with the **GetFileAttributes** function and you can modify the file attributes with the **SetFileAttributes** function. The following table describes what file attributes you can interact with by using the **GetFileAttributes**, **SetFileAttributes**, and **CreateFile** functions.

Flag	CreateFile	GetFile attributes	SetFile attributes
FILE_ATTRIBUTE_ARCHIVE	X	X	X
FILE_ATTRIBUTE_COMPRESSED		X	
FILE_ATTRIBUTE_DIRECTORY		X	
FILE_ATTRIBUTE_ENCRYPTED		X	
FILE_ATTRIBUTE_HIDDEN	X	X	X
FILE_ATTRIBUTE_INROM		X	
FILE_ATTRIBUTE_NORMAL	X	X	X
FILE_ATTRIBUTE_OFFLINE		X	X
FILE_ATTRIBUTE_READONLY	X	X	X

Flag	CreateFile	GetFile attributes	SetFile attributes
FILE_ATTRIBUTE_REPARSE_POINT		X	
FILE_ATTRIBUTE_ROMMODULE		X	
FILE_ATTRIBUTE_SPARSE_FILE		X	
FILE_ATTRIBUTE_SYSTEM	X	X	X
FILE_ATTRIBUTE_TEMPORARY	X	X	X
FILE_FLAG_WRITE_THROUGH	X		
FILE_FLAG_RANDOM_ACCESS	X		
FILE_FLAG_SEQUENTIAL_SCAN	X		
FILE_ATTRIBUTE_ROMSTATICREF		X	

You can also use the **GetFileSize** function to return the size of a file.

Memory Mapping a File

You can access a file though a memory object that is directly mapped to the file. Windows CE reflects any change to the memory-mapped object back to the file. Memory mapping can be more complicated to set up than traditional memory access; however, memory mapping simplifies file access. Instead of using a system-maintained pointer to write to the file, you can write directly to memory.

▶ **To set up and access a file by using memory mapping**

1. Call the **CreateFileForMapping** function to open or create the memory-mapped file.

   You can open any file, including files that are created by **CreateFile**, for memory mapping. Like **CreateFile**, you can also create a file with **CreateFileForMapping**. For Windows CE version 2.10 and earlier, you can use **CreateFileForMapping** for read-only permission.
2. Use the **CreateFileMapping** function to create an object in memory and to tie the object to the file that is opened by **CreateFileForMapping**.
3. Use the **MapViewOfFile** function to create a view of the memory-mapped object.
4. Use the pointer that is returned by **MapViewOfFile** to gain direct access to the memory-mapped object.
5. Call the **UnmapViewOfFile** function to unmap the object view.
6. Call the **CloseHandle** function to close the memory object.
7. Call **CloseHandle** to close the file.

Searching for a File or Directory

Use the **FindFirstFile**, **FindNextFile**, and **FindClose** functions to search for file or directory names that match a specified pattern. The pattern must be a valid file name and can include the asterisk (*) and question mark (?) wildcards.

▶ To find a single file or a series of files

1. Call the **FindFirstFile** function with the file name passed in through the *lpFileName* parameter.

   FindFirstFile returns a handle that you can use in **FindNextFile**. **FindFirstFile** also returns the file name, alternate file name, file size, CEOID, attributes, creation time, last access time, and last write time of the found file.

2. If necessary, call the **FindNextFile** function with the handle that was returned by **FindFirstFile**.

   FindNextFile returns the same data as **FindFirstFile**. You can call **FindNextFile** multiple times to find variations of the same file.

3. Modify the file, as necessary.
4. Destroy the handle that was created by **FindFirstFile** by using **FindClose**.

The following code example shows how to copy all text files to a new directory of read-only files named Textro. If necessary, the example changes all files in the new directory to read-only. The example uses **FindFirstFile** and **FindNextFile** to search the root directory for all .txt files. It then copies each .txt file to Textro. If a file is not read-only, the example changes to the Textro directory and uses **SetFileAttribute** to convert the copied file to read-only. After the example copies all .txt file, it uses **FindClose** to close the search handle.

```
void FindFileExample (void)
{
  WIN32_FIND_DATA FileData;   // Data structure describes the file found
  HANDLE hSearch;             // Search handle returned by FindFirstFile
  TCHAR szMsg[100];           // String to store the error message
  TCHAR szNewPath[MAX_PATH];  // Name and path of the file copied
  TCHAR szDirPath[] = TEXT("\TEXTRO");

  BOOL bFinished = FALSE;

  // Create a new directory.

  if (!CreateDirectory (szDirPath, NULL))
  {
    wsprintf (szMsg, TEXT("Unable to create new directory."));
    return;
  }
```

```
// Start searching for .txt files in the root directory.

hSearch = FindFirstFile (TEXT("\*.txt"), &FileData);
if (hSearch == INVALID_HANDLE_VALUE)
{
  wsprintf (szMsg, TEXT("No .TXT files found."));
  return;
}

// Copy each .txt file to the new directory and change it to
// read-only, if it is not already read-only.

while (!bFinished)
{
  lstrcpy (szNewPath, szDirPath);
  lstrcat (szNewPath, TEXT("\\"));
  lstrcat (szNewPath, FileData.cFileName);

  if (CopyFile (FileData.cFileName, szNewPath, FALSE))
  {
    if (!(FileData.dwFileAttributes & FILE_ATTRIBUTE_READONLY))
    {
      SetFileAttributes (szNewPath,
              FileData.dwFileAttributes | FILE_ATTRIBUTE_READONLY);
    }
  }
  else
  {
    wsprintf (szMsg, TEXT("Unable to copy file."));

    // Your error-handling code goes here.
  }

  if (!FindNextFile (hSearch, &FileData))
  {
    bFinished = TRUE;

    if (GetLastError () == ERROR_NO_MORE_FILES)
    {
      wsprintf (szMsg, TEXT("Found all of the files."));
    }
    else
    {
      wsprintf (szMsg, TEXT("Unable to find next file."));
    }
  }
}
```

```
    // Close the search handle.

    if (!FindClose (hSearch))
    {
      wsprintf (szMsg, TEXT("Unable to close search handle."));
    }
} // End of FindFileExample code
```

Moving and Copying Files and Directories

Use the **CopyFile** and **MoveFile** functions to copy and move files, respectively. Both functions receive the name of the file to copy or move in their respective *lpExistingFileName* parameters and copy or move the file to the location described in their respective *lpNewFileName* parameters. **CopyFile** contains the *bFailIfExists* parameter, which lets you determine if you want **CopyFile** to copy over a file if a version of the file exists in the target directory. **MoveFile** does not have a corresponding *pFailIfExists* parameter; instead, **MoveFile** automatically fails if the file already exists in the target directory. You can use both **CopyFile** and **MoveFile** on directories. Using **MoveFile** on a directory moves all of the files and subdirectories within that directory.

CopyFile and **MoveFile** can copy and move a file within the object store, from a mounted drive to the object store, or from the object store to a mounted drive. To copy or move a file or directory from one volume to another, place the file or directory in the object store as an intermediary step between the two volumes.

You can also use the **DeleteAndRenameFile** function to move a file from one directory to another. Like **MoveFile**, **DeleteAndRenameFile** deletes a file after moving that file to another directory. However, you have the option of renaming the new version of the file. This function works only on RAM-based files.

Manipulating File Times

Windows CE provides the **GetFileTime** function, which returns the file creation time, the last time that the file was accessed, and the last time that the file was written to. Windows CE creates file times in coordinated universal time (UTC) format, also known as Greenwich mean time. Use the **FileTimeToLocalFileTime** and **FileTimeToSystemTime** functions to translate the UTC time into the proper time zone format.

You can also set file times with the **SetFileTime** function. Depending on the file system, before using **SetFileTime**, you might need to gain access to the file by using **CreateFile** to open the file. If you change one of the dates on a file in the Windows CE store—for example, the file creation time—by default the other two dates—for example, the last time that the file was accessed and the last time that the file was written to—are updated to the new time as well. This technique works on the current object store, but does not work on a FAT file system.

Retrieving File and Directory Information

Windows CE provides three functions for retrieving data about a given file or directory. The following table describes these functions.

Function	Returns
GetFileAttributes	The flag settings for the file attributes, including archive, compression, encryption, and read/write status
GetFileSize	The file size
GetFileInformationByHandle	The same information as **GetFileTime**, **GetFileAttributes**, and **GetFileSize**, as well as the object identifier of the file

Deleting a File or Directory

Once you are finished with a file, close the file handle by using the **CloseHandle** function. **CloseHandle** is a generic function that closes a variety of handles, including the handle to a file. If you leave a file open when your application terminates, Windows CE automatically closes the file.

▶ **To close and delete a file**

1. Close the file with a call to **CloseHandle**.
2. Delete the file with a call to the **DeleteFile** function.

   Windows CE cannot delete a file that has an open handle.

Call the **RemoveDirectory** function, with the full path name of the directory in the *lpPathName* parameter, to delete a directory. You must delete all of the files in the directory before calling **RemoveDirectory**.

Accessing Data on Other Storage Media

A Windows CE–based device has other areas to store applications besides the object store. You can access a file stored in ROM just like any other file: by calling the file application programming interface (API). However, you cannot alter a file that is stored in ROM. Instead, ROM-based files are marked with the FILE_ATTRIBUTE_INROM value, indicating that they are read-only. Windows CE also uses the FILE_ATTRIBUTE_ROMMODULE value to indicate that a file is designated to be executed in ROM, rather than copied to RAM. You cannot use **CreateFile** to open files that are designated with FILE_ATTRIBUTE_ROMMODULE. Instead, use the **LoadLibrary** and **CreateProcess** functions to gain access to the module.

Also, Windows CE supports PC Cards, such as Advanced Technology Attachment (ATA) flash cards, and linear flash cards. These cards can have an installed file system that can utilize the storage space for files and databases. However, a mounted file system must be used in conjunction with an installed file system driver, such as FAT. Once you install a PC Card, you can copy database objects between the object store and the mounted volume.

Windows CE does not assign a letter label to a storage card in the same manner that a desktop computer assigns a drive letter to a hard disk. Instead, the file driver creates directories in the root directory representing each partition on each storage card. In Windows CE 2.0 and earlier, these directories were given default names, such as Storage Card or PC Card. In Windows CE 2.10, the FAT file system driver queries the PC Card driver for a default name. If the PC Card driver does not supply a default name, Windows CE uses Storage Card. You can also tell the difference between a object store directory and a mounted volume directory by the file attributes. All directories on a mounted file system have the FILE_ATTRIBUTE_TEMPORARY file attribute flag set.

The following code example tries to open any Storage Card directories that exist and tests them to see if they are located on a storage card.

```
void FindingStorageCards (void)
{
  TCHAR szMsg[100];       // String to store the error message
  HANDLE hSearch;         // Search handle returned by FindFirstFile
  WIN32_FIND_DATA  fd;    // Data structure describes the file found
  BOOL bFinished = FALSE; // Flag to indicate whether the loop is done
  TCHAR *szFname = TEXT("\\Storage Card*");
                          // Name that matches all storage cards

  // Be sure Storage Card exists.
  hSearch = FindFirstFile(szFname, &fd);

  if (hSearch == INVALID_HANDLE_VALUE)
  {
    wsprintf(szMsg, TEXT("No storage card found."));
    return;
  }

  do {
    // Test whether the file is really on a storage card, and
    // not just a directory in the root directory.
    // It must have both the directory attribute and
    // the temporary attribute.
```

```
        if (  (fd.dwFileAttributes & FILE_ATTRIBUTE_TEMPORARY)
           && (fd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) )
        {
          wsprintf (szMsg,
                    TEXT("%s is a storage card."), fd.cFileName);
        }
        else
        {
          wsprintf (szMsg,
                    TEXT("%s is not a storage card."), fd.cFileName);
        }

        if (!FindNextFile (hSearch, &fd))
        {
          bFinished = TRUE;
          if (GetLastError () != ERROR_NO_MORE_FILES)
          {
            wsprintf (szMsg,
                      TEXT("Error trying to find files matching \"%s\"."),
                      szFname);
          }
        }
      }
      while (!bFinished);

      FindClose (hSearch);  // Close the search handle.

    } // End of FindingStorageCards example code
```

Using a Windows CE Database

Windows CE provides a simple database API containing a single level and a maximum of four sort indices. Prior to Windows CE 2.10, a Windows CE database could exist only in the object store. Windows CE 2.10 and later enables you to store and access a database in a database volume, anywhere on the device. This includes PC Cards or other installed file systems. The file is called a volume because it can contain more than one database. Use the database API to create and store a simple database, such as a phone number listing or an e-mail repository.

A Windows CE database consists of one or more records. The maximum size of a record is defined in the CEDB_MAXRECORDSIZE constant in Windbase.h; for Windows CE 2.10, CEDB_MAXRECORDSIZE is 131,072 bytes. A record can have a variable number of properties, but it cannot contain another record. The maximum size of a property is defined in the CEDB_MAXPROPDATASIZE constant, which is 65,471 bytes. Windows CE allocates space for a record or property only when necessary. These same constants are defined in Rapi.h for remote API (RAPI) calls. The following table shows the different types of available record properties.

Record property type	Contains
CEVT_BOOL	Boolean value
CEVT_CEBLOB	Binary object
CEVT_R8	8-byte signed value
CEVT_FILETIME	Time and date data
CEVT_I2	2-byte signed integer
CEVT_I4	4-byte signed integer
CEVT_LPWSTR	Long pointer to a Unicode string
CEVT_UI2	2-byte unsigned integer
CEVT_UI4	4-byte unsigned integer

Note The BOOL and Double properties are available only on Windows CE version 2.10 and later.

The overhead associated with creating a record is 20 bytes per record. The overhead for a property is 4 bytes per property.

In addition to records, a database contains a name and type identifier. The database name is a null-terminated string of up to 32 characters. The type identifier is application-specific and is commonly used to identify similar databases. For example, the Microsoft Pocket suite uses the number 24 as a type identifier for all Contacts databases.

Because Windows CE is designed to operate in a relatively volatile environment, the Windows CE database does not update automatically only when the database opens or closes. Instead, the database updates after each individual transaction, such as a **CeWriteRecordProps** call.

Mounting and Unmounting a Database Volume

In order to create and use a database volume with Windows CE 2.10 and later, you must first mount, or open, the volume. This does not include the object store, which is always mounted and identified by a system globally unique identifier (GUID). Once all operations are complete, you must unmount, or close, the volume. Mounting and unmounting adds a level of complexity to your database programming but allows you to store a database elsewhere on a device besides the object store. Note that you can still use the original database API in Windows CE 2.10 and later to gain access to a database in the object store. However, the earlier API exists for backward compatibility only; all new applications should use the new API.

Call the **CeMountDBVol** function to mount a volume that can store Windows CE databases on any file system. This file system can include the object store or an installed file system, such as the FAT file system on a PC Card. **CEMountDBVol** acts as a specialized version of **CreateFile**; instead of opening or creating a generic file, **CEMountDBVol** opens or creates a database volume. **CEMountDBVol** accepts the location of the file that is the database volume and returns a **PCEGUID**. Functions designed to manipulate mounted database volumes use the **PCEDUID** as a handle to identify a given volume.

Call the **CEUnmountDBVol** function to unmount a database volume. Like **CloseHandle**, **CeUnmountDBVol** returns resources that the database volume was using back to the system. To be sure that Windows CE writes any cached data to permanent storage, call the **CeFlushDBVol** function prior to unmounting the volume. The database can also be closed by user reset or a power failure. Because Windows CE automatically flushes the cache on a periodic basis, you will lose only data altered since the last cache flush. Also, the volume itself is not altered by a power failure or reset.

Note that more than one process can mount the same database volume. Windows CE keeps track of how many processes have access to a given database volume. When the last process unmounts the database, Windows CE closes the file.

The following code example shows how to unmount a database volume. For an example showing how to mount a database volume, see "Mounted Database Example."

```
void UnmountingDBs (PCEGUID pceguid)
{
  TCHAR szMsg[100];

  if (!CeUnmountDBVol (pceguid))
  {
    wsprintf (szMsg, TEXT("Failed unmounting the database."));

    // Your error-handling code goes here.
  }
} // End of UnmountingDBs example code
```

Creating a Database

Use the **CeCreateDatabaseEx** function to create a database in any volume. **CeCreateDatabaseEx** identifies the volume, names the database, identifies the sort order, and lets you pass in a user-defined type identifier. The sort order is an index that is applied to a database to manipulate the record ordering. Although you define the sort order when the database is created, you can alter the sort order later. The type identifier, while user-defined, is commonly used to identify similar types of databases.

▶ **To create a database within the object store with Windows CE 2.10 and later**

1. Call the **CREATE_SYSTEMGUID** macro to create an object identifier for the object store.

   The returned GUID can act as a standard identifier for a handle.

2. Call **CeCreateDatabaseEx**, using the CEGUID created in the previous call to **CREATE_SYSTEMGUID**.

When you create a database on a mounted volume, it is usually a good idea to make the database uncompressed. Accessing a mounted volume is usually slower than accessing the object store: compressing and decompressing slows the process even further. Therefore, create your database as an uncompressed database, unless you expect the database to be larger than the storage card can contain.

▶ **To set compression on a database after the database is created**

1. Create a **CEDBASEINFO** structure with the CEDB_VALIDBFLAGS value set in the *dwFlags* value.
2. Toggle off the CEDB_NOCOMPRESS bit in the *dwFlags* value by using the following code fragment.

```
dwFlags &= ~(CEDB_NOCOMPRESS);
```

3. Call the **CeSetDatabaseInfoEx** function with the previous **CEDBASEINFO** structure.

Opening a Database

Once you create a database, use the **CeOpenDatabaseEx** function to open it. **CeOpenDatabaseEx** specifies the sort order you use on the database during a given session in the *propid* parameter. To use a different sort order, you must close the database and open it up again with a different sort order. You can also pass a handle to a window in the *hwndNotify* member of the **CENOTIFYREQUEST** structure. Windows CE sends messages to the specified window when other processes or other threads modify the open database. When a change occurs, Windows CE sends a WM_DBNOTIFICATION message with the *lParam* set to **CENOTIFICATION**. The following table describes the possible values of the *uType* parameter in the **CENOTIFICATION** structure.

Message	Description
DB_CEIOD_CHANGED	Record changed
DB_CEIOD_CREATED	Record created
DB_CEIOD_RECORD_DELETED	Record deleted

Remember that you must mount the volume with **CEMountDBVol** before opening the database. In addition, **CeOpenDatabaseEx** lets you receive additional data regarding other processes and threads through **CENOTIFYREQUEST**. **CENOTIFYREQUEST** lets you choose either the original Windows CE messages for database changes or the WM_DBNOTIFICATION message. The *lParam* of WM_DBNOTIFICATION points to a **CENOTIFICATION** structure.

The following table describes the different parameters of the **CENOTIFICATION** structure.

Parameter	Description
dwSize	**CENOTIFICATION** size
dwParam	Application-defined value
uType	Why the message was sent
guid	GUID of the relevant database volume
oid	Object identifier of the relevant database record
oidParent	Object identifier of the parent of the database record

Windows CE places **CENOTIFICATION** in a heap that you define in **CENOTIFYREQUEST**. If you do not specify a heap in **CENOTIFYREQUEST**, Windows CE creates the heap in your default process heap. Once finished, free memory is allocated to **CENOTIFICATION** with a call to the **CeFreeNotification** function. You must delete the **CENOTIFICATION** structure each time you receive a WM_DBNOTIFICATION message.

The following code example attempts to open a database of addressees by calling **CeOpenDatabaseEx**. If the database does not exist, the code example calls **CeCreateDatabaseEx** to create a new address database with three different sort orders. After creating the database, the example attempts to open the database again.

```
HANDLE OpenDatabase (
      HWND hwndNotify,  // Handle to the window to which
                        // notification messages are posted
      PCEGUID pceguid,  // Pointer to the mounted database
                        // volume where the database to
                        // be opened resides
      CEOID CeOid)      // Object identifier of the database
                        // to be opened
{
  int index;
  DWORD dwError;              // Return value from GetLastError
  HANDLE hDataBase;           // Open handle to the address database
  CENOTIFYREQUEST *pRequest;  // CENOTIFYREQUEST structure
  CEDBASEINFO CEDBInfo;       // Structure containing the
                              // database data
  TCHAR szError[100];         // String to use with error messages

  // Allocate memory for pRequest.
  pRequest = (CENOTIFYREQUEST *) LocalAlloc (LPTR,
                                    sizeof (CENOTIFYREQUEST));
```

```
pRequest->dwSize = sizeof (CENOTIFYREQUEST);
pRequest->hwnd = hwndNotify;
pRequest->hHeap = NULL; // Let system allocate memory properly.
pRequest->dwFlags = 0;  // Notifications are handled as they
                        // were in Windows CE version 1.0.

hDataBase = CeOpenDatabaseEx (
        pceguid,            // Pointer to the mounted volume
        &CeOid,             // Location for the database identifier
        TEXT("MyDBase"),    // Database name
        0,                  // Sort order; 0 indicates to ignore
        CEDB_AUTOINCREMENT, // Automatically increase seek pointer
        pRequest);          // Pointer to a CENOTIFYREQUEST
                            // structure

if (hDataBase == INVALID_HANDLE_VALUE)
{
  dwError = GetLastError ();

  if (dwError == ERROR_NOT_ENOUGH_MEMORY)
  {
    wsprintf (szError, TEXT("Not enough memory"));
  }
  else
  {
    // Possibility of nonexisting database; create it.

    // Initialize structure CEDBInfo
    memset (&CEDBInfo, 0, sizeof(CEDBInfo));

    // Create the database with the following specified flags.
    // Create the database as uncompressed.
    // You can use CeSetDataBaseInfoEx to compress the database.
    CEDBInfo.dwFlags =
        CEDB_VALIDNAME        // szDbaseName is valid
        | CEDB_VALIDTYPE      // dwDbaseType is valid
        | CEDB_VALIDDBFLAGS   // HIWORD of dwFlag is valid
        | CEDB_VALIDSORTSPEC  // rgSortSpecs is valid
        | CEDB_NOCOMPRESS;    // The database is not compressed.

    // Assign the database name as MyDBase.
    wcscpy (CEDBInfo.szDbaseName, TEXT("MyDBase"));

    // Assign the database type.
    CEDBInfo.dwDbaseType = 0;
```

```
        // Set the number of active sort orders to 4.
        // This is the maximum number allowed.
        CEDBInfo.wNumSortOrder = 4;

        // Initialize the array of sort order descriptions.
        for (index = 0; index < CEDBInfo.wNumSortOrder; ++index)
        {
          // Sort in descending order.
          CEDBInfo.rgSortSpecs[index].dwFlags = CEDB_SORT_DESCENDING;

          // Assign the identifier of the properties to sort by.
          // CEDBInfo.rgSortSpecs[index].propid = ...;
        }

        // Create database "MyDBase".
        CeOid = CeCreateDatabaseEx (pceguid, &CEDBInfo);

        if (CeOid == NULL)
        {
          wsprintf (szError,
                    TEXT("ERROR: CeCreateDatabaseEx failed (%ld)"),
                    GetLastError ());
        }
        else  // Succeeded in creating the database; open it.
        {
          hDataBase = CeOpenDatabaseEx (pceguid, &CeOid,
                            TEXT("MyDBase"), 0, 0, pRequest);
        }
      }
    }

    // Return the database handle.
    return hDataBase;

} // End of OpenDatabase example code

/*
```

Modifying the Sort Order

When you are creating a database, you can define up to four different sort orders. Typically, each record in a database contains a similar set of properties, and each type of property shares the same property identifier. For example, each record in a Contacts database might contain a name, street address, city, state or province, postal code, and telephone number. All name properties would have the same property identifier, all street addresses would have the same property identifier, and so on. You can select one of these properties and direct the system to sort the records based on that property. However, you cannot perform a sort on a binary property. The order in which the records are sorted affects the order in which the **CeSeekDatabase** database-seeking function finds records in the database.

You specify which of the four sort orders you want to use on a database when you call the **CeOpenDatabase** or **CeOpenDatabaseEx** function with the **SORTORDERSPEC** structure. **SORTORDERSPEC** contains the identifier of a single property on which the database properties are to be sorted. **SORTORDERSPEC** also includes a combination of flags that indicate whether to sort the records in ascending or descending order, whether the sort is case-sensitive, and whether to place records that do not contain the specified property before or after all other records. By default, sorting is done in ascending order and is case-sensitive. Windows CE places all records not containing the specified property at the end of all other records. Although you can have only one sort order active for each handle, you can open multiple handles to a given database. Using multiple handles, you can use more than one sort order.

You can change these four sort orders or other database properties with a call to the **CeSetDatabaseInfoEx** function. **CeSetDatabaseInfoEx** lets you change the database name, type, or any of the four sort orders. When you change a sort order, Windows CE parses the database and changes each record. This process might take several minutes on a large database. It is more efficient to create the database with the necessary sort orders to begin with. If you must change the sort order, be sure that you inform the user of the projected time delay.

Searching for a Record

Before you can read and write a record in a database, you must find the record. Use **CeSeekDatabase** to move the database seek pointer to the record that you want to read from or write to. **CeSeekDatabase** always uses the current sort order as it is specified in the call to **CeOpenDatabaseEx**. **CeSeekDatabase** can search through a database for a relative or exact property value, a relative or exact location, or a record object identification. However, Windows CE can perform a seek operation only on a sorted property value. When **CeSeekDatabase** finds a record, Windows CE positions the seek pointer at that record. Any subsequent read operation takes place at the location of the seek pointer. If you set the CEDB_AUTOINCREMENT flag in your call to **CeOpenDatabaseEx**, each read operation automatically increments the seek pointer from the current record to the next record.

Seek operations are affected by the sort order that is associated with the open database handle. For example, suppose that the Contacts database was opened by using a sort operation on the name property. If you specify the CEDB_SEEK_VALUEFIRSTEQUAL flag and a value of "John Smith," **CeSeekDatabase** searches from the beginning of the database looking only at the name property of each record. It stops when, and if, it finds a matching property.

The following code example shows how to search for a record by using **CeSeekDatabase**.

```
void UsingCeSeekDatabase (void)
{
  CEOID CeOid;          // Object identifier
  CEOID CeOidSeek;      // Object identifier
  DWORD dwIndex;        // Index of record to seek
  TCHAR szError[100];   // String for displaying error messages
  HANDLE hDataBase;     // Handle to a user-allocated heap

  // Assign to CeOidSeek the object identifier of the record
  // being searched for.
  // ...

  // Call CeOpenDatabaseEx to open the database.
  // hDataBase = CeOpenDatabaseEx {...}
```

```
    // Perform the seek. This type of seek operation is very efficient.
    if ((CeOid = CeSeekDatabase(
                hDataBase,          // Handle of the database
                CEDB_SEEK_CEOID,    // Finding an object with the
                                    // same identifier
                CeOidSeek,          // Specifies the record to seek
                &dwIndex            // If successful, moves the database
                                    // pointer to point to the record
                  )) == 0)
    {
      wsprintf (szError, TEXT("ERROR: CeSeekDatabase failed (%ld)"),
                            GetLastError ());
    }

} // End of UsingCeSeekDatabase example code
```

Reading a Record

Once you find a record, use the **CeReadRecordProps** or **CeReadRecordPropsEx** functions to read the properties of the record. To indicate the properties to be read, specify an array of property identifiers. Also, specify the buffer to which the functions write the property data and the size of the buffer. Setting the CEDB_ALLOWREALLOC flag in the *dwFlags* parameter instructs Windows CE to reallocate the buffer if the returned data is too large. If the database is stored in a compressed format, the database engine must decompress records in 4-KB sections as they are read.

CeReadRecordProps can return selected record properties or a full view of all the properties. If you choose to view all the properties, you can have **CeReadRecordProps** allocate memory from the local heap to return all the record values. **CeReadRecordPropsEx** can use any heap to return record values, not just the local heap. For efficiency, your application should read all of the desired properties in a single call rather than in several separate calls.

If you do read for a specific property, make sure to check for the CEDB_PROPNOTFOUND flag in the **CEPROPVAL** structure for that property. You should check for this flag because the record in question might not have the property you are looking for.

When Windows CE reads a record property successfully, the system copies the property information into the specified buffer as an array of **CEPROPVAL** structures. **CeReadRecordProps** and **CeReadRecordPropsEx** also return the Windows CE object identifier of the record. All the variable-size data, such as strings and binary large objects (BLOBs), are copied to the end of the buffer. The **CEPROPVAL** structures contain pointers to this data.

The following code example shows how to create and write four properties: a 16-bit signed integer, a 32-bit signed integer, a null-terminated string, and a BLOB to the new record.

```
void ReadingDBRecords (void)
{
  CEOID CeOid;              // Object identifier of the record read
  PCEGUID pceguid;          // Pointer to the mounted database volume
  WORD wcPropID;            // Number of properties retrieved
  DWORD dwcbBuffer,         // Count of bytes of the *lpBuffer
        dwError;            // Return value of the GetLastError function
  TCHAR szMsg[100];         // String for displaying the error message
  LPBYTE lpBuffer = NULL; // Pointer to a buffer that receives record
                            // property data
  HANDLE hDataBase,         // Handle to a database to be opened
         hHeap;             // Handle to the heap for allocating records

  // Assign to pceguid the GUID of the mounted volume
  // where the database resides.
  // ...

  // Create the heap by calling HeapCreate to allocate the database
  // records.
  // ...

  // Open the database with the current seek position to be
  // automatically incremented with each call.

  hDataBase = CeOpenDatabaseEx (
          pceguid,              // Pointer to the mounted volume
          &CeOid,               // Location of the database identifier
          TEXT("MyDBase"),      // Database name
          0,                    // Sort order; 0 indicates to ignore.
          CEDB_AUTOINCREMENT, // Automatically increase seek pointer
          NULL);                // Does not need to receive notification

  // Check for errors on the hDataBase handle before continuing.

  if (hDataBase == INVALID_HANDLE_VALUE)
  {
    // Your error handling code goes here.
    return;
  }
```

```
  while ((CeOid = CeReadRecordPropsEx (
            hDataBase,          // Handle of the database.
            CEDB_ALLOWREALLOC,  // Use LocalAlloc to get the buffer.
            &wcPropID,          // Number of properties retrieved
            NULL,               // NULL means retrieve all properties.
            &lpBuffer,          // Buffer receives property data.
            &dwcbBuffer,        // Count of bytes in *lpBuffer.
            hHeap)) != 0)       // Handle to the heap for allocating
                                // the record when
                                // CEDB_ALLOWREALLOC is specified.
  {
    // The record is now available in the lpBuffer. Add code here to
    // manipulate the properties in this record.
    // ...
  }

  // Error handling if CeReadRecordPropsEx fails

  dwError = GetLastError ();

  if (dwError == ERROR_NO_MORE_ITEMS)
  {
    wsprintf (szMsg,
              TEXT("Read through all records in the database."));
  }
  else if (dwError == ERROR_INSUFFICIENT_BUFFER)
  {
    wsprintf (szMsg,
              TEXT("Re-allocation of database records failed."));
  }
  else
  {
    wsprintf (szMsg, TEXT("Other errors."));
  }

  // Close the database handle.

  CloseHandle (hDataBase);

} // End of ReadingDBRecords example code
```

Writing and Creating a Record

Use the **CeWriteRecordProps** function to write to a record. Like **CeReadRecordProps**, **CeWriteRecordProps** uses an array of CEPROPVAL structures to pass property information into the record. To create a new record, call **CeWriteRecordProps** with the *oidRecord* parameter set to 0. Once you finish writing or creating a record, Windows CE updates the database.

When writing to a mounted volume, Windows CE caches all write operations. The database subsystem periodically requests a cache flush after a series of operations. If memory is low, Windows CE flushes the cache to permanent storage. Windows CE cannot choose to flush only part of the cache: all database blocks are flushed during a flush.

Deleting Database Information

Windows CE enables you to delete properties, records, or entire databases. To delete a property, call **CeWriteRecordProps** with the CEDB_PROPDELETE flag set in the *wFlags* parameter and a **CEPROPVAL** structure describing the property to delete in the *rgPropVal* parameter. You can delete a property with **CeWriteRecordProps** in any volume.

Note A record that has been deleted and then restored receives a new Windows CE object identifier.

Call the **CeDeleteRecord** function to delete a single record in a database. Like **CeWriteRecordProps**, **CeDeleteRecord** works on any record, regardless of the location of the record. Call the **CeDeleteDatabaseEx** function to delete a database in a volume. Be sure that the database is not open before you attempt to delete the database.

Enumerating a Database and Database Volumes

Enumerating databases is the process of sequentially accessing each database in a group. The group can include all databases in the object store, databases of a specified type, or databases in all mounted volumes. Use enumeration when you need to change all databases of a certain type or when you need to synchronize data between a desktop computer and a Windows CE–based device.

Use the **CeFindFirstDatabaseEx** and **CeFindNextDatabaseEx** functions to enumerate databases within a specified database volume. The volume can be the object store or any mounted database volume. The primary difference between these two sets of functions is that **CeFindFirstDatabaseEx** and **CeFindNextDatabaseEx** have an additional parameter identifying the CEGUID of the volume. If you pass in an invalid CEGUID, Windows CE searches all of the volumes on a Windows CE–based device, including the object store. You can also enumerate the mounted database volumes with a call to **CeEnumDBVol**.

When you are finished, call **CloseHandle** to close the enumeration handle.

The following code example shows how to enumerate the databases within the object store.

```
void EnumeratingDBs (void)
{
  DWORD dwError;          // Return value of GetLastError function.
  TCHAR szMsg[100];       // String to display error message
                          // and database data.
  HANDLE hEnumDB;         // Handle to a database enumerator.
  CEOID CeOid;            // Object identifier of a database.
  CEOIDINFO CeObjectInfo; // Structure that contains
                          // database data.
  PCEGUID pceguid = NULL; // Pointer to the mounted volume identifier.
                          // NULL means all mounted database volumes
                          // are to be searched.

  // Find the first database. Set the database type to 0, so all types
  // of databases are enumerated. If pceguid is set to NULL or an
  // invalid GUID is created by CREATE_INVALIDGUID, all mounted
  // database volumes are searched.

  hEnumDB = CeFindFirstDatabaseEx (pceguid, 0);

  if (hEnumDB == INVALID_HANDLE_VALUE)
  {
    if (GetLastError () == ERROR_OUTOFMEMORY)
      wsprintf (szMsg, TEXT("Out of memory."));
    else
      wsprintf (szMsg, TEXT("Unknown error."));

    return;
  }
```

```
  while ((CeOid = CeFindNextDatabaseEx (hEnumDB, pceguid)) != 0)
  {
    // Retrieve database data.
    if (!CeOidGetInfoEx (pceguid, CeOid, &CeObjectInfo))
    {
      // Your error-handling code goes here.

      CloseHandle (hEnumDB);  // Close the search handle.

      return;
    }
    else
    {
      wsprintf (szMsg, TEXT("The name of the database is: %s"),
                  CeObjectInfo.infDatabase.szDbaseName);
    }
  }

  // Error handling if CeFindNextDatabaseEx fails
  dwError = GetLastError ();

  if (dwError == ERROR_KEY_DELETED)
  {
    // A database has been deleted during enumeration. You must
    // restart the enumeration process.
    wsprintf (szMsg,
              TEXT("A database was deleted during enumeration."));
  }
  else if (dwError == ERROR_NO_MORE_ITEMS)
  {
    wsprintf (szMsg, TEXT("No more item to enumerates."));
  }
  else
  {
    wsprintf (szMsg, TEXT("Unknown error."));
  }

  // Close the search handle.
  CloseHandle (hEnumDB);

} // End of EnumeratingDBs example code
```

Mounted Database Example

The following code example shows how to mount the MyDBVol database volume. The example opens the database volume if the volume already exists, or it creates the database if the database does not exist. The example also shows how to enumerate all of the mounted database volumes and transfer the data into permanent storage.

```
void MountingDBVolume (void)
{
  PCEGUID pceguid;            // Pointer to the mounted volume
  LPWSTR lpBuf;               // Buffer to store the volume name
  TCHAR szError[100];         // String to display with error message
  DWORD dwError,              // Return value of GetLastError function
        dwNumChars;           // Length of the buffer in characters
  TCHAR * szFname = TEXT("\\Storage Card\\MyDBVol");
                              // Name of the database volume
  BOOLEAN bFinished = FALSE;  // Loop (enumeration) control

  // Allocate memory for pceguid.
  pceguid = (PCEGUID) LocalAlloc (LPTR, sizeof (CEGUID));

  // Open the database volume MyDBVol on the storage card if it exists.
  // Create it if it does not exist.
  if (!CeMountDBVol (pceguid,         // Pointer to a CEGUID.
                     szFname,         // Database volume name.
                     OPEN_ALWAYS))    // Create the database volume
                                      // if it does not exist.
  {
    // Your error-handling code goes here.

    // If the database volume was not opened or created
    wsprintf (szError, TEXT("ERROR: CeMountDBVol failed (%ld)"),
                       GetLastError());
  }

  // Allocate memory for lpBuf.
  lpBuf = (LPWSTR) LocalAlloc (LPTR, MAX_PATH);

  // Assign MAX_PATH to dwNumChars.
  dwNumChars = MAX_PATH;

  // Create an invalid pceguid as the input parameter of
  // CeEnumDBVolumes.
  CREATE_INVALIDGUID (pceguid);
```

```
  while (!bFinished)
  {
    // Enumerates database volumes
    if (!CeEnumDBVolumes (pceguid,      // Points to a mounted volume
                          lpBuf,        // Buffer to store the
                                        // volume name
                          dwNumChars))  // Length of buffer in char

    {
      // If error occurs in enumerating

      bFinished = TRUE;     // Finished enumerating

      dwError = GetLastError();

      wsprintf (szError, TEXT("ERROR: CeMountDBVol failed (%ld)"),
                          dwError);

      if (dwError == ERROR_INSUFFICIENT_BUFFER)
      {
        // To avoid having to restart the enumeration,
        // free lpBuf and reallocate a bigger buffer.

        bFinished = FALSE;  // Continue with enumerating
        LocalFree (lpBuf);  // Free lpBuf

        // Your code to reallocate a bigger buffer goes here.
        // ...
      }
    }
    else  // Succeeded in enumerating
    {
      // Flush the database volume's data to permanent storage.

      if (!CeFlushDBVol (pceguid))
      {
        wsprintf (szError, TEXT("ERROR: CeFlushDBVol failed."));

        // Your error-handling code goes here.
      }
    }
  }
} // End of MountingDBVolume example code
```

Manipulating the Registry

The registry is a database that stores data about applications, drivers, user preferences, and other data that Windows CE needs to perform properly. For example, a user's default preferences for Pocket Word are stored in the registry. The registry is organized in a hierarchical system of *keys* and *values*. A key is similar to a directory, and can contain values and other keys. Windows CE supports three root keys. The following table shows the three root keys that Windows CE supports, and what type of data you should store under these keys.

Key name	Contains
HKEY_LOCAL_MACHINE	Hardware and driver configuration data
HKEY_CURRENT_USER	User configuration data
HKEY_CLASSES_ROOT	OLE and file type matching configuration data

A value is the basic piece of data that is stored in the registry, and it can be a variety of types, including string or binary. Each value has a name and an associated piece of data. For example, a device running the Windows CE, Handheld PC Professional Edition, software uses **Wrap to Window** in the **HKEY_LOCAL_MACHINE\Software\Microsoft\Pocket Word\Settings** key to store a numerical value.

The following table describes the different limits in the registry.

Limit	Description
Key or value name length	255 characters
Data size	4 KB
Key hierarchy depth	Up to 16 nested subkeys

Use the registry to store data that your application needs for each session. For example, you can save the state of your application during the shutdown process. Your application can search the registry on startup to reinstate the previously saved settings. When programming with the registry, try to keep name and data size small. Note that registry values in Windows CE take up less memory than registry keys. Design your registry hierarchy to use as few keys as possible.

Note Windows CE implements the registry as a RAM-based heap file. If the RAM loses power, the registry data may be lost if the OEM has not implemented a registry backup procedure. Windows CE must then reload the initial registry from ROM.

Creating and Opening a Registry Key

Create a registry key with a call to the **RegCreateKeyEx** function. Windows CE automatically opens the key once you create it. If the key already exists, a call to **RegCreateKeyEx** simply opens the key for processing. Also, you can use the **RegOpenKeyEx** function to open a key. The difference between the two functions is that **RegOpenKeyEx** does not create a new key if the key did not previously exist.

Reading a Registry Key or Value

Use the **RegQueryValueEx** function to read a specified value from the registry. You can also use **RegQueryInfoKey** to receive data about a specified key.

Writing and Creating a Registry Value

Use the **RegSetValueEx** function to add a value to or alter a value in a registry key. If the value that you want to change does not exist, **RegSetValueEx** creates a value and the associated data. You can also choose to name the type of data as you enter the data into the key. Because all values are stored and returned in a binary format, labeling data does not affect how Windows CE views it. However, for the sake of a third-party registry editor, you should attempt to label your values appropriately.

Enumerating Registry Keys

Use the **RegEnumKeyEx** function to enumerate the subkeys of a specified key. Call the **RegEnumValue** function to enumerate the values in a specified key. Both functions receive an index number that moves the function to the next key or value. When there are no more keys or values to return, both functions return the ERROR_NO_MORE_ITEMS value.

Deleting a Registry Key or Value

Call the **RegDeleteKey** or **RegDeleteValue** function to delete registry keys and values, respectively. The key must be closed before you can properly delete it.

Closing the Registry

Once you are done with a registry key, close the key with a call to the **RegCloseKey** function. Calling **RegCloseKey** instructs Windows CE to flush any unwritten data to the key before closing the key.

Flushing the Registry

One option that might be available is the **RegFlushKey** function. **RegFlushKey** is an OEM-implemented function that attempts to flush the entire registry to a platform-supported storage. If it is available, you should call **RegFlushKey** after each major change or group of changes to your registry. Depending on the implementation, **RegFlushKey** can require a great deal of system resources. So while **RegFlushKey** may be an effective way to back up the registry, calling the function multiple times for minor changes might impede the performance of your application.

CHAPTER 5

Integrating Engines into an Application

An *engine* is a section of an application that determines how that application manages and manipulates a type of data. From a developmental standpoint, an engine can also be application or module with an open application programming interface (API) to which your application passes data in order to accesses the engine's processing capabilities.

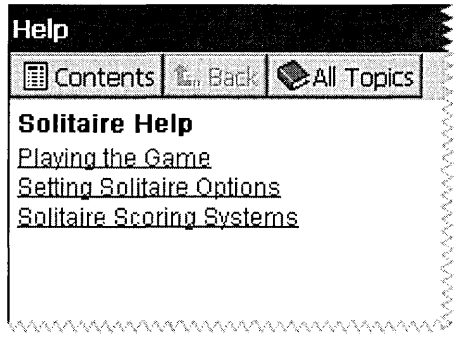
An engine can be another application, such as the Microsoft Help for Windows CE system. Applications pass Help files to the Help system, and Help for Windows CE displays those files. An engine can also be part of the operating system (OS), such as the spelling checker. An application passes the spelling checker text strings, and the spelling checker checks the spelling of the words in those strings.

This section discusses the Help and spelling checker engines, how they work, and how you can use their capabilities in your applications.

Creating a Help System

Using the Help for Windows CE engine, you can display Hypertext Markup Language (HTML)–based Help files on your Handheld PC or Palm-size PC device. Help for Windows CE consists of Peghelp.exe and an HTML rendering application that enables a user to display HTML-based Help files. The display window for Help for Windows CE contains a content area and a toolbar. This window is either full-screen or partial-screen, depending on the specific platform.

The following illustration shows a Help for Windows CE window for the Handheld PC.



Currently, there are three versions of Help for Windows CE: 2.0, 2.01, and 2.10. The version numbers correspond to the version of Windows CE that the Help engine was released with. To determine which version of Help for Windows CE you are using, consult the platform-specific sections of the SDK. The following table shows the user interface (UI) elements that are contained in the different versions of Help for Windows CE.

Button	Versions 2.0 and 2.10	Version 2.01
Contents	Displays the first-level contents for the current Help file.	Displays the first-level contents for the current Help file.
Back	Displays the previous topic view during the current session. The history is not persistent from session to session.	Displays the previous topic in the history list. The history is persistent from session to session and retains the last 10 visited topics.
Forward	N/A	Displays the next topic in the history list.
All Topics/ Other Help	Displays a list of Microsoft-provided Help files.	Displays a list of Microsoft-provided Help files.
Full Screen	Displays Help in a full screen view.	N/A
Partial Screen	Displays Help in a partial screen view.	N/A

The following table describes the three ways a user can gain access to Help files.

Action	Versions 2.0 and 2.10	Version 2.01
Help command, on the **Start** menu	Displays a list of Microsoft-provided Help files	Displays the context-sensitive Help topic for the current area of the UI
? button	Displays the context-sensitive Help topic for the current area of the UI	N/A
ALT + H	Displays the context-sensitive Help topic for the current area of the UI	Displays the context-sensitive Help topic for the current area of the UI

A Help system for Windows CE consists of HTML files and graphics. Help for Windows CE version 2.10 includes support for an **All Topics** list, using link (.lnk) files. You can also create an index.

▶ **To write a Help file**

1. Create the Help file.
2. Add content to the Help file.

   If necessary, check your content with an HTML and link validator.
3. Link the file into your application.

Creating the Help File

To create Help for a Windows CE platform, first create the appropriate files. Use an HTML editor, such as NotePad or Microsoft FrontPage®, a web site creation and management tool, to do so. The following table shows what extensions to use for your Help files.

Help for Windows CE version number	File extension
2.0	.htp
2.01	.htm
2.10	.htm

Note Other than the file extension, .htp and .htm files are identical.

Adding Content to a Help File

Once you create the Help file, use the Windows CE HTML version 3.0 subset to add content. This section discusses the differences between HTML 3.0 and the Windows CE HTML version, including the following topics:

- General Content Guidelines
- Using Jumps in a Help File
- Using Graphics in a Help File
- Separating Help Topics
- Creating an Index

The section concludes with a code example of an HTML topic for Help for Windows CE.

General Content Guidelines

Use the following guidelines to create Help for Windows CE:

- Do not use HTML Help on a Windows CE–based device. HTML Help files do not run on Windows CE platforms.
- Try to make the UI intuitive so that the user does not need Help. If possible, move all Help text into the UI.
- Because the user might not have ready access to the manual, supply as much information as possible in your Help file.
- When writing Help content, use abbreviations and other shorthand methods. In addition to taking up less memory, abbreviations let the user quickly scan a Help topic. For example, use "Choose **File>Open**" instead of "On the **File** menu, choose **Open**."
- Limit the number of styles you use in the Help file to make your Help file consistent and easier to read.
- Use generic wording in your Help files. Because you do not necessarily know what type of device your Help file will run on, you can increase portability by using generic phrases such as "Choose **Find**" instead of "Tap **Find**" or "Click **Find**."
- Although graphics are useful in Help files, limit your use of graphics, to conserve memory. If you must use a graphic, use a black-and-white version.

Using Jumps in a Help File

You can create two kinds of jumps in a Help for Windows CE file:

- A jump that points to another file. This type of jump uses the standard <A> tag. For example:

  ```
  <A HREF="soltr.htm">
  ```

- A jump that points to another anchor within a file. For example:

  ```
  <A HREF="soltr.htm#about">
  ```

  Even if both the jump and the anchor are within the same file, this type of jump requires the file name, in addition to the anchor name.

Using Graphics in a Help File

You can include both graphics and text in your Help for Windows CE files. Use the .2bp file extension for your graphics; Help for Windows CE does not support the .jpg or .gif file extensions.

The following HTML example shows how to add a bitmap to a topic, using the image source tag <IMG SRC> to specify the file name of the bitmap to include.

```
<IMG SRC="button.2bp">
```

If you want to use screen shots in your Help file, use the **Windows CE Zoom** application. This application ships with the Windows CE Toolkit. You can use this application to take a screenshot from a connected Windows CE–based device.

Separating Help Topics

Help for Windows CE uses the <!-- PegHelp --!> tag to separate each topic in a Help file. There is a space on both sides of the word PegHelp. Place this tag at the end of a topic to mark the section of text as an individual topic in Help for Windows CE.

Note Use some sort of tag, such as <!-- **Topic Break** --!>, to mark topic breaks. While not necessary, tags such as this can help you locate a topic in HTML code more easily.

Creating an Index

An index helps the user navigate through a Help file by listing topics alphabetically. Most OEMs create a general index for the applications that ship on their platform. Do not attempt to alter this index. Instead, create your index in your Help file.

The following code example shows a five-word index.

```
<A NAME="index"></A><B>Index</B>
<A HREF="inkwrite.htm#pagestyle">backgrounds, yellow</A><BR>
<A HREF="inkwrite.htm#create_a_bulleted_list">bulleted lists</A><BR>
<A HREF="inkwrite.htm#DocCreate">documents, creating</A><BR>
<A HREF="inkwrite.htm#DocWork">documents, editing</A><BR>
<A HREF="inkwrite.htm#mailrecipient">documents, e-mailing</A><BR>
<A HREF="inkwrite.htm#pagestyle">documents, page style</A><BR>
<A HREF="inkwrite.htm#printingdocument">documents, printing</a><br>
```

To save memory, do not create an index for every Help file. In general, create an index only if the table of contents for your Help file has 10 or more topics.

HTML Topic Example

The following code example shows a topic from **Solitaire** Help.

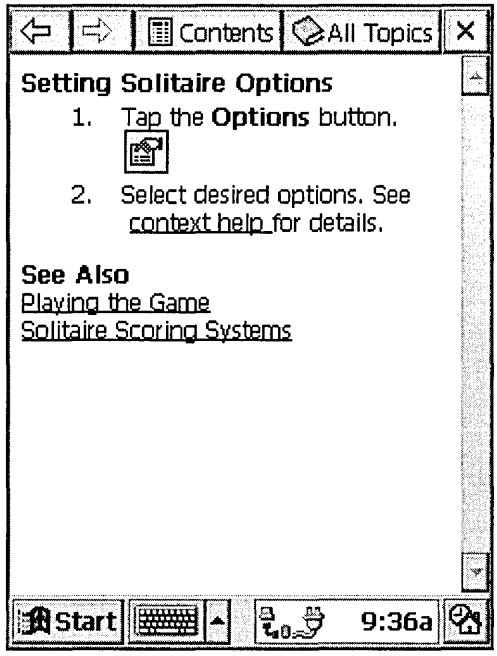
```
<A NAME ="setting_solitaire_options"></A><B>Setting Solitaire
Options</B>

<OL>
<LI>Tap the <b>Options</B> button.<BR>
<IMG SRC="_optonB.2bp">
<LI>Select desired options. See <A
HREF="wince.htm#view_toolbar_button">context help</A> for details.
</ol>

< B>See Also</ B><BR>
<A HREF="soltr.htm#playing_the_game">Playing the Game</A><BR>
<A HREF="soltr.htm#solitaire_scoring_systems">Solitaire Scoring
Systems</A>

<BR CLEAR=ALL >
<!-- PegHelp -->
<HR>
<!-- **********Topic Break********** -->
```

The following illustration shows the preceding code example on the Palm-size PC.



Testing a Help File

When you finish adding content to your Help for Windows CE file, test the links in the file. Because Help for Windows CE requires you to insert the name of the file in a jump, some link validators do not work correctly with Help for Windows CE files.

▶ **To test Help for Windows CE files in an incompatible link validator**

1. Create a copy of your Help file.
2. In this copy, replace all instances of .htm# with #.
3. Run the link validator on the copied file.

   This test will show any errors both in the copied file and the original file.
4. Make any changes you need in the original file.

Adding Help to an Application

Once you test your Help file, link the Help file to an application in one of two ways: through the **All Topics** list or through context-sensitive Help.

Adding a Help File to the All Topics List

The **All Topics** list is a list of HTML links to all the available Help files on a Windows CE platform. The **All Topics** list for Help for Windows CE versions 2.0 and 2.01 is a separate HTML file that lists links to all the Help files on the Windows CE platform. The manufacturer creates this file; do not alter it. Version 2.10 creates an **All Topics** list dynamically by selecting the name of the first topic in each Help file and a corresponding link (.lnk) file.

▶ **To add a topic to the All Topics list for version 2.10**

1. Name the first topic in each Help file **Main_Contents**.

   Place this name only in files that you want to include in the table of contents.

2. Include "Main_Contents" in the <META> tag for each of these files.

   The following code example shows how to include a topic in the dynamic table of contents.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
<HEAD>
<META HTTP-EQUIV="Htm-Help" CONTENT="soltr.htm#Main_Contents">
<TITLE>Solitaire Help</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000">
<!-- PegHelp -->
<P><A NAME="Main_Contents"></A><B>Solitaire Help</B></P>
<A HREF="soltr.htm#playing_the_game">Playing the Game</A><BR>
<A HREF="soltr.htm#setting_solitaire_options">Setting Solitaire
Options</A><BR>
<A HREF="soltr.htm#solitaire_scoring_systems">Solitaire Scoring
Systems</A><BR>
<BR CLEAR=ALL>
<!-- PegHelp --><HR>
<!-- ************************Topic Break************************* -->
```

3. Create a link file for each Help file in the table of contents.

   Create a link file with **NotePad** or another ASCII editor. Name the link file the way you want Help to display the link in the table of contents. For example, the **Solitaire** Help file is Soltr.htm and the link file is Solitaire.lnk. The link in the table of contents is Solitaire. You can place spaces in the link file name. For example, the link file name for **Pocket Word** is Pocket Word.lnk.

   The text in the link file tells Help for Windows CE where to aim the link in the table of contents. The following code example is one entry in the Solitaire.lnk file:

   ```
   18#\windows\soltr.htm
   ```

   The number 18 in this example represents the number of characters in \Windows\Soltr.htm

4. During installation, install your link file in the \Windows\Help directory.

   Install the Help file anywhere you want in the Windows CE platform.

Creating Context-Sensitive Help

The most common way to call Help in Help for Windows CE versions 2.0 and 2.10 is from the ? button in the upper-right corner of the application screen or dialog box. The user accesses Help in version 2.01 by choosing **Help** on the **Start** menu.

The following code example shows how to call Help from an application by using the **CreateProcess** function.

```
CreateProcess(TEXT,"peghelp.exe",
TEXT("file:inkwrite.htm#font_and_ink"), NULL, NULL, FALSE, 0, NULL,
NULL, NULL, NULL;
```

Pass the "peghelp.exe" string in as the first parameter, using the **TEXT** macro first to convert the string to Unicode. Use **TEXT** on the second argument, this time passing a string containing the file name of the contents or specific Help topic that you want to display. Set all other parameters to NULL, FALSE, or 0, respectively.

Creating Pop-up Help

Many localizers use pop-up Help to provide complete names or information when the UI does not provide enough display area. To access pop-up Help, a user selects and holds the title of an item in the UI. Windows CE either provides more information or displays the message "No help provided."

Note Pop-up Help is not part of the Help for Windows CE application, but is mentioned here because it performs a similar function.

Create a pop-up Help topic in the same manner as a Windows CE ToolTip. The following code example shows how to declare a pop-up Help topic in a resource file for a Palm-size PC.

```
CONTROL     "Static text~~This is a pop-up Help topic\rIt is also a
            ToolTip",IDC_STATIC,"TTSTATIC",WS_VISIBLE,19,24,56,8
```

Working with the Spelling Checker

Windows CE provides a spelling checker engine. Unlike the Help engine, the spelling checker is integrated into the Windows CE operating system as a module. Also unlike the Help engine, the spelling checker does not have any UI. Instead, your application passes text strings directly to the spelling checker. The spelling checker searches supplied dictionaries and returns suggestions for correct spelling. You must decide how to display this data to users.

▶ **To integrate the spelling checker engine into an application**

1. Initialize the spelling checker engine with a call to the **SplInit** function.

   Use the call to **SplInit** to add any user-defined spelling dictionaries and to initialize user settings.

2. Use the **SplCheck**, **SplSuggest**, **SplOptionSet**, **SplReplace**, and **SplLimitSet** functions to check spelling, suggest alternate words, and replace misspellings.

3. When you are finished, close the spelling checker engine with a call to the **SplQuit** function.

Initializing the Spelling Checker

Before you can use the spelling checker in your application, call the **SplInit** function to initialize a spelling session. **SplInit** performs the following tasks to initialize the spelling checker engine:

- Creates a handle for the spelling session

  Your application uses this handle to associate a spelling checker function call to a specific spelling session.
- Loads the main, internal, and external user dictionaries

Creating the Spelling Checker Handle

A *spelling session* is defined as all of the resources that the spelling checker is using for a particular application, including any dictionaries and created structures. The first instruction your spelling checker performs is to create a spelling checker handle. Your application uses this handle to associate a spelling checker function call to a specific spelling session. Because a device may have several applications using the spelling checker engine, all of the functions in the spelling checker API use the spelling checker handle to distinguish different spelling sessions.

Loading the Dictionaries

Once **SplInit** creates the spelling checker handle, the function loads the four required dictionaries and any other optional dictionaries:

- First, **SplInit** loads the read-only main dictionary. Microsoft defines the main dictionary for each localized version of Windows CE. Currently, Windows CE does not support replacing the main directory with a dictionary supplied by an application. However, Windows CE can specify additional dictionaries to load while initializing the spelling checker.
- Next, in addition to loading the main directory, **SplInit** creates three internal user dictionaries for each spelling session. **SplInit** creates these dictionaries empty. During a spelling session, you can add to and modify these dictionaries. However, Windows CE does not specifically save these dictionaries when you exit a spelling session.

You can instruct **SplInit** to load external user dictionaries at the beginning of a spelling session. An external user dictionary is a dictionary that is defined by an application, user, or other third party. A common use of an external user directory is to store unique words that are not found in the main directory but are commonly used by an individual user. You can save the external dictionary at the end of a spelling session, and load it again at the beginning of the next spelling session.

Use the *ppwz* parameter of **SplInit** to point to a list of external user dictionaries. Each element on the list includes the dictionary path and file name. **SplInit** assigns identifiers to the internal user dictionaries and to any external user dictionaries pointed to by *ppwsz*.

The following table describes the values you can provide in the *ppwz* parameter.

Dictionary name	*nID* value	Description
SPL_IGNORE_LEX	–3	Contains words that the spelling checker finds, but the user does not want to change. The spelling checker ignores words in this list for the remainder of the spelling session.
SPL_CHANGE_ALWAYS_LEX	–2	Contains words that the spelling checker finds, paired with words with which the spelling checker replaces subsequent occurrences of the word. For example, thru paired with through.
SPL_CHANGE_ONCE	–1	Contains words that the spelling checker finds and the user changes. The spelling checker continues to find subsequent occurrences of these words.
External user dictionary 1	0	The first external user dictionary described in *ppwz*.
External user dictionary 2	1	The second external user directory described in *ppwz*.
External user dictionary *clex*	*clex*-1	The last external user directory described in *ppwz*.

Use the *nID* value to direct the spelling checker API to a specific dictionary. The *clex* value is the total number of external dictionaries you provide in the *ppwz* parameter.

Initializing the Spelling Session with Single and Multiple Applications

Initializing a spelling session requires the device to load a set of data into active memory. Further, the initialization process might reserve a large amount of resources. Monopolizing resources might not be a problem if a user is running only one application on the device. However, you need to take into account the possibility that other applications may need to use the spelling checker at the same time as your application.

In situations where you can be certain that only one application uses the spelling checker, initialize the spelling session when the application starts. Initializing the spelling checker at the beginning of an application means that you can avoid unnecessary processing in the middle of your application. Similarly, keep the spelling session active until the application ends.

Initializing a spelling session is different when you program in a multiple-application environment. A spelling session has exclusive access to external user directories until the calling application closes the spelling session. This means that a second application cannot load external user dictionaries already in use. Therefore, if your application runs in an environment where other applications use the spelling checker, you may want to initialize the spelling checker when the application requests it. Initializing the spelling checker only when an application needs it reduces the number of external user dictionaries tied up in any given application. When the application is done with the spelling checker, call the **SplQuit** function to release the dictionaries so that other applications can use them.

The main directory is available for multiple spelling sessions because it is read-only. The internal user directories are created for each spelling session and are removed at the end of each spelling session. Therefore, access to the internal user dictionaries by multiple spelling sessions is not a concern.

Setting the Spelling Session Options

After initializing a spelling session, call the **SplOptionSet** function to customize the Spelling Checker operations. For example:

- Use **SplOptionSet** to instruct the spelling checker to find repeating words, ignore Roman numerals, or ignore capitalization errors.
- Call **SplOptionGet** to retrieve the current settings of the spelling session. You can then display the options to the user, and change the options with a call to **SplOptionSet**. By default, all options are turned off at the beginning of the spelling session.

Using the Spelling Checker

Once you initialize the spelling session and set the spelling checker options, you can call a variety of functions to check spelling and suggest alternate words to use.

▶ **To use the spelling checker**

1. Set up the **SPLBUFFER** structure.

   SPLBUFFER contains data concerning the words to check, the location of the input and output buffers, and the status of the spelling session in progress.
2. Perform the spelling check with a call to the **SplCheck** or **SplReplace** functions:
   - **SplCheck** checks the input buffer of **SPLBUFFER** with the dictionaries.
   - **SplReplace** checks the input buffer against the dictionaries and suggests a limited number of possible alternative spellings.
3. If you called **SplCheck**, call **SplSuggest** to receive more possible alternative spellings.
4. If necessary, alter the spelling of the misspelled word or ignore the suggestion and move to the next word.

Setting Up the SPLBUFFER Structure

Before you call **SplCheck** or **SplSuggest**, set up the **SPLBUFFER** structure. A **SPLBUFFER** structure contains a variety of data for the spelling checker functions. First, it contains the input buffer that holds the text that you are checking. It also contains parameters for containing spelling suggestions returned by the spelling checker, the type of spelling error that occurred, where in the input buffer the spelling checker should look next, and how the spelling checker should proceed.

▶ **To set up the SPLBUFFER structure for a call to SplCheck**

1. Set *pwzIn* to point to the buffer containing the text that you want to check the spelling of.
2. Set *IwcIn* to 0.

   IIwcIn contains the offset for where **SplCheck** begins searching for misspelled words. Setting *IwcIn* to 0 causes **SplCheck** to begin searching at the beginning of the input buffer.
3. Set *cwcIn* to the size of the input buffer relative to *pwzIn*.

4. Set *dwMode* to SPL_NO_STATE_INFO or SPL_STARTS_SENTENCE.
   - Use SPL_NO_STATE_INFO if you do not know anything about the contents of the input buffer.
   - Use SPL_STARTS_SENTENCE if you know that the input buffer contains text that starts at the beginning of a sentence.
5. Set *dwError* to receive an error code from **SplCheck**.

   SplCheck places the results of the spelling session in *dwError*.
6. Set *iwcErr* to receive the index of the location of the spelling error, if such an error exists.
7. Set *cwcErr* to receive the length of the word containing the spelling error, if such an error exists.

Performing a Spelling Check with the SplCheck Function

Once you have set **SPLBUFFER**, call **SplCheck**. **SplCheck** finds the first word in the input buffer and searches the main dictionary, internal user dictionaries, and external dictionaries for that word.

If the spelling checker finds that the word is spelled correctly, it moves on to the next word. If **SplCheck** finds a misspelled word, it returns the location of the word in the buffer, the length of the word, and an error code describing the general type of misspelling. If the spelling checker gets to the end of the input buffer without finding a spelling error, it returns SPL_OK to **SPLBUFFER**.

Performing a Spelling Check with the SplReplace Function

You can also perform a spelling check with the **SplReplace** function. In addition to checking the spelling of the text in the input buffer, **SplReplace** returns a list of suggested replacements in the output buffer. However, **SplReplace** returns only a limited list of possible suggestions. You can define the limits placed on **SplReplace** with the **SplLimitSet** function. A common use of **SplLimitSet** is to limit the suggestions to obvious mistakes, such as suggesting "the" for "teh."

Note Because the spelling checker API is designed for a variety of purposes, **SplReplace** does not replace the checked word with a suggestion. You need to add this capability to your application.

Receiving Spelling Suggestions from the Spelling Checker

Once you have determined that a word in the input buffer of **SPLBUFFER** is misspelled, call **SplSuggest**. Like **SplReplace**, **SplSuggest** returns one or more suggestions for a misspelled word. However, **SplReplace** can return up to eight possible suggestions.

In addition to returning spelling suggestions, **SplSuggest** can also **score** the suggestions. A score is a number that indicates how much a replacement word differs from the original misspelled word. A low score indicates that the misspelled word was changed slightly, while a high score indicates that the word was changed a great deal. You can use suggestion scores in your application to exclude some words from the list of alternatives.

Once **SplSuggest** finishes processing, you can access the output buffer through the *aspl* member of **SPLBUFFER**. The *aspl* member is an array of **SPLSUGGEST** structures. The first member of a **SPLSUGGEST** structure points to a word in the output buffer. If you chose to use scores, the second member contains a score for that word.

▶ **To receive spelling suggestions from the spelling checker**

1. If you want to use scores, set SPL_SCORE_SUGGESTIONS with **SplSetOptions**.
2. Set the size and location of the output buffer with the *cwcOut* and *pwszOut* parameters of **SPLBUFFER**.

   This **SPLBUFFER** is the same structure returned by your original call to **SplCheck**.
3. Set the *aspl*, *cwcUsed*, and *cspl* parameters of the **SPLBUFFER** structure to receive the appropriate data.

   SplSuggest returns an array of pointers to the output buffer in *aspl*, the size of the output buffer used in *cwcUsed*, and the number of suggestions in *cspl*.
4. Call **SplSuggest**.

Changing a Spelling Error in Your Application

Once you receive the suggested spellings, you must implement the actual change. A common practice is to alter the misspelled word in the input buffer. When you finish the entire spelling check on the text string in the input buffer, you can then make any modifications to the actual information.

Ignoring a Spelling Error or Moving to the Next Word

There may be times when you want to skip an error without correcting it. For example, a user may type in his or her name and choose not to add it to one of the dictionaries. Because the name may not be in any defined dictionary, it will show up as a spelling error. Therefore, you need to instruct the spelling checker to skip over a word. Use the same technique to move to the next misspelled word in the input buffer.

▶ **To ignore a spelling error or move to the next word**

1. To continue with a spelling check:
   - If you are skipping a misspelled word, set the *dwMode* parameter of **SPLBUFFER** to SPL_IS_CONTINUED.

     SPL_IS_CONTINUED instructs the spelling checker to ignore the spelling error.
   - If you are continuing with the spelling check after correcting a misspelled word, set *dwMode* to SPL_IS_EDITED_CHANGE.
2. Call **SplCheck**.

   SplCheck will replace the value contained in *iwcIn* with the value contained in *iwcErr* plus the value contained in *cwcEIrr*. **SplCheck** continues the spelling check just after the misspelled word.

Modifying External and Internal Dictionary Lists

There may be times when you need to check the spelling of words that are not in the main dictionary or in the external user-defined dictionary. For example, a user may want to enter his or her name, or may want to use a variety of technical language in a specific document. You can modify both the internal and external dictionary lists to suit the needs of your users.

The internal and external user dictionaries are lists of null-terminated words. Two null characters in a row indicate the end of a dictionary. The following code example shows a series of possible listings from a user dictionary:

```
…thier\0their\0wierd\0weird\0\0
```

Words are listed with the incorrect word first and the correct word second. This is the format that the SPL_CHANGE_ALWAYS and SPL_CHANGE_ONCE internal dictionaries use.

The following table describes the functions that you can use during a spelling session to modify internal and external user dictionaries.

Function	Description
SplAddUserDict	Adds a single word or word pair to the specified dictionary
SplRemUserDict	Removes a single word or word pair from the specified dictionary
SplClrUserDict	Clears all entries from the specified dictionary

SplAddUserDict, **SplRemUseDict**, and **SplClrUserDict** all use the spelling dictionary identifiers specified in the *ppwz* parameter of **SplInit**.

The **SplEnumUserDict** function enumerates the contents of a specified internal or external user dictionary. The function writes the dictionary contents to the output buffer specified by the *pwszOut* parameter in the **SPLBUFFER** structure. One use of enumeration is to present a dictionary's contents to a user, who can then select words to be added and removed with **SplAddUserDict** and **SplRemUserDict**.

Ending the Spelling Session

Once you finish with the spelling checker, call the **SplQuit** function to end the spelling session. **SplQuit** performs the following tasks:

- Invalidates the spelling session handle that was created in the **SplInit** function call
- Frees all resources that were allocated by **SplInit**
- Deletes the internal user dictionaries
- Saves external user dictionaries to the files that are specified in the **SplInit** parameters

PART 2

Connection Services

This part contains the following chapters:

- **Overview of Connection Services**
- **Working with RAPI**
- **Managing the Connection Partnership**
- **Synchronizing Data**
- **Installing Applications**

CHAPTER 6

Overview of Connection Services

One of the fastest-growing market segments in the computing world today is for portable desktop companions. In creating the Windows CE operating system (OS), Microsoft fulfilled a need for a compact, scalable OS that communicates with the Internet, networks, and desktop computers while delivering the same familiar and easy to use Windows interface. Products such as the Handheld PC and the Palm-size PC device categories deliver connectivity that allows a strong partnership between a Windows CE–based device and the Windows-based desktop computer.

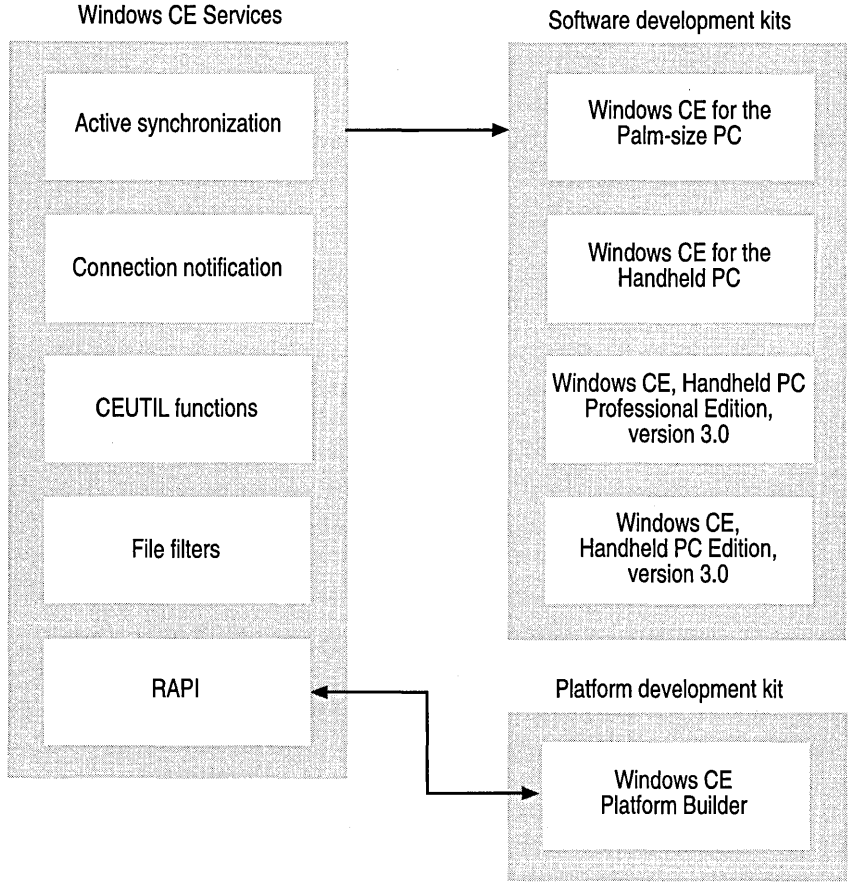
To accommodate the need for this highly functional connectivity, Windows CE provides a multitude of functions that allow communication between applications on the desktop computer and the remote Windows CE–based device.

Enabling a Partnership in Windows CE

This chapter discusses the tools and components that you use to design applications that enable functional partnerships between a desktop computer and a Windows CE–based device. By using the following components, you program applications with the capability of making Windows CE–based devices the perfect mobile companion to a desktop computer:

- Remote API (RAPI)
- File filters
- Connection notification
- CEUTIL functions
- Windows CE Services

The following illustration shows the relationship between the various components that are available on the Windows CE OS and Windows CE–based devices.



RAPI

This set of application programming interfaces (APIs) allows applications running on the desktop computer to invoke functions directly on the remote Windows CE–based device. Windows CE provides one-way remote API (RAPI), with which the Windows CE–based device is the RAPI server and the desktop computer is the RAPI client. The application runs on the desktop computer—the client side—which calls functions that are executed on the Windows CE–based device, the server side.

RAPI under Windows CE is designed so that desktop computer applications can manage the Windows CE–based device remotely. The exported functions relate to the registry, file system, and databases, as well as to functions for querying the system configuration. While most RAPI functions are duplicates of functions in the Windows CE API, a few functions extend the API. You use these functions to initialize the RAPI subsystem and enhance performance of the communication link by compressing iterative operations into one RAPI call.

Essentially, RAPI is a remote procedure call (RPC). RAPI communicates requests from a desktop computer application to invoke a function and returns the results of that function.

File Filters

Windows CE file filters are Component Object Model (COM) objects that exist on the desktop computer. They are loaded and called by *Windows CE Services*, which is an application that runs on the desktop computer and enables connection with a Windows CE–based device. When a file is copied to or from the Windows CE–based device to the desktop computer by using Windows CE Services, Windows CE Services checks to see whether a file converter is registered for the file type being transferred. If so, Windows CE Services loads the file filter and issues a request to convert the file. This all takes place on the desktop computer (the client side of the link), where most processing between the desktop computer and the device takes place.

If a file is being exported from a Windows CE–based device to the desktop computer, it is copied in its original form to the desktop computer, converted by the file filter, and stored on the desktop computer. In much the same way, if a file is being imported to a Windows CE–based device, it is converted by the file filter, and then the file is copied to the Windows CE–based device.

Windows CE file filters are dependent on the Mobile Devices folder that is provided by Windows CE Services. Only files that are moved to and from a Windows CE–based device by users dragging them to the Mobile Devices folder are converted. If a file is transferred to a Windows CE–based device by any other method, such as downloading a file from the Internet by using Pocket Inbox, the file filter is not loaded and the file is not converted.

Connection Notification

Windows CE Services gives you two ways of notifying desktop computer-based applications when a connection is made with a Windows CE–based device:

- Registry-based notification, in which all the applications that are listed under a given registry are launched. When a connection is broken, all applications that are listed under a different key are launched.
- COM interface–based notification, which involves two interfaces: **IDccManSink**, which must be implemented by the application seeking notification, and **IDccMan**, which is provided by Rapi.dll.

Registry-based notification is appropriate for applications that do not need some control of the connection manager or the ability to register and deregister for connection notifications. The COM interface–based notification method is more complex than registry-based notification; however, it provides you with some control of the connection manager and the ability to register and deregister for connection notifications.

CEUTIL Functions

Windows CE Services uses the registry on the desktop computer to store large amounts of information about the Windows CE–based devices that have created a partnership with the desktop computer. Windows CE Services uses the registry on the desktop computer to store configuration information. While most of these registry keys are documented, if you access them by name you are assuming that those key names always remain the same. However, this might not be the case, especially in international versions of Windows where registry keys can be in a different language.

The CEUTIL DLL exports functions that provide an abstraction layer over the registry keys that are used by Windows CE Services. Using this DLL allows a desktop computer application to query the devices that are currently registered, and to add or delete registry values underneath the keys that hold data for specific devices. The CEUTIL DLL does not communicate with a remote Windows CE–based device. Instead, it looks in the desktop registry for information that has previously been placed there by Windows CE Services.

You can use the CEUTIL DLL to manage desktop registry entries for Windows CE Services; register desktop file filters and synchronization services; access device-partnership settings that are used for both file filters and synchronization services; and add custom menu items to Windows CE Explorer.

Windows CE Services

The Windows CE system comprises three components:

- The OS, which runs on the Windows CE–based device.
- The Windows CE–based device, which runs Windows CE–based applications.
- Windows CE Services, an application that provides connection services and runs on the desktop computer.

Windows CE Services is an application that runs on the desktop computer and facilitates connections with the Windows CE–based device. The helper DLLs, communications support, and functionality described in the preceding sections are available in the Windows CE Services application. Though not supported through the Windows CE OS, Windows CE Services is shipped with Windows CE platforms, such as a Handheld PC running Microsoft Windows CE for the Handheld PC.

You add components to Windows CE Services to allow data sharing between a desktop computer and a platform-specific Windows CE–based device, and also between a Windows CE–based application and its counterpart on the desktop computer. By defining application-specific file filters and registering applications with Windows CE Services, you ensure data transmission and conversion capability between desktop computer and Windows CE based–device formats.

Users open Windows CE Services by double-clicking the Mobile Devices icon on their desktop computer. In the resulting window, users can establish a connection partnership between their desktop computer and the Windows CE–based device by choosing options from the Partnership Wizard during initial setup. Later, users can change connection settings by choosing options that are available on the **Mobile Devices** menu.

The following screen shot shows the **Tools** menu in the Mobile Devices window.

Once the connection partnership process is complete, Windows CE Services provides the following functionality between the desktop computer and the Windows CE–based device:

- Data synchronization
- File conversion between the desktop computer and device formats
- Importing and exporting database tables
- Preparing the desktop for remote connections

Additionally, Windows CE Services provides the following functionality on the Windows CE–based device:

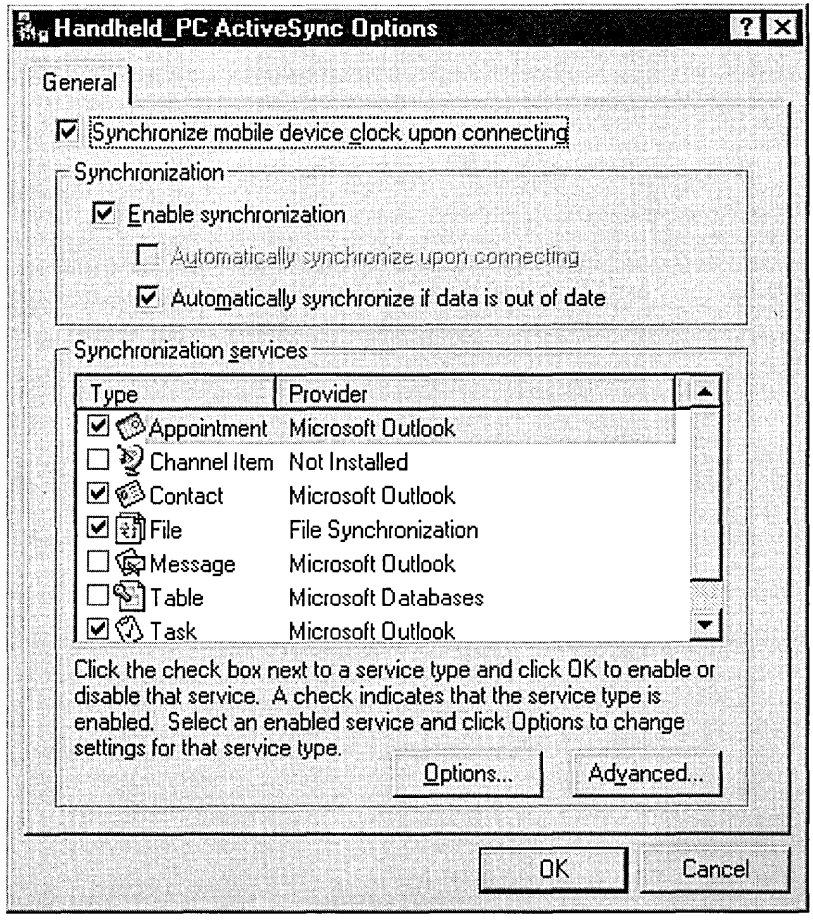
- Backing up and restoring device data
- Adding and removing programs

Synchronizing Data with ActiveSync

Windows CE Services contains Microsoft ActiveSync™ technology, which provides data replication and synchronization. ActiveSync does this by comparing the data on a Windows CE–based device with the data on the desktop computer to which the device is connected. After comparing data, ActiveSync updates both the device and the desktop computer with the most recent information.

ActiveSync is composed of the service manager and the service provider. The service manager is a synchronization engine that is built into Windows CE Services and resides on both the desktop computer and the Windows CE–based device. The service provider comprises two modules that you must implement in your application to perform the synchronization tasks that are specific to your data. One module, called the desktop provider module, resides on the desktop computer and the other module, called the device provider module, resides on the device. By creating and registering the service provider, you decide which data is tracked for changes by the service manager. And by creating the proper file converters, you enable ActiveSync to maintain the same information on a device as on the desktop computer.

Built-in functionality gives users the option of choosing a subset of information to synchronize. For example, they can synchronize only e-mail, only certain files, only their Pocket Outlook data, or only certain categories of information that they define, such as business contacts or personal appointments. The following screen shot shows the **ActiveSync Options** page for the Handheld PC.



ActiveSync gives a user the option to synchronize data either manually, automatically on connection, or continuously while a connection is active so that changes appear immediately on both the device and the desktop computer. In addition, the user can synchronize both at their desktop computer and away from it, over serial, infrared, Ethernet LAN, and modem connections. Users can synchronize their device information with two desktop computers, such as their home and office computer, or they can synchronize information on one desktop computer with several devices—for example, to share files within a department.

Backing Up and Restoring Device Data

Using the built-in Windows CE Services backup and restore feature, a user can create a backup file on their desktop computer containing all of the files, databases, Pocket Outlook data, RAM-based programs, and other information on their device. If their device data becomes lost or corrupted, they can restore it from this backup file.

Transferring Files Between a Device and the Desktop Computer

Windows CE Services provides the Mobile Devices folder in Microsoft Windows Explorer on a user's desktop computer. When a device is connected to the desktop computer, this folder provides a view of the files and folders on the device. Users can move, copy, and automatically convert files between the device and the desktop computer by dragging them to and from this folder.

Adding Programs to and Removing Programs from a Device

The Application Manager, in the Mobile Devices folder, enables users to see programs that are currently installed on their device, as well as applications on their desktop computer that are available for installation on the device. They can also add programs to and remove programs from the device or a memory card.

Importing and Exporting Database Tables

When the user chooses the **Import Database Table** command on the **Tools** menu of the Mobile Devices folder, an **Open** window is exposed in which the user can select the location of the database. This option can be used to move database tables from the desktop computer to the device. Alternately, by choosing **Export Database Table**, the user initiates the copy and convert process. During this process, database information from the device is collected, converted, and transferred to the selected location on the desktop computer.

Preparing for Remote Connection

If users want to establish a remote connection between their desktop computer and their device, Windows CE Services helps them prepare their device for the remote connection. They follow a series of steps to set up a dial-up connection to a modem, a dial-up connection to a network, or a network (Ethernet) connection.

CHAPTER 7

Working with RAPI

This section discusses how to use remote application programming interface (RAPI), which you can use to port your knowledge of standard Microsoft Win32 APIs into the Microsoft Windows CE programming environment. The main difference is that with RAPI, you can write data to or read data from a Windows CE–based device remotely with a desktop computer.

Invoking Functions from a Desktop Computer

RAPI gives an application running on a desktop computer the ability to invoke function calls on a Windows CE–based platform. The desktop computer is the RAPI client and the Windows CE–based platform is the RAPI server. The communication uses Windows Sockets API (Winsock) and can take place over a serial link, a modem connection, or a network connection.

The function calls behave much like the equivalent Windows CE functions. For the most part, RAPI functions have the same syntax, parameters, and return values as the corresponding Windows CE versions. The RAPI functions contain a **Ce** prefix to differentiate them from the Windows CE functions. Any other differences are noted in the reference documentation for the RAPI functions.

Note String and character parameters must be in Unicode format. Use the appropriate conversion routines, if necessary.

Initializing and Terminating Remote Applications

Before making RAPI calls, you must call the **CeRapiInit** or **CeRapiInitEx** function. These functions perform routine initialization and set up the communications link between the desktop computer and the Windows CE platform.

The **CeRapiInit** call is a synchronous operation. It does not return control to the application until a connection is made or an error occurs. In contrast, the **CeRapiInitEx** call is an asynchronous operation and it returns immediately. **CeRapiInitEx** continues the initialization until a connection is made, an error occurs, or there is a call to **CeRapiUninit**. Although **CeRapiInitEx** avoids blocking any threads, it is a more complicated method of initialization.

▶ **To initialize RAPI by using CeRapiInitEx**

1. Call **CeRapiInitEx**.

   If an error is returned, exit.
2. If successful, call **WaitForSingleObject** or **WaitForMultipleObjects** to wait on the event handle passed back in the **heRapiInit** member of **RAPIINIT**.
3. When **heRapiInit** is signaled, check for a successful connection or an error value.

Check the **hrRapiInit** member of the **RAPIINIT** structure for the final return value.

When you are finished with RAPI, call **CeRapiUninit** to terminate the connection and perform any necessary cleanup. Because creating and terminating connections are fairly expensive operations, establish and terminate the link only once per session, and not on a per-call basis.

The following code example shows how to use the **CeRapiInitEx** function. Following the **CeRapiInitEx** call, the **MsgWaitForMultipleObjects** function is used to wait on one of two events. The first event is when the event handle is passed back through the **heRapiInit** member of the **RAPIINIT** structure. The second event is when a user terminates a connection.

```
#define ARRAYSIZE(hArray)   sizeof (hArray) / sizeof (HANDLE)

enum
{
  WAD_ALLINPUT    = 0x0000,
  WAD_SENDMESSAGE = 0x0001,
};
```

```
DWORD WaitAndDispatch (DWORD nCount, HANDLE *phWait, DWORD dwTimeout,
                       UINT uFlags)
{
  DWORD dwObj;
  DWORD dwStart = GetTickCount ();
  DWORD dwTimeLeft = dwTimeout;

  for (;;)
  {
    dwObj= MsgWaitForMultipleObjects (nCount, phWait, FALSE, dwTimeLeft,
             (uFlags & WAD_SENDMESSAGE) ? QS_SENDMESSAGE : QS_ALLINPUT);

    if (dwObj == (DWORD)-1)
    {
      dwObj = WaitForMultipleObjects (nCount, phWait, FALSE, 100);

      if (dwObj == (DWORD)-1)
        break;
    }
    else
    {
      if (dwObj == WAIT_TIMEOUT)
        break;
    }

    if ((UINT)(dwObj - WAIT_OBJECT_0) < nCount)
      break;

    MSG msg;

    if (uFlags & WAD_SENDMESSAGE)
      PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE);
    else
    {
      while (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
        DispatchMessage (&msg);
    }

    if (INFINITE != dwTimeout)
    {
      dwTimeLeft = dwTimeout - (GetTickCount () - dwStart);
      if ((int)dwTimeLeft < 0)
        break;
    }
  }

  return dwObj;
}
```

```
HRESULT InitRapi (HANDLE hExit)
{
  RAPIINIT rapiinit = {sizeof (RAPIINIT)};
  HRESULT hResult = CeRapiInitEx (&rapiinit);

  if (FAILED(hResult))
    return hResult;

  HANDLE hWait[] = {hExit, rapiinit.heRapiInit};
  enum {WAIT_EXIT=WAIT_OBJECT_0, WAIT_INIT};

  DWORD dwObj = WaitAndDispatch (ARRAYSIZE(hWait), hWait, 1000, 1);

  // Event signaled by RAPI
  if (WAIT_INIT == dwObj)
  {
    // If the connection failed, uninitialize the
    // Windows CE RAPI.
    if (FAILED(rapiinit.hrRapiInit))
      CeRapiUninit ();

    return rapiinit.hrRapiInit;
  }

  // Either event signaled by user or a time-out occurred.
  CeRapiUninit ();

  if (WAIT_EXIT == dwObj)
    return HRESULT_FROM_WIN32(ERROR_CANCELLED);

  return E_FAIL;
}
```

Predefined RAPI Functions

RAPI includes a group of predefined functions that duplicate Windows CE functions on the desktop computer side of the connection. The following sections group RAPI functions by use. Because the actions of the functions are mirror-images of their Windows CE–based counterparts, details of each function are not provided. For information on a particular function, see the *Windows CE Platform SDK Reference*.

System Information Functions

The following table shows the RAPI system information functions. Most will be familiar to those of you working with Windows CE APIs, with the possible exception of **CeGetPassword** and **CeGetDesktopDeviceCaps**. The **CeGetPassword** function compares a string to the current system password. If the strings match, the function returns TRUE. Note that the comparison is case-specific. The **CeGetDesktopDeviceCaps** function is identical to the Windows CE counterpart, which is **GetDeviceCaps**.

System information functions

CeGetVersion	**CeGetDesktopDeviceCaps**
CeGlobalMemoryStatus	**CeGetSystemInfo**
CeGetSystemPowerStatusEx	**CeGetPassword**
CeGetStoreInformation	**CeCheckPassword**
CeGetSystemMetrics	**CeCreateProcess**

Database Functions

The following table shows the RAPI database functions. Many of them are supported in Windows CE 2.1 or later. Note that the RAPI functions that support the extended database API of Windows CE 2.1 and later are not exported by previous RAPI DLLs. Your application will not load if the desktop computer has a previous version of Rapi.dll and has attempted to implicitly load one of these functions.

Database management functions

CeCreateDatabase	**CeOpenDatabaseEx**
CeCreateDatabaseEx	**CeReadRecordProps**
CeDeleteDatabase	**CeReadRecordPropsEx**
CeDeleteDatabaseEx	**CeSeekDatabase**
CeDeleteRecord	**CeSetDatabaseInfo**
CeFindFirstDatabase	**CeSetDatabaseInfoEx**
CeFindFirstDatabaseEx	**CeWriteRecordProps**
CeFindNextDatabase	**CeMountDBVol**
CeFindNextDatabaseEx	**CeUnmountDBVol**
CeOidGetInfoEx	**CeEnumDBVolumes**
CeOpenDatabase	**CeFindAllDatabases**

Call the **CeFindAllDatabases** function to get information about all databases of a specified type. The information is returned in an array of **CE_FIND_DATA** structures.

You must free the memory allocated by the **CeFindAllDatabases** or **CeReadRecordProps** function by calling the **CeRapiFreeBuffer** function.

File and Directory Management Functions

The following table shows the RAPI file and directory management functions.

File and directory management functions

CeFindAllFiles	**CeSetFilePointer**
CeFindFirstFile	**CeSetEndOfFile**
CeFindNextFile	**CeCreateDirectory**
CeFindClose	**CeRemoveDirectory**
CeGetFileAttributes	**CeMoveFile**
CeSetFileAttributes	**CeCopyFile**
CeCreateFile	**CeDeleteFile**
CeReadFile	**CeGetFileSize**
CeWriteFile	**CeGetFileTime**
CeCloseHandle	**CeSetFileTime**

▶ **To retrieve path information**

- Call the **CeGetTempPath** function to get the path of the directory that is designated for temporary files.

  –Or–

  Call the **CeGetSpecialFolderPath** function to get the path of a specific desktop folder, which depends on the input parameter. The possibilities include the Recycle Bin, **Start** menu directory, document template directory, network directory, and folders for fonts or installed printers.

▶ **To retrieve other information**

- Call the **CeFindAllFiles** function to get information about all files and directories in a specified directory of the Windows CE object store.

  As with the **CeFindAllDatabases** function, the information is returned in an array of **CE_FIND_DATA** structures.

Also, you must free the memory allocated by the **CeFindAllFiles** function by calling the **CeRapiFreeBuffer** function.

Registry Management Functions

The following table shows the RAPI registry management functions.

Registry management functions

CeRegOpen KeyEx	**CeRegEnumValue**
CeRegEnumKeyText	**CeRegDeleteValue**
CeRegCreateKeyEx	**CeRegQueryInfoKey**
CeRegCloseKey	**CeRegQueryValueEx**
CeRegDeleteKey	**CeRegSetValueEx**

Shell Management Functions

The following table shows the RAPI shell management functions.

Shell management functions

CeSHCreateShortcut	**CeGetTempPath**
CeSHGetShortcutTarget	**CeGetSpecialFolderPath**

Window Management Functions

The following list shows the RAPI window management functions.

Window management functions

CeGetWindow	**CeGetWindowText**
CeGetWindowLong	**CeGetClassName**

Invoking Functions and Applications

Among all RAPI functions, there are two functions that invoke functions and applications residing on the Windows CE platform:

- **CeCreateProcess**

  This function creates a new process that runs a specified executable file residing on the Windows CE platform.

- **CeRapiInvoke**

  This function remotely executes a function residing on the Windows CE platform and provides for both input parameters and output data. It operates in either of two modes:

  - Block, which is known as synchronous. In *block mode*, the caller passes both input parameters and output data in a single buffer. Because this is a synchronous call, all input data must be present in memory at the time of the call and all output data must be present before the function finishes.
  - Stream, which is known as asynchronous. In *stream mode*, an **Istream**-type interface is used to exchange arbitrarily-sized data in any order and direction. The caller can pass input data in a single buffer, but from that point on all data should be exchanged through the stream. Because the data can be read, written, and stored in chunks, stream code is significantly faster than block mode. The interface used is based on **Istream**, but has two additional methods to allow you to do time-outs.

Note **LocalAlloc** allocates the memory passed for both the *pInput* and *ppOutput* parameters of **CeRapiInvoke**. The called function frees the input memory allocation, and the calling application frees the output memory allocation.

Handling RAPI Errors

In addition to errors associated with their non-RAPI counterparts, RAPI functions can fail because of RAPI-related errors. Network errors, for example, will need to be communicated back to the calling application.

RAPI functions that fail due to a RAPI-related error will return the error value defined for their Win32-based counterpart. To distinguish between RAPI and non-RAPI errors, use either the **CeRapiGetError** function or the **CeGetLastError** function. To determine if a function failed because of RAPI errors, call **CeRapiGetError**. To determine if a function failed because of non-RAPI errors, call **CeGetLastError**, which works the same as the **GetLastError** function does on Windows-based platforms.

Sample RAPI Application

The following code example shows how to initialize the RAPI client, make calls, and handle errors.

```
#include <stdio.h>
#include <tchar.h>
#include <rapi.h>

void PrintDirectory (LPWSTR lpszPath, UINT Indent)
{
  DWORD dwFoundCount;
  LPCE_FIND_DATA findDataArray;
  WCHAR szSearchPath[MAX_PATH];

  wcscpy (szSearchPath, lpszPath);
  wcscat (szSearchPath, L"*");

  if(!CeFindAllFiles (szSearchPath,
                      FAF_ATTRIBUTES | FAF_NAME,
                      &dwFoundCount,
                      &findDataArray))
  {
    _tprintf (TEXT("*** CeFindAllFiles failed. ***\n"));
    return;
  }

  if (!dwFoundCount)
    return;

  for (UINT i = 0; i < dwFoundCount; i++)
  {
    for (UINT indCount = 0; indCount < Indent; indCount++)
      _tprintf( TEXT("  "));

    if (findDataArray[i].dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
    {
      wprintf (TEXT("[%s]\n"), findDataArray[i].cFileName);

      WCHAR szNewPath[MAX_PATH];

      wcscpy (szNewPath, lpszPath);
      wcscat (szNewPath, findDataArray[i].cFileName);
      wcscat (szNewPath, L"\\");
```

```
        PrintDirectory (szNewPath, Indent + 1);
      }
      else
        wprintf (TEXT("%s\n"), findDataArray[i].cFileName);
    }

    if (findDataArray)
      RapiFreeBuffer (findDataArray);
}

void main()
{
  HRESULT hRapiResult;

  _tprintf (TEXT("Connecting to Windows CE..."));

  hRapiResult = CeRapiInit ();

  if (FAILED(hRapiResult))
  {
    _tprintf (TEXT("Failed\n"));
    return;
  }

  _tprintf (TEXT("Success\n"));

  PrintDirectory (L"\\", 0);

  CeRapiUninit ();
}
```

CHAPTER 8

Managing the Connection Partnership

This section provides information on methods and tools that you can use to manage aspects of a connection partnership between a Microsoft Windows CE–based device and its companion Windows-based desktop computer. This section includes:

- "Receiving Connection Notification," which explains how to launch applications automatically when a device is connected to or disconnected from the desktop computer by using either registry-based or COM interface–based notification.
- "Transferring Files," which describes how to register file types; how to register, implement, and use a file filter; and how to implement a dummy file filter for transferring and converting data.
- "Using the CEUTIL Helper DLL for Windows CE Services," which explains how to use the CEUTIL utility to manage registry entries for Windows CE Services and to gain access to device partnerships.

Receiving Connection Notification

Connection notification is the method by which applications on a desktop computer are notified when a Windows CE–based device is either connected to or disconnected from the desktop computer. To launch applications automatically, an application must be registered in the system registry on the desktop computer for each connection event. Once the application is registered, the Windows CE Services connection manager starts the application whenever the specified event occurs.

The following applications are registered in the system registry on the desktop computer:

- Windows CE Explorer.
- Remote tools (for example, the debugger), which are provided with Windows CE Platform Builder.
- Connection notification client sample code, which provides a basic template for receiving connection notifications. For a description of this sample, see the CD-ROM that accompanies this documentation.

There are two methods to register the desktop application:

- Registry-based notification, by using command lines that are registered in the registry on the desktop computer.
- COM interface–based notification, by using two Component Object Model (COM) interfaces—one that is implemented by the connection manager and the other by the application—to perform the registration.

Registry-based Notification

In registry-based notification, an application places a command line in the desktop system registry in one of two keys:

- **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\AutoStartOnConnect**
- **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\AutoStartOnDisconnect**

When the event specified by the key occurs, the command line is executed. That is, when a Windows CE–based device is connected to a desktop computer, the command line under **AutoStartOnConnect** is executed; likewise, when the device is disconnected, the command line under **AutoStartOnDisconnect** is executed.

Registry-based notification is appropriate for applications that need neither some control of the connection manager nor the ability to register and deregister for connection notifications.

▶ **To register an application for automatic execution**

1. Construct a named value that uniquely identifies the application.

   It should include a company and product name—for example, **MicrosoftHPCExplorerAutoConnect**. Enter the named value under the appropriate key, either **AutoStartOnConnect** or **AutoStartOnDisconnect**.

2. Define the named value as the application that is to be executed and include command line arguments. The application file path must be wrapped with double quotes if arguments are provided.

The following registry editor (.reg) file shows how to register a command line for both **AutoStartOnConnect** and **AutoStartOnDisconnect**. In this example, when the device is connected, Notepad.exe is started with a command line argument of **c:\config.sys**. When the device is disconnected, Notepad.exe is started with a command line argument of **c:\autoexec.bat**.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE
Services\AutoStartOnConnect]
"MicrosoftAutoConnectSample"="\"notepad"\ c:\\config.sys"
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE
Services\AutoStartOnDisconnect]
"MicrosoftAutoDisconnectSample"="\"notepad"\ c:\\autoexec.bat"
```

COM Interface–based Notification

The second method you can use to register a desktop application is COM interface–based notification. In this method, you use two COM interfaces to register an application in the desktop registry:

- **IDccMan**, which is implemented by the connection manager.
- **IDccManSink**, which is implemented by the application.

Although the COM interface–based notification method is more complex than registry-based notification, by using it you get some control of the connection manager and the ability to register and deregister for connection notifications.

The connection manager, which resides on the desktop, displays an icon next to the clock in the taskbar when the Windows CE–based device is connected or is waiting to be connected to the desktop computer. This icon—two terminals with a connecting cable—indicates the connection status. By right-clicking the icon, you can start Windows CE Explorer.

Notifying and Deregistering Procedures

The notification process follows the same basic steps for connection and disconnection of the Windows CE–based device. The following procedure assumes that the **IDccMan** and **IDccManSink** interfaces have been implemented.

▶ **To register an application by using COM interface–based notification**

1. Initialize the COM library and register the application for the appropriate event.
2. Connect or disconnect the device.
3. Perform the application processing.

The Connection Notification Client sample application, provided on the CD-ROM that accompanies this documentation, shows several connection notification scenarios, including a new remote connect, a disconnect, and a reconnect. To see the actual sequence of interface method calls for any of these scenarios, build and run the application. Then, in the **Connection Notification Test** dialog box, view the **Notification Messages** list.

▶ **To receive notification upon connection of the Windows CE–based device to a desktop computer**

1. Initialize the COM library and register with the connection manager.
   - Call the COM function **CoInitialize** to initialize the Component Object library.
   - Call the COM function **CoCreateInstance** with the DccMan class identifier (CLSID_DccMan) and **IDccMan** interface identifier (IID_IDccMan), and receive a pointer to the **IDccMan** interface.

     For more information on CLSID_DccMan and IID_IDccMan, see "Registering the **IDccMan** Class Identifiers."
   - Call the **IDccMan::Advise** method, which provides the connection manager with a pointer to the **IDccManSink** interface that you implemented and registers the application with the connection manager.

     The connection manager calls the **IDccManSink::OnLogInactive** method, notifying the application that there is no connection between the desktop computer and the device.

2. Establish the connection between the desktop computer and the device.
   - The connection manager calls the **IDccManSink::OnLogListen** method.

     The connection manager waits for the remote connection services for both the desktop computer and the device to respond. Until they are both running, the connection manager will not proceed.
   - For Windows 95–based systems only, the connection manager calls the **IDccManSink::OnLogAnswered** method when the connection manager has detected the communications interface.
   - The connection manager calls the **IDccManSink::OnLogActive** method when the connection is established between the device and connection manager.
   - The connection manager calls the **IDccManSink::OnLogIpAddr** method, providing the Internet Protocol (IP) address that it obtained for the communications socket.

   Note When the **IDccManSink::OnLogIpAddr** notification occurs, the connection is completely established.

3. Perform your processing in the application.

   This can include processing on the desktop computer, remote processing on the device by using remote application programming interface (RAPI), or calling the **IDccMan** methods. However, the application should wait to use **CeRapiInit** to initialize RAPI until the **IDccManSink::OnLogActive** notification is received. This ensures that a connection is established between the desktop computer and the device.

▶ **To receive notification upon disconnection of the Windows CE–based device from the desktop computer**

1. Initialize the COM library and register the application.
2. Disconnect the device from the desktop computer.

   Windows CE Services notifies the application when the desktop computer and device are disconnected by calling the **IDccManSink::OnLogDisconnection** method.
3. Perform your processing in the application.

   Because there is no connection to the device, this processing can only take place on the desktop computer.

▶ **To receive notification when reestablishing a remote connection**

The Connection Notification Client source code uses the **IDccMan** interface and implements the **IDccManSink** interface.

- If a connection was established, but then was disconnected by the desktop computer or the device, the **IDccManSink::OnLogActive** notification occurs when the connection is reestablished.

  When an application calls the **IDccMan::ShowCommSettings** function and the **OK** button is clicked in the **Communications Properties** dialog box, the following notification sequence occurs:

  - IDccManSink::OnLogListen
  - IDccManSink::OnLogDisconnection
  - IDccManSink::OnLogInactive
  - IDccManSink::OnLogListen

  If the **Cancel** button is clicked instead, no notification is sent and a Listen state is maintained.

▶ **To deregister an application from being notified**

One of the advantages to using the COM interface–based notification process is that it allows an application to deregister itself from being notified. This might be helpful when an application needs to run only once.

1. Call **IDccMan::Unadvise**, which releases the memory associated with the **IDccManSink** interface.
2. Call **IDccMan::Release**, which releases the **IDccMan** object.
3. Call **CoUninitialize** to perform any OLE cleanup.

   Note that a call to **CoUninitialize** is required for each successful call to **CoInitialize**.

Registering the IDccMan Class Identifiers

Both the DccMan class identifier, CLSID_DccMan, and the **IDccMan** interface identifier, IID_IDccMan, are passed in the call to **CoCreateInstance**. Because the Windows CE Services Setup application registers CLSID_DccMan, you only need to register IID_IDccMan in your application.

The following code example shows how to initialize the IDccMan interface in the registry.

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\499c0c20-A766-11cf-8011-00A0c90A8F78]
@="Connection Manager"

[HKEY_CLASSES_ROOT\CLSID\499c0c20-A766-11cf-8011-   /
 00A0c90A8F78\InprocServer32]
@="C:\\Windows\\System\\Rapi.dll"
   "ThreadingModel"="Apartment"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\499c0c20-A766-11cf-8011-   /
 00A0c90A8F78]
@="Connection Manager"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\499c0c20-A766-11cf-8011-   /
 00A0c90A8F78\InprocServer32]
@="C:\\Windows\\System\\Rapi.dll"
   "ThreadingModel" = "Apartment"
```

Windows CE–based Device Notification

In addition to supporting notification for the desktop computer–based application, Windows CE supports Notification API, which allows applications on the mobile device to receive connection notification.

The **CeRunAppAtEvent** function provides Windows CE–based applications with the ability to be notified when a connection or other event occurs. When a specified event occurs, applications registered by **CeRunAppAtEvent** are launched. For example, events include when a device is connected to a desktop computer, when an operating system is restored from a backup, or when the system time is changed.

The function is prototyped as:

```
BOOL CeRunAppAtEvent (TCHAR *pwszAppName, LONG lWhichEvent);
```

The name of the application to be launched when the event occurs is indicated by the first parameter. The second parameter is a set of bit flags that indicate which events you want to track.

Transferring Files

A file filter is a dynamic-link library (DLL) that controls the transfer of data between the desktop computer and the Windows CE–based device. File filters are used by the Windows CE Services application on the desktop computer to automatically convert files as they are transferred.

File formats used by the Windows CE operating system and Windows CE–based applications are generally different from those of the corresponding Windows-based applications. For example, Microsoft Pocket Word does not support OLE compound files. Windows CE Services automatically adjusts file formats as files are transferred between the desktop computer and the device.

Windows CE Services includes the following filters:

- Pocket Word (.pwd) to Microsoft Word (.doc)
- Word (.doc) to Pocket Word (.pwd)
- Microsoft Pocket Excel (.pxl) to Microsoft Excel 5.0 (.xls)
- Excel (.xls) to Pocket Excel (.pxl)
- Windows bitmap (.bmp) to Windows CE 4-color bitmap (.2bp)

You can extend the file-filtering capability of Windows CE Services by defining your own application-specific filters. This section describes file filters and the interfaces that you use to create them.

Implementing a file filter is similar for both importing and exporting files. The only differences are in the registry settings and in how the body of the file filter—the converter function—changes data. The examples in this section demonstrate the procedure for importing files, but typically you would write a converter function that handles both importing and exporting, by using dual registry settings that indicate both the import and export functionality.

Note The words "importing" and "exporting" in this section are from the perspective of the device. Thus, importing a file with a file filter transfers a file from the desktop computer to the device, whereas exporting a file with a file filter transfers a file from the device to the desktop computer.

Registering File Types and File Filters

Windows CE Services uses the registry entries to determine which conversions are available for a given file type and how to invoke the filter that supports the conversion. For this reason, you must register each file type and file filter properly by using the following procedure.

▶ **To register file types and their filters**

1. Register the file extension type.
2. Generate a class identifier (CLSID) for the file filter.
3. Register the file filter.

The following sections describe each step in detail and provide a sample file filter registry entry.

Note CEUTIL, a utility DLL, has functions that are especially helpful when you are dealing with the desktop registry entries for Windows CE Services. For information about CEUTIL, see "Using the CEUTIL Helper DLL for Windows CE Services."

Registering a File Extension Type

Windows CE Explorer, like Windows Explorer, allows you to customize a type name, as displayed in the details view of Windows Explorer, and an icon for any file extension (for example, .pwd). You must register file filters under **HKEY_CLASSES_ROOT**.

The following is the structure of **HKEY_CLASSES_ROOT**.

```
HKEY_CLASSES_ROOT\.<file extension>
   \(Default) = <class name>
HKEY_CLASSES_ROOT\<class name>
   \(Default) = <name to be displayed in the "Type" column of Explorer>
   \DefaultIcon = <file name or index of the icon for this type>
```

Generating a Class Identifier

You must give every file filter a unique class identifier (CLSID), which identifies class objects to OLE. CLSIDs are universally unique identifiers (UUIDs), also called globally unique identifiers (GUIDs). You must include the CLSID for the file filter in your application and you must register it with the operating system when your application is installed.

If the file filter supports both importing and exporting, a unique CLSID must be associated with each file filter that is to be registered for the respective import and export registry setting.

The GUID Generator tool lets you generate a GUID that you can use to identify your file filter. A GUID Generator application, named Guidgen.exe, is provided with Microsoft Visual C++ development system. The GUID Generator calls the **CoCreateGuid** function to generate a new GUID. It also lets you copy the GUID to the clipboard so that you can insert the GUID into the source code for your application by using one of the following formats:

- IMPLEMENT_OLECREATE macro format

  Defined in an IMPLEMENT_OLECREATE macro, which allows instances of a *CCmdTarget*-derived class to be created by Automation clients. For example:

```
// {CA761230-ED42-11CE-BACD-00AA0057B223}
IMPLEMENT_OLECREATE(<<class>>, <<external_name>>,
0xca761230, 0xed42, 0x11ce, 0xba, 0xcd, 0x0, 0xaa,
0x0, 0x57, 0xb2, 0x23);
```

- DEFINE_GUID macro format

  Defined in an IMPLEMENT_OLECREATE macro, which is included with Visual C++ in the file Afxdisip.h. It allows instances of a *CCmdTarget*-derived class to be created by Automation clients. For example:

```
// {CA761230-ED42-11CE-BACD-00AA0057B223}
DEFINE_GUID(<<name>>,
0xca761230, 0xed42, 0x11ce, 0xba, 0xcd, 0x0, 0xaa, 0x0, 0x57, 0xb2,
0x23);
```

- Statically allocated structure format

  Declared as a statically allocated structure. For example:

```
// {CA761232-ED42-11CE-BACD-00AA0057B223}
static const GUID <<name>> = { 0xca761232, 0xed42, 0x11ce,
{ 0xba, 0xcd, 0x0, 0xaa, 0x0, 0x57, 0xb2, 0x23 } };
```

- Registry entry

  Specified in a form suitable for registry entries or registry editor scripts. For example:

```
{CA761233-ED42-11CE-BACD-00AA0057B223}
```

Registering a File Filter

An application registers a file filter by placing its CLSID in the following registry key locations:

- **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\Filters\.**<*file extension*>**\InstalledFilters**. This registration associates the file filter with the file type that it converts.
- **HKEY_CLASSES_ROOT\CLSID**. This registration provides information on the file filter's capabilities and its DLL.

Registering a file filter in two locations guarantees that your file filter is available for any new *device partnership*. Windows CE Services uses device partnerships to allow multiple Windows CE–based devices to be connected to the same desktop computer. If device partnerships exist when an application registers a file filter, an application must create a file filter extension key for each existing device partnership. Connecting a device to a desktop computer for the first time establishes a new device partnership with a unique identifier. Devices use the partnership identifier to store unique settings for synchronization, file conversions, and backup-and-restore information.

All filters registered under the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\Filters** key are copied to the **HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows CE Services\Partners**<*partner identifier*> key, where <*partner identifier*> is the identifier of the new device partnership.

A file type can have more than one file filter associated with it. The Windows CE Services user interface (UI) lists the registered filters as filter options. A file filter can be registered under the file type's **InstalledFilters** key in the **DefaultImport** or **DefaultExport** subkeys. As their names imply, these subkeys define the default file filters for the file type. The file filter specified under the extension key as the **DefaultImport** or **DefaultExport** value will be shown as the default.

Note Any filter defined as the **DefaultImport** or **DefaultExport** value must be an **InstalledFilters** value, also.

A file filter can also be registered for a particular device. For example, for the Palm-size PC device category, the file filter extension subkey needs to be created under the **HKEY_LOCAL_MACHINE\SOFTWARE\Windows CE Services\SpecialDefaults\Palm PC\Filters** key.

The following is the structure of the **Filters** key and the **InstalledFilters** subkey.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\Filters

   \.<file extension>
      [DefaultImport = <default import filter CLSID>]
      [DefaultExport = <default export filter CLSID>]
      \InstalledFilters
         [<clsid1>]
         . . .
         [More CLSIDs for this extension.]
    . . .
    [More extensions.]
```

The **HKEY_CLASSES_ROOT\CLSID** key provides basic information about file filters. Each file filter that has been identified in the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\Filters** key must be registered in this key. The following is the structure for this key and its subkeys.

```
HKEY_CLASSES_ROOT\CLSID

   \<clsid>
      \(Default) = <description in "Edit Conversion Settings" list>
      \DefaultIcon = <file name,index for the icon for this type>
      \InProcServer32 = <file name of the DLL that handles this type>
         ThreadingModel = Apartment
      \PegasusFilter
         [Import]
         [HasOptions]
         Description = <string to display in the conversion dialog box>
         NewExtension = <extension of the converted file>
   \. . . [More CLSIDs for filters.]
```

The *<clsid>* key is a named value that is the CLSID of the registered file filter. This key contains the following subkeys:

- **DefaultIcon**, which defines the icon name string or icon resource identifier for the icon associated with the file filter DLL.
- **InProcServer32**, which identifies the file filter DLL using the default value, and defines the apartment model capabilities of the file filter in the **ThreadingModel** named value.
- **PegasusFilter**, which provides information on the specific capabilities of the file filer.

The following table shows possible named values for the **PegasusFilter** subkey.

Named value	Description
Import	If this named value exists, the conversion type is for importing files from the desktop computer to the device. Otherwise, the conversion type is for exporting files from the desktop computer to the device.
HasOptions	If this named value exists, the file filter supports the **ICeFileFilter::FilterOptions** method.
Description	The data for this named value is a string that describes the conversion. Windows CE Services displays this text on the property sheets that are displayed by selecting the **Device→Desktop** or **Desktop→Device** tab control selections in the **File Conversion Properties** dialog box, and then choosing **Edit** to display the **Edit Conversion Settings** dialog box. For example, if the **Import** named value exists, then, on the **Desktop→Device** property sheet, the data value defined by the **Description** named value is displayed under the file conversions details **Convert to HPC files of the type**.
NewExtension	Defines the extension of the file that will be created on the destination device.

Sample File Filter Registry Entry

The following is a sample registry editor (.reg) file that is used to register the Bitmap Image file filter converter. This converter is to be used when a bitmap file is imported from a desktop computer to a Windows CE–based device. The sample file can be used to convert a bitmap file with a .bmp format to a bitmap file with the .2bp format used by Windows CE. The last three entries register the .2bp file extension to be displayed with a specific icon and name.

Note The 2bp.dll file converter is registered and installed when Windows CE Services is installed on the desktop computer.

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\Filters\.bmp]
"DefaultImport"="{DA01ED80-97E8-11cf-8011-00A0C90A8F78}"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE
Services\Filters\.bmp\InstalledFilters]
"{DA01ED80-97E8-11cf-8011-00A0C90A8F78}"=""

[HKEY_CLASSES_ROOT\CLSID\{DA01ED80-97E8-11cf-8011-00A0C90A8F78}]
@="Bitmap Image"
```

```
[HKEY_CLASSES_ROOT\CLSID\{DA01ED80-97E8-11cf-8011-
00A0C90A8F78}\DefaultIcon]
@="c:\Program Files\Windows CE Services\2bp.dll,-1000"

[HKEY_CLASSES_ROOT\CLSID\{DA01ED80-97E8-11cf-8011-
00A0C90A8F78}\InProcServer32]
@="2bp.dll"
"ThreadingModel"="Apartment"

[HKEY_CLASSES_ROOT\CLSID\{DA01ED80-97E8-11cf-8011-
00A0C90A8F78}\PegasusFilter]
"Import"=""
"Description"="Bitmap Image."
"NewExtension"="2bp"

[HKEY_CLASSES_ROOT\.2bp]
@="2bpfile"

[HKEY_CLASSES_ROOT\2bpfile]
@="Bitmap Image"

[HKEY_CLASSES_ROOT\2bpfile\DefaultIcon]
@="c:\Program Files\Windows CE Services\minshell.dll,-2025"
```

Implementing and Using a File Filter

The CD-ROM that accompanies this documentation includes a sample file filter named Copyfilt that imports a binary file (.bin) from a desktop computer to a binary file (.pbn) on a Windows CE–based device. The Copyfilt sample file filter demonstrates basic operations for implementing a file filter, which are described in the following procedure.

▶ **To implement a file filter**

1. Register the file filter DLL.

   For a description of how to register a file filter, see "Registering a File Filter."

2. Implement the **ICeFileFilter** interface and methods.

3. Windows CE Services calls the **QueryInterface** method for the file filter's **ICeFileFilterOptions** interface.
   - If this interface is available, Windows CE then calls the **ICeFileFilterOptions::SetFilterOptions** method with a correctly initialized **CFF_CONVERTOPTIONS** structure. The *bNoModalUI* member specifies whether the converter is allowed to bring up modal UI while performing the conversion.

     For a file filter that includes selectable conversion options, implement the **ICeFileFilter::FilterOptions** method to allow users to select among the conversion options supported by the file filter.

▶ **To use a file filter**

1. The user transfers a file by dragging it between Windows Explorer on the desktop computer and Windows CE Exploreron the device.
2. Windows CE Services prompts the user for a conversion type, using the **File Conversion Properties** dialog box.
3. Windows CE Services calls the **ICeFileFilter::NextConvertFile** method for the file filter to perform the custom file conversion.
4. Information about the file conversion and about the source and destination files is passed by pointers to the **CFF_CONVERTINFO**, **CFF_DESTINATIONFILE**, and **CFF_SOURCEFILE** structures.

Within the **ICeFileFilter::NextConvertFile** method:

1. Call **ICeFileFilterSite::OpenSourceFile** to open the source file.
2. Call **ICeFileFilterSite::OpenDestinationFile** to open the destination file.
3. Use the **OpenSourceFile** method to read data from the stream file that was opened.
4. Convert the data.

   This can include independent software vendor (ISV)-developed code and RAPI calls.
5. Check the status of the **NextConvertFile pbCancel** parameter occasionally to ensure that the user has not stopped the conversion process.

   If the conversion has been stopped, perform all cleanup operations, and then exit.
6. Use the **OpenDestinationFile** method to write the converted data to the stream file that was opened.

7. Call the **ICeFileFilterSite::ReportProgress** method occasionally to report the progress of the file conversion.

   Windows CE Services uses this information to update a status bar that shows the percentage of the conversion that is complete. You should limit your use of this method because it can add substantially to the conversion time.

8. Call the **ICeFileFilterSite::ReportLoss** method to report data that is intentionally discarded during conversion.

   Windows CE Services displays a message with this information when the file conversion is complete. Depending on the error format passed in the call, Windows CE Services may call the **ICeFileFilter::FormatMessage** method for the file filter, in order to properly format the message.

9. Use the **ICeFileFilterSite::CloseSourceFile** method to close the source file, and then use the **ICeFileFilterSite::CloseDestinationFile** method to close the destination file.

Using RAPI Calls in a File Filter

You can use RAPI calls in a file filter. This allows use of any RAPI functions that are appropriate to your application, such as registry or file functions.

Do not initialize RAPI in the file filter DLL by using **CeRapiInit**. Rather, the **NextConvertFile** method should have already performed the RAPI initialization and established a connection between the desktop computer and the Windows CE–based device. If a RAPI call fails because there is no connection established, the file converter should perform some type of default action rather than just failing. For example, this default action could involve querying the user to select from various options.

To determine if a call failed due to a failure in the RAPI, use **CeRapiGetError**. To diagnose non-RAPI related errors, use **CeGetLastError**.

For more information on RAPI, see "Working with RAPI."

Implementing a Dummy File Filter

A dummy file filter gives the appearance that files are being converted without actually implementing a file filter or performing a filter conversion. Instead, the file is passed without any conversion.

You may want to implement a dummy file filter for a file that has a unique file type or one that has not been registered already in the desktop registry. A dummy file filter can also be useful for a file that does not need any conversion when it is transferred between Windows Explorer on the desktop computer and Windows CE Explorer for the Windows CE–based device.

Usually, if a file with a deregistered file type is copied to the device, the device displays the warning "No Converter Selected." This warns the user that the file will be transferred without conversion. In this situation, you could implement a dummy file filter to avoid alarming the user with the file conversion warning.

Note The "No Converter Selected" warning is displayed only if the **File Conversions Properties** for the device is set to enable file conversion. If the **Enable File Conversion** check box is cleared, the "No Converter Selected" warning is not displayed.

▶ **To register a dummy file filter**

1. In the desktop registry, modify the **HKLM\Software\Microsoft\Windows CE Services\Filters** key by adding a subkey.

   This subkey should name the file extension for the type of files that should be converted with the NULL file conversion. For example, if you are converting files with extension .abc, then you must add an **.abc** subkey.

2. Under the **.abc** subkey, create a string value named **DefaultImport** that is set to **Binary Copy**.

   This string value identifies the conversion for files with .abc extensions that are imported from the desktop computer to the device.

3. Under the **.abc** subkey, create a string value named **DefaultExport** that is set to **Binary Copy**.

   This string value identifies the conversion for files with .abc extensions that are exported from the device to the desktop computer.

The following registry editor (.reg) file shows how to register the example .abc dummy file filter.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\Filters\.abc]
"DefaultImport"="Binary Copy"
"DefaultExport"="Binary Copy"
```

In this example, when a file with an .abc extension is copied between the desktop computer and the device, it will seem as though a conversion process is taking place because you do not receive the "No Converter Selected" warning from Windows CE Services. However, no filter actually is being used, because an **InstalledFilters** subkey has not been added under the **.abc** key.

Using the CEUTIL Helper DLL for Windows CE Services

You can use the CEUTIL DLL to manage desktop registry entries for Windows CE Services. CEUTIL encapsulates the registry top-level locations to ensure forward-compatibility for applications. It also provides helper functions for browsing device partnerships and querying the currently connected, or selected, device settings. In general, this DLL is a replacement for and compatible with the Microsoft Win32 registry API that is used when referring to any subkeys under the Windows CE Services root.

Use CEUTIL to do the following tasks:

- Register desktop file filters
- Register desktop synchronization services
- Access device partnership settings that are used for both file filters and synchronization services
- Add custom menu items

Desktop Registry Structure

The following list describes the desktop registry structure that is used by Windows CE Services and the corresponding identifiers that are used in CEUTIL to refer to particular keys in the structure:

- **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows CE Services**, hereafter referred to as *machine_root*, stores general information.
- **HKEY_CURRENT_USER\Software\Microsoft\Windows CE Services**, hereafter referred to as *local_root*, stores device partnership information.

The first time a Windows CE–based device is connected to a desktop computer and a device partnership is created, the various synchronization and filter settings are copied from the *machine_root* to the partnership subkey under *local_root*.

Examples of CEUTIL Functions

The following code example shows how to enumerate device partnerships and get the path for the file synchronization folder.

```
#include "ceutil.h"

void Example1 (void)
{
  // Note that this code runs on the desktop (or host) computer,
  // and not on the Windows CE-based device, so set
  // your project settings accordingly.

  HCESVC   hsvc            = NULL;
  HCESVC   hsvcSync        = NULL;
  HCESVC   hsvcProfile     = NULL;
  DWORD    cProfilesEnum   = 0;
  DWORD    nProfileID      = 0;

  while (SUCCEEDED (CeSvcEnumProfiles (&hsvc,
                                       cProfilesEnum,
                                       &nProfileID)))
  {
    if (nProfileID != (DWORD)-1)
    {
      if (SUCCEEDED (CeSvcOpenEx (hsvcProfile,
                                  TEXT("Services\\Synchronization"),
                                  FALSE, &hsvcSync)))
      {
        TCHAR szPath[MAX_PATH];

        if (SUCCEEDED (CeSvcGetString (hsvcSync,
                                       TEXT("Briefcase Path"),
                                       szPath,
                                       sizeof(szPath)/sizeof(TCHAR))))
        {
          // Found a path to use.

          // Your code to complete tasks goes here.
        }

        CeSvcClose (hsvcSync);
      }
```

```
      CeSvcClose (hsvcProfile);
    }
  }

  cProfilesEnum++;
}
```

The following code example shows how to add a custom menu.

```
{
  HCESVC hsvcMyMenu = NULL;

  if (SUCCEEDED (CeSvcOpen (CESVC_CUSTOM_MENUS,
                            TEXT("MyApp"), TRUE, &hsvcMyMenu)))
  {
    CeSvcSetString (hsvcMyMenu, TEXT("DisplayName"),
                    TEXT("&My Calculator"));

    CeSvcSetString (hsvcMyMenu, TEXT("Command"), TEXT("calc.exe"));

    CeSvcSetString (hsvcMyMenu, TEXT("StatusHelp"),
                    TEXT("Displays calculator"));

    CeSvcSetDword (hsvcMyMenu, TEXT("Version"), 0x00020000);

    CeSvcClose (hsvcMyMenu);
  }
}
```

CHAPTER 9

Synchronizing Data

Microsoft Windows CE Services includes Microsoft® ActiveSync™, which allows you to synchronize, or coordinate, data stored on a Windows-based desktop computer with data stored on a Windows CE–based device. ActiveSync also enables you to coordinate device data with two or more computers or synchronize information on one computer with several devices.

Note ActiveSync does not support synchronization between two devices, a device and a server, or a device and mounted volumes.

Through serial, infrared, Ethernet LAN, or modem connections, the synchronization process automatically transfers and tracks data changes, including additions, deletions, and other revisions to information. As a result, shared information is always up to date and accurate. ActiveSync is a client/server architecture that consists of a *service manager* (the server) and a *service provider* (the client):

- The service manager, which is built into Windows CE Services, is a synchronization engine that resides on both the desktop and the device. The service manager performs many common synchronization tasks, which include providing connectivity, detecting changes in data, resolving data conflicts, as well as mapping and transferring data objects.
- The service provider comprises two modules, which typically are dynamic-link libraries (DLLs), that you must implement in your application to perform the synchronization tasks that are specific to your data. One module, called the desktop provider module, resides on the desktop and the other module, called the device provider module, resides on the device.

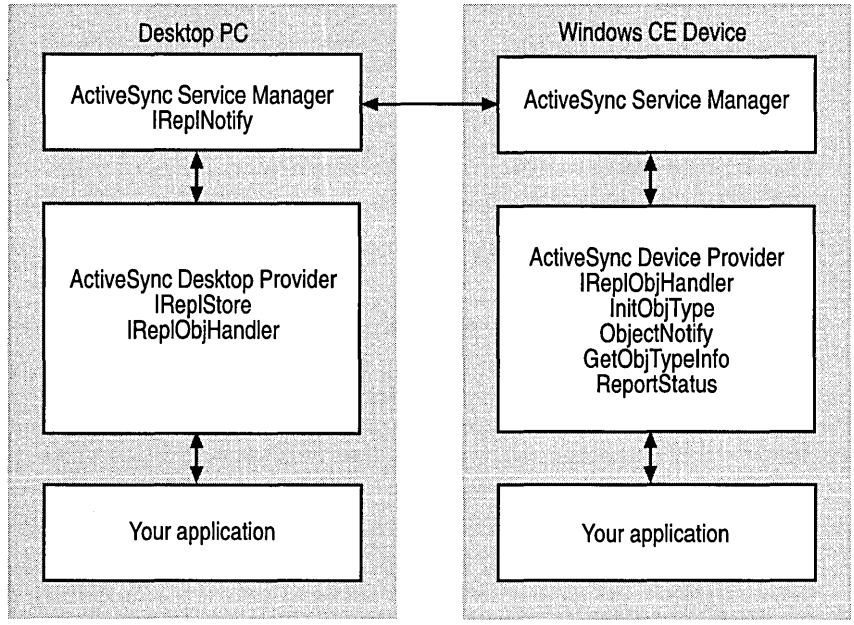
The service provider interfaces with both the service manager and the application, thereby exposing the application and the application's data to the service manager. In turn, this interaction enables the service manager to synchronize different types of data from different types of applications. The service provider also facilitates all requests made by the service manager, such as displaying a user interface (UI) or reporting status.

Creating a service provider is a two-part process:

1. Create the service provider by developing the desktop provider module and the device provider module.
2. Register the service provider on both the Windows-based computer and the Windows CE–based device.

The service manager implements a single interface, **IReplNotify**, which is built into ActiveSync and Windows CE Services. Thus, once you register the service provider, the synchronization process is automatic. The service manager, with the help of the service provider, directs and controls all synchronization tasks.

The following illustration shows how the service manager interfaces with the service provider to access data.



Creating an ActiveSync Service Provider

The synchronization process transfers only data that has changed. For this reason, the service provider must track data changes as efficiently as possible to ensure that the least amount of data is transferred during synchronization.

To create an efficient ActiveSync service provider for an application, you first must analyze the data to be synchronized by defining the *object*, the *object type*, and the *object identifier*, as well as the folder and the store to hold the objects. You also must decide on a method to compare objects and a method to indicate that an object has changed. As you do this, keep in mind the following terminology and requirements, which are specific to the synchronization process:

- An object is a logical unit of data, such as a single appointment. The object definition depends on your application. The definition can be an appointment, an address, or some other item.
- An object type is a name for a particular group of objects contained in a folder. For example, appointment is an object type naming all appointments in a Microsoft Schedule+ folder.
- An object identifier is a value that uniquely identifies an object. The object identifier must satisfy the following criteria:
  - It cannot change once it is created.
  - It cannot be reused for any other object.
  - It must be ordered, so that you can determine which object comes first.
  - It must signify changes since the last synchronization.

  For example, globally unique identifiers (GUIDs) satisfy the criteria. The criteria allow the service manager to compare object identifiers to aid in synchronization.
- The store is a database that holds the data, or objects, to be synchronized. The store must accommodate the objects, but its format can vary. The store can be a flat file, a database, or some other custom format.

Once you have analyzed the data, you are ready to create a service provider by developing the *desktop provider module* and the *device provider module*:

- The desktop provider module handles the bulk of communication with the service manager and implements two Component Object Model (COM) interfaces: **IReplStore** and **IReplObjHandler**:

COM interface	Description
IReplStore	Enumerates objects in a store, checks for changes in an object, and displays a UI so that the user can set synchronization options and resolve conflicts.
IReplObjHandler	Serializes data and deletes objects.

- The device provider module must implement the same **IReplObjHandler** interface as the desktop provider module, as well as the following functions:

Function	Description
InitObjType	Initializes data and returns a pointer to the **IReplObjHandler** interface when the device provider module loads and, when the device provider module terminates, frees allocated resources.
ObjectNotify	Calls the service manager when an object in the object store for a device is changed or deleted.
GetObjNotify	Returns the type of the specified object.
ReportStatus	Optional; gets status for synchronization objects.

Developing the Desktop Provider Module

In contrast to the device provider, which has limited functionality, the desktop provider handles the bulk of all communication with the manager. The desktop provider, hereafter referred to simply as the provider, implements two COM interfaces: **IReplStore** and **IReplObjHandler**. **IReplStore** enumerates objects in a store, checks for changes in an object, and displays a user interface so that the user can set synchronization options and resolve conflicts. **IReplObjHandler** serializes data and deletes objects. For more information on the methods contained in **IReplStore** and **IReplObjHandler**, see the *Windows CE Programmer Reference.*

▶ **To implement the desktop provider module**

1. Create one GUID for the store, by using the Microsoft Visual C++® GUID generator tool, Guidgen.exe.
2. Initialize the store.
3. Compare store identifiers and handle mismapped store identifiers.

4. Define object handles to provide access to data objects stored on a device.

   The object handles typically are pointers to data structures and include HREPLITEM, HREPLFLD, and HREPLOBJ. HREPLOBJ is a generic handle to either an item or a folder.
5. Provide a folder handle for the specified object type and, to access folders, return a pointer to the **IReplObjHandler** interface.
6. Enumerate objects for a specified object type.
7. Detect changes in object types.
8. Send and receive objects.
9. Create a UI to set synchronization options for a device and handle conflicts.

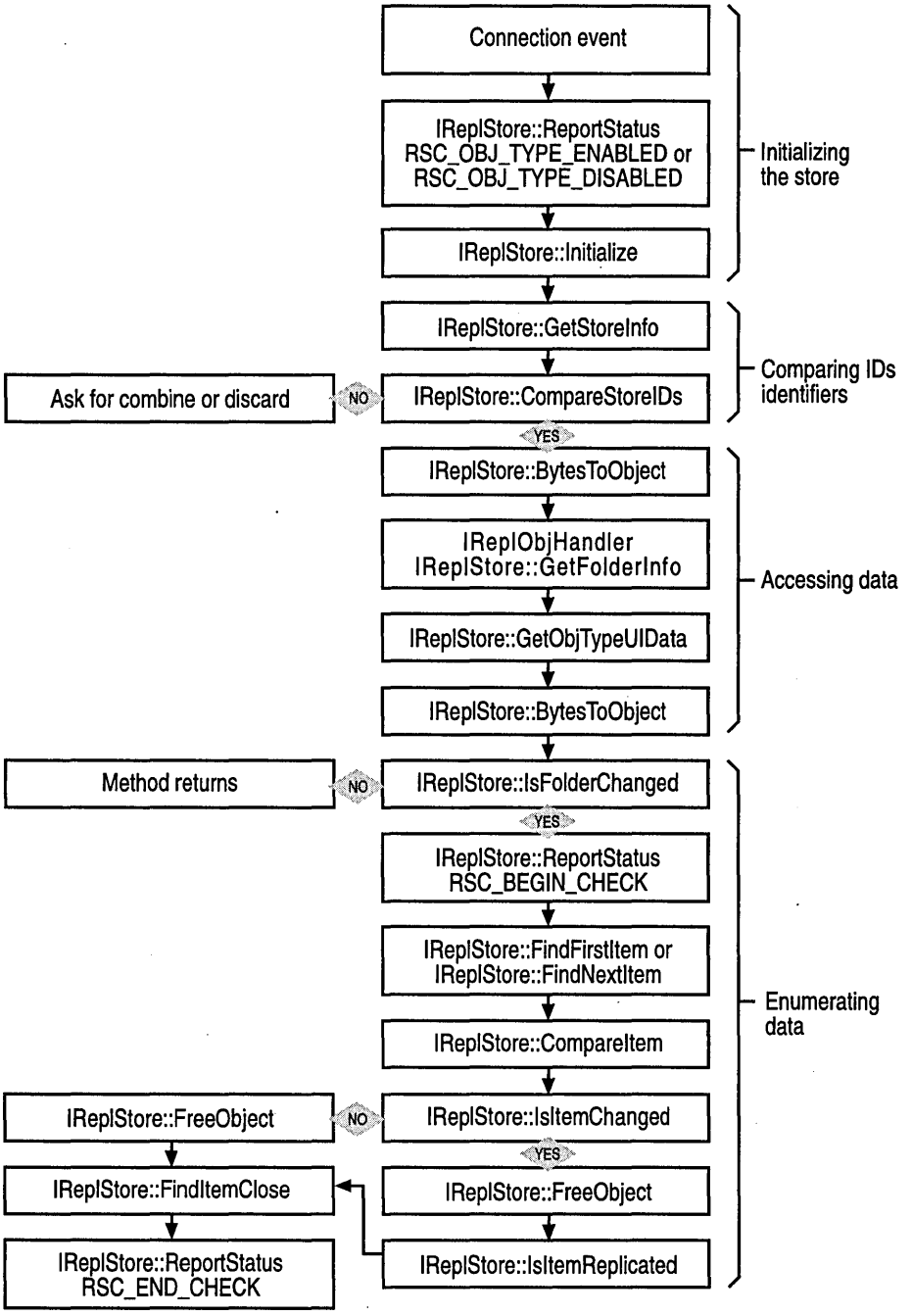
Initializing the Store

When a connection event occurs, the service manager calls **IReplStore::Initialize** to ensure that the data file for the application exists and is open. The service manager:

- Initializes the data store and compares store identifiers to determine if the desktop object and the device object are mapped correctly.
- Accesses the data store and enumerates each desktop object.
- Determines what data has changed and transfers that data to the device.

To accomplish these tasks, the service manager places a series of calls to the desktop provider module, prompting the desktop provider module to respond with the necessary information.

The following illustration shows the order in which the service manager calls the desktop provider module.



In order for the service manager to initialize the desktop provider module, you must store the name of the file that needs to be synchronized in the registry. The registry location for this file name depends on whether you are initializing the desktop provider module for a connected device or a selected device.

A selected device is a disconnected device that a user selects from the list of profiles stored in the Windows CE Services Mobile Device folder. A user selects a profile to change synchronization options. When a user selects a device, the service manager passes the bit flag ISF_SELECTED_DEVICE into **IReplStore::Initialize**.

To access the profile information for a device:

- To get the registry key for a connected device profile, call **IReplNotify::QueryDevice** with QDC_CON_DEVICE_KEY.
- To get the registry key for a selected device profile, call **IReplNotify::QueryDevice** with QDC_SEL_DEVICE_KEY.

If a device is remotely connected to the desktop, the service manager passes the ISF_REMOTE_CONNECTED bit flag into **IReplStore::Initialize**. Whenever this flag is set, the desktop provider module should not display any blocking UI, such as a message box or a dialog box, because the user may not be able to respond to the UI. Instead, the desktop provider module should accept all default actions, without prompting the user on the desktop.

If the service provider returns an error in **IReplStore::Initialize**, the service manager automatically displays an error message. If you want the desktop provider module rather than the service manager to display an error message that you create, return RERR_NO_ERR_PROMPT.

The service manager typically calls **IReplStore::Initialize** immediately after a connection event occurs. However, the service manager can call other methods first, such as when a user changes the synchronization options for a disconnected device.

The following table shows the methods that can be called before **IReplStore::Initialize**.

IReplStore method	Description
IReplStore::GetStoreInfo	Gets information about a store, which is displayed in the ActiveSync option dialog box.
IReplStore::GetObjTypeUIData	Gets information about an object type, which is displayed in the ActiveSync option dialog box under **Status**.
IReplStore::GetFolderInfo	Gives the folder handle, HREPLFLD, to the desktop provider module.
IReplStore::ActivateDialog	Allows a user to change options for an ActiveSync service.
IReplStore::BytesToObject	Converts a series of bytes to a HREPLITEM handle or a HREPLFLD handle.
IReplStore::ObjectToBytes	Converts a HREPLITEM handle or a HREPLFLD handle to a series of bytes.
IReplStore::ReportStatus	Informs the desktop provider module about events that are taking place. This method is optional.

The following code example shows how to implement **IReplStore::Initialize**.

```
STDMETHODIMP CStore::Initialize
(
    IReplNotify *pNotify,    // Pointer to the IReplNotify interface
    UINT uFlags              // Either ISF_SELECTED_DEVICE or
                             // ISF_REMOTE_CONNECTED
)
{
    m_pNotify = pNotify;
    m_uFlags = uFlags;

    // Get the correct registry for the synchronization options.
    HKEY hKey;
    HRESULT hr;
    hr = m_pNotify->QueryDevice( ( uFlags & ISF_SELECTED_DEVICE )?
            QDC_SEL_DEVICE_KEY : QDC_CON_DEVICE_KEY, (LPVOID *)&hKey );

 // Read the registry for the type of files to synchronize.
 // ...
```

```
    // The service provider should suppress all UI that requires
    // user inputif ISF_REMOTE_CONNECTED is set in uFlags.
    // ...

    return NOERROR;
}
```

CStore is a COM class whose base is **IReplStore**. The following code example shows the **CStore** definition.

```
class CStore: public IReplStore
{
private:
    LONG            m_cRef;
    LPUNKNOWN       m_pUnkOuter;

public:
    CStore( LPUNKNOWN );
    ~CStore();

    // ******** IUnknown methods **************
    // QueryInterface, AddRef, and Release

    // ******** IReplStore methods **************
    // Store manipulation routines
    // Initialize, GetStoreInfo, ReportStatus, CompareStoreIDs

    // Object-related routines
    // CompareItem, IsItemChanged, IsItemReplicated, UpdateItem

    // Folder-related routines
    // GetFolderInfo, IsFolderChanged

    // Enumeration of folder objects
    // FindFirstItem, FindNextItem, FindItemClose

    // STD management routines
    // ObjectToBytes, BytesToObject, FreeObject, CopyObject,
    // IsValidObject

    // UI-related routines
    // ActivateDialog, GetObjTypeUIData, GetConflictInfo,
    // RemoveDuplicates

private:
    IreplNotify     *m_pNotify;
    CDataHandler    *m_pObjHandler;
    UINT            m_uFlags;
};
```

Comparing Store Identifiers

When a user connects a device to a desktop:

1. The service manager calls **IReplStore::GetStoreInfo** to retrieve the store's identifier number.

   This number is stored in Repl.dat.
2. The service manager reads the store identifier for the store to be synchronized and passes this identifier number to **IReplStore::CompareStoreIDs**.
3. The desktop provider module determines whether this store identifier matches the one loaded from Repl.dat, which identifies the store that was used in the previous synchronization.

If the identifiers do not match, the service manager reestablishes the mapping between the desktop and device objects by issuing a request to the user either to combine two sets of data or to discard the device data and send all desktop objects to the device.

The combine and discard process is required when Repl.dat is corrupt and cannot be read, as well as when the user:

- Synchronizes a device with existing data for the first time.
- Uses an existing device to delete the device profile, and then reconnects the device to create a new partnership.
- Chooses a desktop store that is different from the one that was used in the last synchronization. The service manager detects the new selection by comparing store identifiers.
- Restores the device data from a backup file.
- Does a discard operation on one computer, and then synchronizes the device with a second computer.

If a user chooses to combine the data, all objects on both the device and the desktop are marked as changed. If a user chooses to discard the data, all objects on the device are marked as deleted and all desktop objects are marked as changed. Synchronization occurs immediately after a user makes a selection.

Combining data typically creates duplicate objects in the desktop store. To find the duplicate objects in the desktop store and remove them, use **IReplStore::RemoveDuplicates**. The service manager calls this method after the first successful synchronization occurs. Once duplicated objects are removed, the desktop provider module returns control to the service manager to begin enumerating the store. Through enumeration, the service manager determines which desktop objects are deleted and informs the device to remove the corresponding device objects.

In addition to using **IReplStore::GetStoreInfo** to provide identification information, you can use this method to perform other tasks, such as changing the size of an object store or the time interval between enumerations. When the service manager calls **IReplStore::GetStoreInfo**, it passes the structure **STOREINFO**. Use **STOREINFO** to change store information:

- To receive a variable-sized store identifier, the service manager calls **IReplStore::GetStoreInfo** with **STOREINFO::cbMaxStoreId** set to 0. This call should prompt the desktop provider module to set the required size for the store identifier in **STOREINFO::cbStoreId** and return E_OUTOFMEMORY. In response, the service manager allocates the required memory and passes a pointer to the store identifier in **STOREINFO::lpbStoreId.** The desktop provider module then can use this pointer to save the store identifier.
- If your desktop provider module does not support real-time notification of changes, you need to set **STOREINFO::uTimerRes**:
  - To enable the service manager to automatically start enumerating the store at a set interval, set **STOREINFO::uTimerRes** to the non-zero time, in micro-seconds, that you want to use for the interval.
  - To enable the service manager to begin the enumeration process only when the user activates the ActiveSync status window or immediately before synchronization begins, set **STOREINFO::uTimerRes** to –1.

The following code example shows how to implement the **IReplStore::CompareStoreIDs** and **IReplStore::GetStoreInfo** methods and the **STOREINFO** structure.

```
STDMETHODIMP_(int) CStore::CompareStoreIDs
(
    LPBYTE  lpbID1,     // Points to the first store identifier
    UINT    cbID1,      // size of the first store identifier
    LPBYTE  lpbID2,     // Points to the second store identifier
    UINT    cbID2       // size of the second store identifier
)
{
    // If the size of the first store identifier is smaller than
    // the size of the second store identifier
    if ( cbID1 < cbID2 )
        return -1;

    // If the size of the first store identifier is larger than
    // the size of the second store identifier
    if ( cbID1 > cbID2 )
        return 1;

    return memcmp( lpbID1, lpbID2, cbID1 );
}
```

```
STDMETHODIMP CStore::GetStoreInfo
(
    PSTOREINFO pInfo    // Pointers to the STOREINFO structure
)
{
    if ( pInfo->cbStruct != sizeof( STOREINFO ) )
        return E_INVALIDARG;

    pInfo->uFlags = SCF_SINGLE_THREAD | SCF_SIMULATE_RTS;

    // ProgId of the store. You can change this to match your company
    // and your product.
    lstrcpy( pInfo->szProgId, "MyCompany.WinCE.DeskSamp" );

    // The description of the store. This will be displayed to the user.
    // You can change this to suit your requirements.
    lstrcpy( pInfo->szStoreDesc, "Files" );

    // Let replication scan the store every 5 seconds.
    pInfo->uTimerRes = 5000;

    // Construct something that uniquely identifies the store. In this
    // example, because the differences in the stores are
    //inconsequential,
    // set the size of the store identifier to any value.
    pInfo->cbStoreId = 10;

    // Compare the size of the the store identifier with the maximum
    // size of the store identifier.
    if ( pInfo->cbStoreId > pInfo->cbMaxStoreId )
        return E_OUTOFMEMORY;

    // Check if the pointer to the store identifier is NULL.
    if ( pInfo->lpbStoreId == NULL )
        return E_POINTER;

    memset( pInfo->lpbStoreId, 0, 10 );
    return NOERROR;
}
```

Accessing Objects

In order for the service manager to access a data object, you must provide an HREPLITEM handle to the object. HREPLITEM is an important data type because each handle uniquely identifies one object. To the service manager, this handle is a 32-bit number created by the desktop provider module. To the desktop provider module, this handle is a pointer to an internal structure or class instance.

Whenever the service manager needs information about the object, it calls methods in **IReplStore** or **IReplObjHandler** and passes the HREPLITEM of the object as a parameter. From the object identifier and the time stamp or version number contained in the HREPLITEM, the desktop provider module can determine whether two handles represent the same object, as well as which of the two handles represents a more recent version of the object.

To compare information, **IReplStore::CompareItem** checks the object identifiers contained in the two handles. The following table shows the possible return values for this method.

Value	Condition
1	The first object identifier is greater than the second object identifier.
–1	The first object identifier is less than the second object identifier.
0	The first object identifier is equal to the second object identifier.

The ordering of the object identifiers allows the service manager to use a binary search on its table of object identifiers.

The following code example shows how to implement **IReplStore::CompareItem**.

```
STDMETHODIMP_(int) CStore::CompareItem
(
    HREPLITEM hItem1,    // Points to the handle of the first object.
    HREPLITEM hItem2     // Points to the handle of the second object.
                         // Both handles are guaranteed to be
                         // returned by IReplStore::FindFirstObject or
                         // IReplStore::FindNextObject.
)
{
    CItem   *pItem1 = (CItem *)hItem1;
    CItem   *pItem2 = (CItem *)hItem2;

    if ( pItem1->m_uid == pItem2->m_uid )
        return 0;
```

```
    if ( pItem1->m_uid < pItem2->m_uid )
        return -1;

    return 1;
}
```

CFolder and **CItem** are COM classes based on **CReplObject**. The following code example shows the definition of these classes.

```
#define OT_ITEM     1
#define OT_FOLDER   2

class CReplObject
{
public:
    virtual ~CReplObject() {}
    UINT    m_uType;
};

class CFolder: public CReplObject
{
public:
    CFolder( void ) { m_uType = OT_FOLDER; }
    virtual ~CFolder() {}
};

class CItem: public CReplObject
{
public:
    CItem( void ) { m_uType = OT_ITEM;
                    memset( &m_ftModified, 0, sizeof( m_ftModified )); }
    virtual ~CItem() {}
    UINT        m_uid;
    FILETIME    m_ftModified;
};
```

Accessing Folders

For the service manager to access a folder, you must implement HREPLFLD, which is a handle that identifies a folder. This handle contains the filter for the object type and other object-specific information.

The service manager calls **IReplStore::GetFolderInfo** to obtain the folder handle. The service manager also calls **IReplStore::IsFolderChanged**, **IReplStore::CopyObject**, **IReplStore::IsValidObject**, and **IReplStore::FreeObject** to manipulate folder and item handles.

Because the service manager saves the data that is stored in the HREPLITEM or HREPLFLD handles to Repl.dat, which is a file that the service manager creates and maintains, you must also implement **IReplStore::ObjectToBytes** and **IReplStore::ObjectToBytes.**

The following code example shows how to implement **IReplStore::GetFolderInfo.**

```
STDMETHODIMP CStore::GetFolderInfo
(
    LPSTR lpszName,          // Name of the object type taken from
                             // the registry
    HREPLFLD *phFolder,      // Output pointers to the handle of the
                             // new folder
    IUnknown **ppObjHandler  // Output pointers to the pointer of
                             // the IReplObjHandler interface
)
{
    // Check if phFolder points to a handle that has NULL value.
    CFolder *pFolder = (CFolder *)*phFolder;
    BOOL fNew = (pFolder == NULL);

    // Create a new handle for the specific folder.
    if ( fNew )
        pFolder = new CFolder;

    // Either set up the new CFolder class (when fNew is TRUE) or
    // reinitialize the class (when fNew is FALSE).
    // ...


    *phFolder = (HREPLFLD)pFolder;
    *ppObjHandler = m_pObjHandler;

    return NOERROR;
}
```

To determine if any object contained in a folder has changed, the service manager calls **IReplStore::IsFolderChanged**. The service manager also can call the following methods to manipulate folder or item handles:

- **IReplStore::CopyObject** to copy the data from one handle to another handle that represents the same object.
- **IReplStore::IsValidObject** to ensure that the handle still represents a valid object, not a deleted object.
- **IReplStore::FreeObject** to remove a handle from memory, thereby allowing the desktop provider module to free the memory resources that are used by the handle.

The following code examples show how to implement each of these methods.

```
STDMETHODIMP_(BOOL) CStore::CopyObject
(
    HREPLOBJ    hObjSrc,    // Handle to the source object
    HREPLOBJ    hObjDst     // Handle to the destination object
)
{
    CReplObject *pObjSrc = (CReplObject *)hObjSrc;
    CReplObject *pObjDst = (CReplObject *)hObjDst;

    // Check to see if the source and destination types are the same.
    if ( pObjSrc->m_uType != pObjDst->m_uType )
        return FALSE;

    switch( pObjSrc->m_uType )
    {
    case OT_ITEM:           // If the source object is an item
        ((CItem *)pObjDst)->m_uid = ((CItem *)pObjSrc)->m_uid;
        ((CItem *)pObjDst)->m_ftModified =
                            ((CItem *)pObjSrc)->m_ftModified;
        break;

    case OT_FOLDER:         // If the source object is a folder
        break;
    }
    return TRUE;
}

STDMETHODIMP CStore::IsValidObject
(
    HREPLFLD hFolder,   // Handle of the folder where this
                        // item belongs.
    HREPLITEM hItem,    // Handle of the object; could be NULL.
    UINT uFlags         // Reserved; must be 0.
)
{
    CFolder *pFolder = (CFolder *)hFolder;
    CItem   *pItem = (CItem *)hItem;

    if ( pFolder )
    {
        // Check whether hFolder is a valid folder handle.
        if ( pFolder->m_uType != OT_FOLDER )
            return HRESULT_FROM_WIN32( ERROR_INVALID_HANDLE );
    }
```

```
    if ( pItem )
    {
        // Check whether hItem is a valid item handle.
        if ( pFolder->m_uType != OT_ITEM )
            return HRESULT_FROM_WIN32( ERROR_INVALID_HANDLE );

        // Search for the item. If the item is not found, return
        // to HRESULT_FROM_WIN32( ERROR_FILE_NOT_FOUND ).
        // ...
    }

    return NOERROR;
}

STDMETHODIMP_(void) CStore::FreeObject
(
    HREPLOBJ    hObject     // Handle of the object whose contents
                            // need to be freed
)
{
    delete (CReplObject *)hObject;
}
```

The service manager saves the data stored in the HREPLITEM or HREPLFLD handles to Repl.dat. The desktop provider module must implement **IReplStore::ObjectToBytes** order to convert an HREPLITEM object or HREPLFLD object into a series of bytes so that the service manager can save the data. The desktop provider module also must implement **IReplStore::BytesToObject** to convert the same series of bytes back to an object.

When a user connects a device to a desktop, the service manager reads Repl.dat and restores all handles that were used in the previous synchronization. As long as a device is connected, the service manager continues to save handles into Repl.dat.

The following code examples show how to implement **IReplStore::ObjectToBytes** and **IReplStore::BytesToObject**.

```
STDMETHODIMP_(UINT) CStore::ObjectToBytes
(
    HREPLOBJ    hObject,    // Handle to the object.
    LPBYTE      lpb         // Points to a buffer where the array of
                            // bytes should be stored; could be NULL.
)
```

```
{
    LPBYTE      lpbStart = lpb;
    CReplObject *pObject = (CReplObject *)hObject;
    CFolder     *pFolder = (CFolder *)pObject;
    CItem       *pItem = (CItem *)pObject;

    if ( lpbStart )
        *lpb = OBJECT_VERSION;
    lpb++;

    if ( lpbStart )
        *(PUINT)lpb = pObject->m_uType;
    lpb += sizeof( pObject->m_uType );

    switch( pObject->m_uType )
    {
    case OT_FOLDER:          // If the object is a folder
        break;

    case OT_ITEM:            // If the object is an item
        if ( lpbStart )
            *(PUINT)lpb = pItem->m_uid;
        lpb += sizeof( pItem->m_uid );

        if ( lpbStart )
            *(FILETIME *)lpb = pItem->m_ftModified;
        lpb += sizeof( pItem->m_ftModified );
        break;
    }
    return lpb - lpbStart;
}

STDMETHODIMP_(HREPLOBJ) CStore::BytesToObject
(
    LPBYTE  lpb,         // Points to a buffer where the array of bytes
                         // should be stored; could be NULL.
    UINT    cb           // Size of the buffer.
)
{
    CReplObject *pObject = NULL;
    CFolder *pFolder;
    CItem *pItem;

    BYTE bVersion = *lpb++;
    UINT uType = *(PUINT)lpb;

    lpb += sizeof( uType );
```

```
    if ( bVersion != OBJECT_VERSION )
    {
        // Convert the data based on bVersion.
    }

    switch( uType )
    {
    case OT_FOLDER:      // If the object is a folder
        pObject = pFolder = new CFolder;
        break;

    case OT_ITEM:        // If the object is an item
        pObject = pItem = new CItem;

        pItem->m_uid = *(PUINT)lpb;
        lpb += sizeof( pItem->m_uid );

        pItem->m_ftModified = *(FILETIME *)lpb;
        lpb += sizeof( pItem->m_ftModified );

        break;
    }

    return (HREPLOBJ)pObject;
}
```

Enumerating Objects

Enumeration is the process of accessing each object in a directory or folder to determine whether it has changed and whether it meets any specified filtering criteria. The enumeration process is as follows:

1. The service manager calls **IReplStore::IsFolderChanged**.
2. If **IReplStore::IsFolderChanged** returns TRUE, the service manager calls **IReplStore::FindFirstItem**.
3. If additional objects exist, **IReplStore::FindFirstItem** sets **pfExist* to TRUE.
4. The service manager calls **IReplStore::FindNextItem** repeatedly until **pfExist* is FALSE, which indicates that there are no more objects.
5. The service manager calls **IReplStore::FindItemClose** to free any resources that were used during enumeration.

Note If an enumeration takes more than a few seconds to complete, you can use **IReplNotify::SetStatusText** to have the desktop provider module display text that notifies the user how much time remains until completion.

The following code example shows how to implement **IReplStore::FindFirstItem**.

```
STDMETHODIMP CStore::FindFirstItem
(
    HREPLFLD hFolder,    // Handle to a folder
    HREPLITEM *phItem,   // Output pointer to the handle of the first
                         // item in the folder
    BOOL *pfExist        // Output pointer to a Boolean value that is set
                         // to TRUE if there is an object in the folder
)
{
    CFolder *pFolder = (CFolder *)hFolder;

    // Find the first item. pItem used in the following code points to
    // the first item.
    // ...



    // Implementation should retrieve a unique identifier from
    // the first item.
    // pItem->m_uid = retrieved_unique_ID;

    // Implementation also should retrieve whatever it is using to
    // verify if the first item is changed; for example, a time stamp.
    // pItem->m_ftModified = retrieved_time_stamp;

    *phItem = (HREPLITEM)pItem;

    if ( pfExist )
        *pfExist = TRUE;

    return NOERROR;
}
```

You can use the desktop provider module to filter synchronization objects during enumeration by returning to the service manager only those objects that meet your filtering criteria. For example, you may want to synchronize only appointments that fall within the next three days. The service manager accesses the filtering criteria—which is stored with the folder handle—by calling **IReplStore::IsItemReplicated**.

Note If a filter is implemented incorrectly, any object that fails the filter criteria appears as a deleted object on the desktop. If the desktop enumeration does not return the object, the service manager assumes that the object is deleted and deletes the corresponding device object. To avoid this, be sure that the desktop provider module returns every object in the store during enumeration.

You can reapply a filter as certain conditions change. For example, you may want to synchronize the appointments falling within the next three days, everyday. This would mean that the date criteria for the filter would change as the days pass. In order to determine exactly what date range to filter for, the service manager calls **IReplStore::ReportStatus** with RSC_DATE_CHANGED to initiate the synchronization process.

Detecting Desktop Object Changes

The service manager automatically detects object changes and deletions by comparing the list of handles that are returned during enumeration with the handles that are saved in Repl.dat. Before starting an enumeration process, the service manager:

1. Marks a bit for each handle that is stored in Repl.dat.
2. Calls **IReplStore::FindFirstItem** and **IReplStore::FindNextItem**.
3. Performs a binary search on the handles in Repl.dat, each time one of these methods returns a new handle, in order to find a handle representing the same object.
   - If no matching handle is found, it creates a new object on the desktop store.
   - If a matching handle is found, it clears the bit from the handle in Repl.dat and calls **IReplStore::IsItemChanged** to see if the object has changed since the last synchronization.

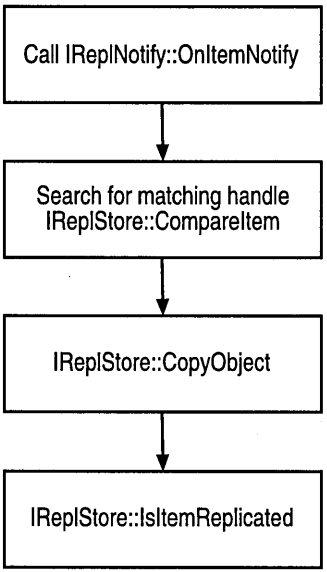
     If the object has changed, it calls **IReplStore::CopyObject** to copy the data from the returned handle into the handle that is saved in Repl.dat.
4. Calls **IReplStore::IsItemReplicated** to see if it should send the object to the device.

All handles in Repl.dat that remain marked once enumeration is complete represent deleted objects.

To hasten the enumeration process, the desktop provider module can detect and report desktop changes to the service manager in real time. To do this:

1. The service manager calls **IReplNotify::OnItemNotify** to inform the desktop provider module of the status for an object.
2. The desktop provider module passes RNC_MODIFIED for a modified object, RNC_CREATED for a created object, and RNC_DELETED for a deleted object.
3. The desktop provider module passes a handle to the object so that the service manager can search Repl.dat for the corresponding device object.

The following illustration shows the sequence of real-time notification calls.



The desktop provider module also can call **IReplNotify::OnItemNotify** with RNC_SHUTDOWN if it detects that the desktop application has closed. The service manager responds by unloading the desktop provider module and updating the status display.

The following code examples show how to implement **IReplStore::IsItemChanged** and **IReplStore::IsItemReplicated**.

```
STDMETHODIMP_(BOOL) CStore::IsItemChanged
(
    HREPLFLD    hFolder,    // Handle of the folder or the container
                            // that stores the object
    HREPLITEM   hItem,      // Handle of the object
    HREPLITEM   hItemComp   // Handle of the object that is used
                            // for comparison
)
{
    CFolder *pFolder = (CFolder *)hFolder;
    CItem *pItem = (CItem *)hItem;
    CItem *pItemComp = (CItem *)hItemComp;
    BOOL fChanged = FALSE;

    if ( pItemComp )
        fChanged = CompareFileTime( &pItem->m_ftModified,
                                    &pItemComp->m_ftModified );
    else
    {
        FILETIME ft;

        // Read the modification time stamp from the object into ft.
        // Compare it with the time stamp given in the object.
        fChanged = CompareFileTime( &pItem->m_ftModified, &ft );
    }
    return fChanged;
}

STDMETHODIMP_(BOOL) CStore::IsItemReplicated
(
    HREPLFLD    hFolder,    // Handle of the folder or the container
                            // that stores the object
    HREPLITEM   hItem       // Handle of the object
)
{
    CFolder *pFolder = (CFolder *)hFolder;
    CItem *pItem = (CItem *)hItem;

    // hItem can be passed NULL.
    if ( pItem == NULL )
        return TRUE;

    // Search for the item; return FALSE if the item is not found.
```

```
    // Check if pItem should be replicated by using information stored
    //in pFolder & pItem. If so, return TRUE.

    return FALSE;
}
```

Sending and Receiving Objects

You use **IReplObjHandler** to serialize or convert an object to a series of bytes. It also deserializes an object or converts the bytes back to an object to transmit data to a device or desktop. There is no limitation or specification on how to serialize an object. A desktop provider module can serialize the object into any number of bytes, and it can group these bytes into any number of packets.

Whenever an object requires serialization, the service manager calls the **IReplObjHandler** methods in the following order:

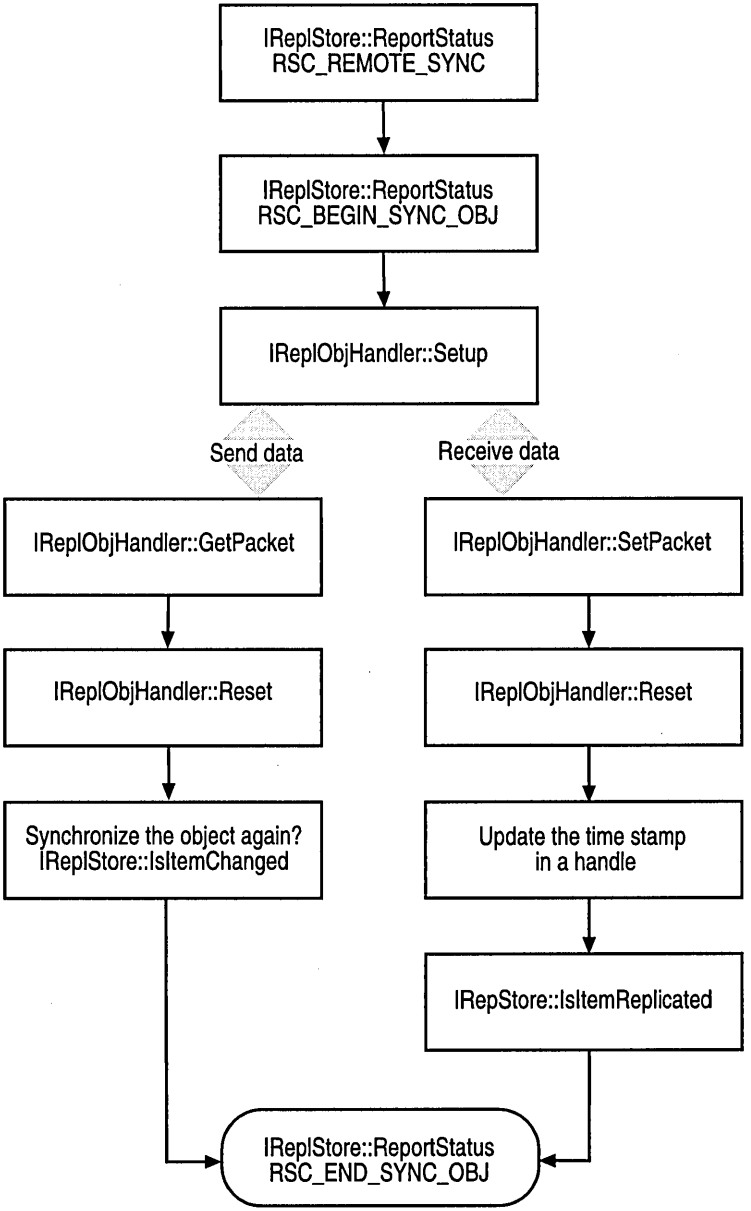
1. **IReplObjHandler::Setup** to tell the desktop provider module what object to serialize and to enable the desktop provider module to allocate any resources that are required for serialization.
2. **IReplObjHandler::GetPacket** to let the desktop provider module create one or more packets of bytes of any size.

   The service manager calls this method repeatedly until RWRN_LAST_PACKET is returned.
3. **IReplObjHandler::Reset** to let the desktop provider module free any used resources.

   The service manager calls this method when serialization is completed.

The service manager does not recognize byte format during serialization. The service manager simply sends the packets in the same byte number and sequence as they were originally received.

The following illustration shows the sequence of calls necessary to send and receive data.



The following code example shows how to implement **IReplObjHandler::Setup** and **IReplObjHandler::GetPacket**. It also shows the definition for **CDataHandler**, which is a COM class whose base is **IReplStore**.

```
class CDataHandler : public IReplObjHandler
{
public:
    CDataHandler( CStore *pStore );
    ~CDataHandler();

    // ******** IUnknown methods **************
    // AddRef, Release, QueryInterface

    // ******** IReplObjHandler methods **************
    // Setup, Reset, GetPacket, SetPacket, DeleteObj

private:
    long m_cRef;
    PREPLSETUP m_pWriteSetup, m_pReadSetup;
};

STDMETHODIMP CDataHandler::Setup
(
    PREPLSETUP pSetup  // Points a REPLSETUP structure, which contains
                       // information about the object to be serialized
                       // or deserialized.
)
{
    // Could be reading and writing at the same time, so it is necessary
    // to save the pointer to set up the structure differently.
    if ( pSetup->fRead )
        m_pReadSetup = pSetup;
    else
        m_pWriteSetup = pSetup;

    return NOERROR;
}

STDMETHODIMP CDataHandler::GetPacket
(
    LPBYTE  *lppbPacket,     // Pointer to the pointer of the outgoing
                             // packet
    DWORD   *pcbPacket,      // Pointer to a DWORD for the packet size
    DWORD   cbRecommend      // Recommended maximum size of the packet
)
{
    if ( m_pReadSetup->hItem == NULL )
        return E_UNEXPECTED;
```

```
    // Initialize the packet.
    // ...


    // Fill up the packet.
    // ...


    // Assign the size of the packet to *pcbPacket and the packet
    // to *lppbPacket.
    //*pcbPacket = sizeof( packet );
    //*lppbPacket = (LPBYTE)&packet;

    return NOERROR;
    //Return RWRN_LAST_PACKET if it is the final packet.
}
```

During synchronization, the service manager creates an instance of **IReplObjHandler** for each object type. The desktop provider module recognizes when **IReplObjHandler::SetPacket** is sending information about a new object by checking for RSF_NEW_OBJECT in **REPLSETUP::dwFlags**. The service manager passes **REPLSETUP::dwFlags** in the **IReplObjHandler::Setup** call. The structure **REPLSETUP** is passed in **IReplObjHandler::Setup**.

The following table shows the **REPLSETUP** members that are used by the desktop provider module. All other members are internal to the service manager and should not be changed.

REPLSETUP member	Description
fRead	Set to TRUE for reading an object from the desktop store. Set to FALSE for writing an object to the desktop store.
dwFlags	A collection of bit flags related to serialization and deserialization.
Hfolder	A handle to the folder.
HItem	A handle to the object to be serialized. The desktop provider module uses the information in this handle to identify and convert the object into packets of bytes.

The process of receiving an object from the device is similar to sending an object to the device. After the packets of data arrive from the device, the service manager calls the **IReplObjHandler** interface methods, which enable the desktop provider module to convert those packets back into an object. The desktop provider module must take the data packets and create an object. It then must create a new **HREPLITEM** that represents the object in **REPLSETUP::hItem**.

The **IReplObjhandler** methods are always called in the following sequence:

1. **IReplObjHandler::Setup** tells the desktop provider module what object to deserialize and enables the desktop provider module to allocate any resource needed for deserialization.
2. **IReplObjHandler::SetPacket** sends packets to the desktop provider module so that it can recreate the object.

   Packets are sent in the exact same number, size, and sequence. The service manager calls this method repeatedly until the last packet is received from the device.
3. **IReplObjHandler::Reset** enables the desktop provider module to free any used resources.

   The service manager calls this method when deserialization is completed.

When the service manager writes an object to the desktop store, it calls **IReplStore::UpdateItem**. This prompts the desktop provider module to open the object and update the **HREPLITEM** handle with a time stamp or version number, thus ensuring that the object is not marked as changed on the desktop.

The following code example shows how to implement **IReplStore::UpdateItem**.

```
STDMETHODIMP_(void) CStore::UpdateItem
(
    HREPLFLD    hFolder,    // Handle of a folder
    HREPLITEM   hItemDst,   // Handle of the destination object
    HREPLITEM   hItemSrc    // Handle to the source object
)
{
    CFolder *pFolder = (CFolder *)hFolder;
    CItem   *pItemDst = (CItem *)hItemDst;
    CItem   *pItemSrc = (CItem *)hItemSrc;

    if ( pItemSrc )
    {
        pItemDst->m_ftModified = pItemSrc->m_ftModified;
    }
    else
    {
        // Implementation should update whatever it has used to validate
        // object changes, such as a time stamp, by reading it from the
        // object.

        // Update the time stamp.
        // pItemDst->m_ftModified = time stamp read from object.
    }
}
```

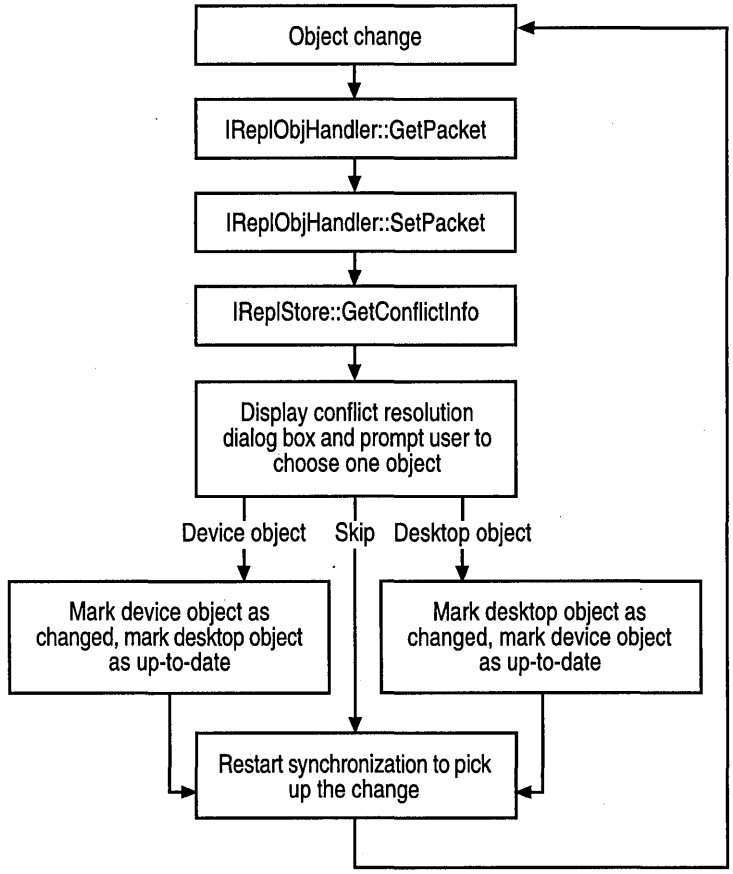
Depending on the type of application, a desktop provider module may synchronize objects coming from a device as deletions. For example, when a user deletes an e-mail message on a device, the message is moved to the Deleted Item folder and marked as changed. The desktop provider module receives the changed object and automatically attempts to delete the message on the desktop. In response, **IReplObjHandler::SetPacket** returns one of the following errors:

- RERR_DISCARD when the desktop provider module tries to delete the device object immediately after the change is synchronized. The service manager sends a command to the device to delete the corresponding object.
- RERR_DISCARD_LOCAL when the desktop provider module tries to delete the desktop object immediately after the change is synchronized. The service manager calls **IReplObjHandler::DeleteObject** to delete the existing desktop object.

Handling Conflicts

A conflict occurs when an object has changed on both the device and the desktop since the last synchronization. To resolve a conflict, the service manager calls **IReplObjHandler::GetPacket** on the device to send the object to the desktop. The service manager then calls **IReplObjHandler::SetPacket** on the desktop to create a temporary object. During both the device and the desktop call, the service manager passes RSF_CONFLICT_OBJECT in **REPLSETUP::dwFlags**.

The following illustration shows the call sequence for conflict resolution.



After the service manager receives the data from the device, it calls **IReplStore::GetConflictInfo** and passes a handle to both the original desktop object and the temporary device object. Next, the desktop provider module must fill in the **CONFINFO** structure to customize the description text displayed in a standard **Conflict Resolution** dialog box, which is supplied by the service manager.

The following code example shows how to implement **IReplStore::GetConflictInfo**.

```
STDMETHODIMP CStore::GetConflictInfo
(
    PCONFINFO pConfInfo      // Pointer to a CONFINFO structure
)
{
    // Verify that you have the right version of OBJUIDATA.
    if ( pConfInfo->cbStruct != sizeof( CONFINFO ) )
        return E_INVALIDARG;

    // Copy "Stock" to szLocalName and szRemoteName. You can use your
    // own local and remote object names to replace "Stock".
    lstrcpy( pConfInfo->szLocalName, "Stock" );
    lstrcpy( pConfInfo->szRemoteName, "Stock" );

    CItem   *pLocalItem = (CItem *)pConfInfo->hLocalItem;
    CItem   *pRemoteItem = (CItem *)pConfInfo->hRemoteItem;

    // Find the local object, and then the remote object.
    // pLocalObject points to the local object.
    // pRemoteObject points to the remote object.

    // If both the local and remote objects are found
    if (pLocalObject && pRemoteObject )
    {
        // Compare the local and remote objects.
        // If the local and remote objects are identical
        return RERR_IGNORE;
    }

    // If a local object is found
    if ( pLocalObject )
        // Store information for the local object
        // in pConfInfo->szLocalDesc.

    // If a remote object is found
    if ( pRemoteObject )
        // Store information for the remote object in
        // pConfInfo->szRemoteDesc.

    return NOERROR;
}
```

If the desktop provider module cannot write a temporary object on the desktop, it can save the packets into memory and return **HREPLITEM** containing a pointer to the memory location. In this case, the desktop provider module must implement this handle in all methods in **IReplStore** that accept an **HREPLITEM** handle, such as **CopyObject** or **FreeObject**. When the service manager calls **IReplStore::GetConflictInfo**, the handle becomes **CONFINFO**::**hRemoteItem**. The desktop provider module then can extract descriptive text from the handle and save it into **CONFINFO**.

The **Conflict Resolution** dialog box enables the user to:

- Discard the original desktop object and keep the newly written device object.

  If the user chooses to keep the newly written device object, the service manager marks the desktop object as up to date and the device object as changed. This ensures that the device object is transferred to the desktop during the next synchronization.
- Keep the original desktop object and discard the device object.

  If the user chooses to keep the desktop object, the service manager marks the device object as up-to-date.

In either case, the service manager calls **IReplObjHandler::DeleteObject** to delete the temporary object.

Conflict situations do not always require a **Conflict Resolution** dialog box. The following table describes special error values from **IReplStore::GetConflictInfo** that resolve the conflict automatically.

Error value	Action
RERR_IGNOR	The desktop provider module compares two handles in **CONFINFO**, determines that they are identical, and takes no action.
RERR_DISCARD	The service manager detects that the desktop object represented by a handle is already deleted and deletes the device object accordingly.
RERR_DISCARD_LOCAL	The service manager resolves a conflict by deleting a desktop object instead of allowing the desktop provider module to delete the object.

Setting Synchronization Options

For a user to change synchronization options, the desktop provider module must supply a UI that enables a user to select options. There is no limitation or specification for the UI.

A user can gain access to the UI in two ways:

- If a device is connected to the desktop, a user selects a service provider from the desktop.
- If the device is not connected to the desktop, a user selects a service provider from the Mobile Device folder.

Once a user selects a service provider, the service manager calls **IReplStore::ActivateDialog**, which displays the UI for the selected service provider. If the desktop provider module does not support a synchronization options dialog box, the call to **IReplStore::ActiveDialog** must return E_NOTIMPL. To get a handle to a parent window to display a dialog box or message box, have the desktop provider module call **IReplNotify::GetWindow.**

If a user chooses to remove a service provider, **IReplStore::ActivateDialog** returns RERR_UNLOAD. If a user cancels the synchronization options dialog box, **IReplStore::ActivateDialog** returns RERR_CANCEL.

You can save synchronization options for the desktop provider module either in the registry or in the HREPLFLD handle. If you save synchronization options in a HREPLFLD handle, the desktop provider module sets default option values in **IReplStore::GetFolderInfo**. If the pointer to which *phFolder* points is NULL, the desktop provider module must create a new **HREPLFLD**. If the pointer is not NULL, the desktop provider module loads the options from Repl.dat.

Developing the Device Provider Module

The service manager interfaces with the device provider module to access data. To facilitate this interaction, a device provider module must implement the same **IReplObjHandler** interface that you use for the desktop provider module. It also must implement the **InitObjType**, **GetObjTypeInfo**, **ObjectNotify** functions. The device provider module can also implement the **ReportStatus** function, which is optional.

▶ **To implement the device provider module**

1. Create one GUID for the store, by using the Visual C++ GUID generator tool, Guidgen.exe.
2. Initialize the device store.
3. Enumerate device objects.
4. Detect device object changes and send and receive device data.

Initializing the Device Store

To initialize the device provider module, call **InitObjType**. If the device provider module supports synchronization of multiple object types, it calls **InitObjType** for each object type whose *lpszObjType* is not NULL. When an ActiveSync service terminates, the *lpszObjType* of its objects are set to NULL, thereby allowing the device provider module to free any resources it may have allocated.

The following code example shows how to implement **InitObjType**.

```
EXTERN_C BOOL InitObjType
(
    LPWSTR lpszObjType,
    IReplObjHandler **ppObjHandler,
    UINT uPartnerBit
)
{
    if ( lpszObjType == NULL )
    {
        // Terminates the device provider module and frees all
        // allocated resources.
        // ...

        return TRUE;
    }

    // Allocates a new IReplObjHandler.
    *ppObjHandler = new CDataHandler;

    // Saves the uPartnerBit so that you can use it later in
    // ObjectNotify.

    // Initializing the module.
    // ...

    return TRUE;
}
```

ActiveSync supports the synchronization of two desktop computers with a Windows CE–based device. To differentiate between two computers, the service manager passes a partner bit into **InitObjType** when initializing the device provider module. This bit is set to 1 if the connected desktop computer is the first partner and 2 if it is the second partner. The device provider module must use this partner bit when setting and resetting dirty bits of an object.

Enumerating Device Objects

Enumerating objects on a device differs from enumerating objects on a desktop. In contrast to desktop enumeration, the service manager enumerates each object and calls the **ObjectNotify** function of each device provider module when a user connects a device. If an object on a device needs synchronizing, the device provider module must inform the service manager.

Detecting Device Object Changes

Changes on the device are handled through the **ObjectNotify** function. The service manager calls **ObjectNotify** in the following cases:

- Immediately after connection.
- When an object changes and a user connects the device.
- After an acknowledgement is received from the desktop that the object has been synchronized successfully.
- Two seconds after the device provider module sets ONF_CALLING_BACK.

ObjectNotify checks the flags and the file system identifier information in the **OBJNOTIFY** structure for a changed object.

The following table shows the values for the **OBJNOTIFY::uFlags** member.

Uflags member	Definition
ONF_FILE	**OBJNOTIFY::oidObject** is a file.
ONF_DIRECTORY	**OBJNOTIFY::oidObject** is a directory.
ONF_RECORD	**OBJNOTIFY::oidObject** is a record.
ONF_DATABASE	**OBJNOTIFY::oidObject** is a database.
ONF_CHANGED	The file system object is changed.
ONF_DELETED	The file system object is deleted. Only **oidParent** in **OBJNOTIFY::oidInfo** is defined. All other members in **OBJNOTIFY::oidInfo** are 0.
ONF_CLEAR_CHANGE	The desktop provider module should mark the object as up to date. In this case, **OBJNOTIFY::oidObject** is the synchronized object identifier, and not the Windows CE object identifier.
ONF_CALL_BACK	Set by the device provider module to ask the service manager to call back.
ONF_CALLING_BACK	The service manager sets this flag and calls **ObjectNotify**.

The following code example shows how to implement **ObjectNotify**.

```
EXTERN_C BOOL ObjectNotify( POBJNOTIFY pNotify )
{
    // Check to see if the structure size is the same.
    if ( pNotify->cbStruct != sizeof( OBJNOTIFY ) )
         return FALSE;

    // Check ONF_* flags to see if the notification is
    // relevant.
    if ( !( pNotify->uFlags & ONF_DELETED ) )
    {
        // Make sure that you are dealing with the records in your
        // database.
        // The object (a record or a file) must exist.
    }

    if ( pNotify->uFlags & ONF_CLEAR_CHANGE )
    {
        // Check whether the object was changed again
        // during synchronization.
        // If so, return TRUE; if not, return FALSE.
    }

    pNotify->poid = (UINT *)&pNotify->oidObject;

    // Determine what object identifier to return. Consider
    // uPartnerBit in your decision.

    // If you store one object per file and/or record, you simply
    // need to return the file system object identifier. Otherwise,
    // you need to read the file system object and determine the list of
    // object identifiers that have changed.


    return TRUE;
}
```

To get information from a database with the **oidDataBase** object identifier, the service manager calls **GetObjTypeInfo**.

The following code example shows how to implement **GetObjTypeInfo**.

```
EXTERN_C BOOL GetObjTypeInfo
(
POBJTYPEINFO pInfo      // Pointer to the OBJTYPEINFO structure
)
{
CEOIDINFO   oidInfo;

if ( pInfo->cbStruct != sizeof( OBJTYPEINFO ) )
return FALSE;

    // Clear the structure.
    memset( &(oidInfo), 0, sizeof(oidInfo));

    // Retrieve information about the object in the object store.
CeOidGetInfo( oidDataBase, &oidInfo );

    // Store the database information into the OBJTYPEINFO structure.
wcscpy( pInfo->szName, oidInfo.infDatabase.szDbaseName );
    pInfo->cObjects = oidInfo.infDatabase.wNumRecords;
pInfo->cbAllObj = oidInfo.infDatabase.dwSize;
pInfo->ftLastModified = oidInfo.infDatabase.ftLastModified;

    return TRUE;
}
```

Registering the Service Provider Module

For Windows CE Services to recognize a service provider, you must create valid registry entries in the registry on both the Windows-based desktop computer and on the Windows CE–based device. You also must register the object types.

Note The CEUTIL utility DLL functions are especially helpful when adding desktop registry entries for Windows CE. For more information, see "Installing and Managing Applications."

▶ **To register a service provider on the Windows-based desktop computer**

1. Provide a programmatic identifier (*ProgID*) for the desktop provider module. The *ProgID* must be a unique name.
2. Generate a GUID (*Class ID*) for the service provider.
3. Create the following keys in the Windows registry:

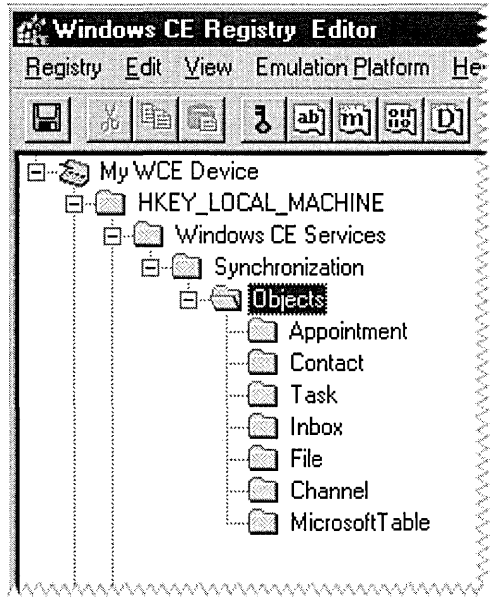
   HKEY_CLASSES_ROOT\CLSID*Class ID***\InProcServer32**

   HKEY_CLASSES_ROOT\CLSID*Class ID***\ProgID**

   HKEY_CLASSES_ROOT*ProgID***\CLSID**

The default value of the **InProcServer32** key is the full path of the 32-bit DLL that implements the **IReplStore** interface (for example, for Microsoft Outlook the path is Outstore.dll); the default value of the **ProgID** key is MS.WinCE.Outlook; and the default value of the **CLSID** key is the GUID for the store.

After registering the service provider, you must register each object type that the service provider synchronizes. Register the object types in a subdirectory under **HKEY_LOCAL_MACHINE**. The following screen shot illustrates the desktop registry location for the appointment, contact, and task object types.



Each object type name is a key. Under each key, you must define the following five values:

- **Default**

  A description of the object type; for example, **Outlook Appointment Object**.
- **Display Name**

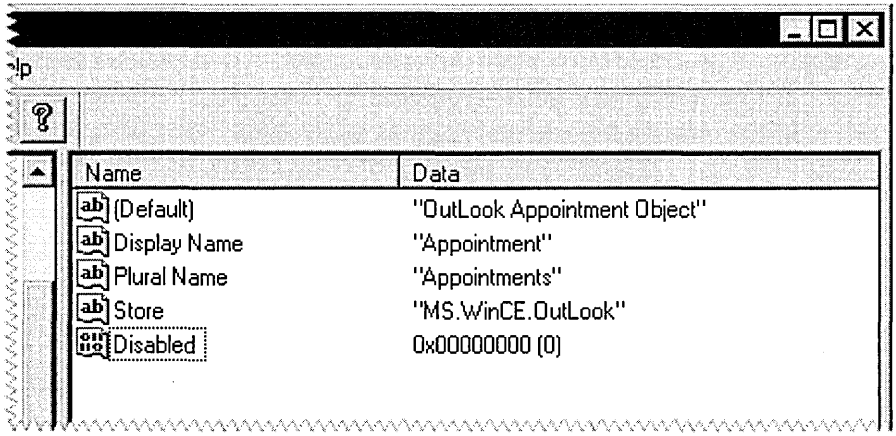
  The name of the object that you want to display; for example, **Appointment**.
- **Plural Name**

  The plural name of the object; for example, **Appointments**.
- **Store**

  The OLE programmatic identifier, *ProgID*, of the store that implements the **IReplStore** and **IReplObjHandler** interfaces; for example, **MS.WinCE.Outlook**.
- **Disabled**

  A value indicating whether the service provider is displayed as disabled or enabled in Windows CE Services. A nonzero value indicates that the service provider is disabled.

The following screen shot illustrates the values defined under the **Appointment** object type.



Whenever a user connects a new device to the desktop and a new device profile is added to Mobile Device folder, the registry keys for the synchronization objects under **HKEY_LOCAL_MACHINE** are automatically copied to **HKEY_CURRENT_USER**.

▶ **To register a service provider on the Windows CE–based device**

- Register the device provider module under **HKEY_LOCAL_MACHINE\Windows CE Services\Synchronization\Objects**.

On the device, each object type name is a key, but you only define two values: **Store** and **Display Name**. **Store** refers to the module that exports the functions for this object type, and **Display Name** refers to the name of the object type.

The following registry data shows the device registration for the **Appointment** object type that was explained above.

```
Store   "pegobj.dll"
Display Name"Appointment"
```

CHAPTER 10

Installing Applications

Developers of Windows CE–based platforms have created additional capabilities for use on the Handheld PC and Palm-size PC devices. This section describes how to install applications on and remove applications from a Windows CE–based device. This includes:

- Using the CAB Wizard to create a Windows CE cabinet (.cab) file to install applications on a Windows CE–based device.
- Using the Application Manager application, CeAppMgr.exe, to install applications on and remove applications from a Windows CE–based device, as well as to remove the application's files from the desktop computer.

This section also explains how to add additional menu items to the **Tools** menu in the Windows CE Explorer window by using code to directly place values in the proper registry locations. You can also use the CEUTIL utility dynamic-link library (DLL), which is described in "Using the CEUTIL Helper DLL for Windows CE Services," to create custom menus and perform other tasks.

Using the CAB Wizard

The Windows CE operating system (OS) uses a .cab file to install an application on a Windows CE–based device. A .cab file is composed of multiple files that have been compressed into one file. Compressing multiple files into one file provides the following benefits:

- All of the application's files are present.
- You can prevent a partial installation.
- You can install your application from several sources, such as a desktop computer or a Web site.

Use the CAB Wizard application (Cabwiz.exe) to generate a .cab file for your application.

▶ **To create a device-specific .cab file for an application**

1. Create an .inf file with Windows CE–specific modifications.
2. Optionally, create a Setup.dll file to provide custom control of the installation process.
3. Use the CAB Wizard to create the .cab file, using the .inf file, the Setup.dll file, and the device-specific application files as parameters.

Creating an .inf File for the CAB Wizard

An .inf file specifies information about an application for the CAB Wizard. The following table shows the sections of an .inf file.

Section	Required	Describes
Version	Yes	The application's creator and version
CEStrings	Yes	String substitutions for application and directory names
Strings	No	String definitions for one or more strings
CEDevice	Yes	The device platform for which the application is targeted
DefaultInstall	Yes	The default installation of the application
SourceDiskNames	Yes	The name and path of the disk on which the application resides
SourceDiskFiles	Yes	The name and path of the files in which the application resides
DestinationDirs	Yes	The names and paths of the destination directories for the application on the target device
CopyFiles	Yes	Default files to copy to the target device
AddReg	Yes	Keys and values that the .cab file will add to the registry on the device
CEShortCuts	No	Shortcuts that the installation application creates on the device

Version

The [Version] section is required and specifies the creator of the file and other relevant information.

```
[Version]
Signature = "signature_name"
Provider = "INF_creator"
CESignature = "$Windows CE$"
```

signature_name
Must be **"$Windows NT$"** or **"$Windows 95$"**.

INF_creator
Company name of the application. For example:

```
Provider = "Microsoft"
```

The following code example shows a typical [Version] section.

```
[Version]
Signature = "$Windows NT$"
Provider = "Microsoft"
CESignature = "$Windows CE$"
```

CEStrings

The [CEStrings] section is required and specifies string substitutions for the application name and the default installation directory.

```
[CEStrings]
   AppName = app_name
InstallDir = default_install_dir
```

app_name
Name of the application. Other instances of *%AppName%* in the .inf file will be replaced with this string value.

default_install_dir
Default installation directory on the device. Other instances of *%InstallDir%* in the .inf file will be replaced with this string value.

The following code example shows a typical [CEStrings] section.

```
[CEStrings]
AppName="Game Pack"
InstallDir=%CE1%\%AppName%
```

Strings

The [Strings] section is optional and defines one or more string keys. A string key represents a string of printable characters.

```
[Strings]
string_key = value
[string_key = value]
```

value

String consisting of letters, digits, or other printable characters. Enclose *value* in double quotation marks (“” ””) if the corresponding string key is used in an item that requires double quotation marks.

The following code example shows a typical [Strings] section.

```
[Strings]
reg_path = Software\Microsoft\My Test App
```

CEDevice

The [CEDevice] section is optional and describes the platform for which your application is targeted. All keys in this section are optional. If a key is nonexistent, Windows CE does not perform any checking. Windows CE also does not perform any checking if a key has no data. The exception is **UnsupportedPlatforms**; if this key exists but there is no data, the previous value is not overridden.

```
[CEDevice]
[ProcessorType =[processor_type]]
[UnsupportedPlatforms = platform_family_name[,platform_family_name]]
[VersionMin = [major_version.minor_version]]
[VersionMax = [major_version.minor_version]]
[BuildMin = [build_number]]
[BuildMax = [build_number]]
```

processor_type

Value that is returned by **SYSTEMINFO.***dwProcessorType*. For example, the value for the SH3 CPU is 10003 and the MIPS CPU is 4000.

platform_family_name
List of platform family names that are known to be unsupported. If the name specified in the [CEDevice.*xxx*] section is different from that in the [CEDevice] section, both *platform_family_name* values are unsupported for the microprocessor that is specified by *xxx*. That is, the list of specific unsupported platform family names is appended to the previous list of unsupported platform family names. Application Manager will not display the application for an unsupported platform. Also, a user will be warned during the setup process if the .cab file is copied to an unsupported device. For example:

```
[CEDevice]
UnsupportedPlatforms = pltfrm1      ; pltfrm1 is unsupported
[CEDevice.SH3]
UnsupportedPlatforms =              ; pltfrm1 is still unsupported
```

major_version and *minor_version*
Numeric value that is returned by **OSVERSIONINFO.***dwVersionMinor* and **OSVERSIONINFO.***dwVersionMajor*. The .cab file is valid for the currently connected device if the version of the currently connected device is less than or equal to *VersionMax* and also greater than or equal to *VersionMin*. For Windows CE Japanese-language devices, set *VersionMin* and *VersionMax* to 2.01.

Note The supported Windows CE OS versions include 1.0, 1.01, 2.0, 2.01, and 2.10. When you use these numbers, be sure to include all significant digits.

---

build_number
Numeric value returned by **OSVERSIONINFO.***dwBuildNumber*. The .cab file is valid for the currently connected device if the version of the currently connected device is less than or equal to *BuildMax* and also greater than or equal to *BuildMin*.

The following code example shows three [CEDevice] sections: one that gives basic information for any CPU and two that are specific to the SH3 and the MIPS microprocessors.

```
[CEDevice]                          ; A "template" for all platforms
UnsupportedPlatforms = pltfrm1      ; Does not support pltfrm1
; The following specifies version 1.0 devices only.
VersionMin = 1.0
VersionMax = 1.0
```

```
[CEDevice.SH3]                  ; Inherits all [CEDevice] settings
; This will create a .cab file specific to SH3 devices.
ProcessorType = 10003           ; The SH3 .cab file is only valid
                                ; for the SH3 microprocessors.
UnsupportedPlatforms =          ; pltfrm1 is still unsupported
; The following overrides the version settings so that no version
; checking is performed.
VersionMin =
VersionMax =

[CEDevice.MIPS]                  ; Inherits all [CEDevice] settings
; This will create a .cab file specific to "MIPS" devices.
ProcessorType = 4000             ; The MIPS .cab file is only valid
                                 ; for the MIPS microprocessor.
UnsupportedPlatforms =pltfrm2    ; pltfrm1 and pltfrm2 are
                                 ; unsupported for the "MIPs" .cab file.
```

Note To create the two CPU-specific .cab files for the setup .inf file in the previous example, you must run the CAB Wizard with the **/cpu sh3 mips** parameter.

DefaultInstall

The [DefaultInstall] section is required and describes the default installation of your application.

```
[DefaultInstall]
Copyfiles=copyfile_list_section[,copyfile_list_section]
AddReg=add_registry_section[,add_registry_section]
[CEShortcuts=shortcut_list_section[,shortcut_list_section]]   ; new key
[CESetupDLL=setup_DLL]                                        ; new key
[CESelfRegister=self_reg_DLL_filename[,self_reg_DLL_filename] ; new key
```

shortcut_list_section
String that identifies one more section that defines shortcuts to a file, as defined in the [CEShortcuts] section.

setup_DLL
Optimal string that specifies a Setup.dll. It is written by the independent software vendor (ISV) and contains customized functions for operations during installation and removal of the application. The file must be specified in the [SourceDisksFiles] section.

self_reg_DLL_filename
String that identifies files that self-register by exporting the **DllRegisterServer** and **DllUnregisterServer** Component Object Model (COM) functions. You must specify the files in the [SourceDiskFiles] section.

During installation, if installation on the device fails to call the file's exported **DllRegisterServer** function, the file's exported **DllUnregisterServer** function will not be called during removal.

The following code example shows a typical [DefaultInstall] section.

```
[DefaultInstall]
AddReg = RegSettings.All
CEShortcuts = Shortcuts.All
```

SourceDiskNames

The [SourceDiskNames] section is required and describes the name and path of the disk on which your application resides.

```
[SourceDisksNames]
disk_ordinal= ,disk_label,,path
[disk_ordinal= ,disk_label,,path]
```

The following code example shows a typical [SourceDiskNames] section.

```
[SourceDisksNames]                  ; Required section
1 = ,"Common files",,C:\app\common  ; Using an absolute path

[SourceDisksNames.SH3]
2 = ,"SH3 files",,sh3               ; Using a relative path

[SourceDisksNames.MIPS]
2 = ,"MIPS files",,mips             ; Using a relative path
```

SourceDiskFiles

The [SourceDiskFiles] section is required and describes the name and path of the files in which your application resides.

```
 [SourceDisksFiles]
filename=disk_number[,subdir]
[filename=disk_number[,subdir]]
```

The following code example shows a typical [SourceDiskFiles] section.

```
[SourceDisksFiles]          ; Required section
begin.wav = 1
end.wav = 1
sample.hlp = 1

[SourceDisksFiles.SH3]
sample.exe = 2              ; Uses the SourceDisksNames.SH3
                            ; identification of 2.

[SourceDisksFiles.MIPS]
sample.exe = 2              ; Uses the SourceDisksNames.MIPS
                            ; identification of 2.
```

DestinationDirs

The [DestinationDirs] section is required and describes the names and paths of the destination directories for your application on the target device.

```
[DestinationDirs]
file_list_section = 0,subdir
[file_list_section = 0,subdir]
[DefaultDestDir=0,subdir]
```

Note Windows CE does not support directory identifiers.

subdir

String that identifies the destination directory. The following table shows the string substitutions that are supported by Windows CE. These can be used only for the beginning of the path.

String	Replacement value
%CE1%	**\Program Files**
%CE2%	**\Windows**
%CE3%	**\Windows\Desktop**
%CE4%	**\Windows\Startup**
%CE5%	**\My Documents**
%CE6%	**\Program Files\Accessories**
%CE7%	**\Program Files\Communication**
%CE8%	**\Program Files\Games**
%CE9%	**\Program Files\Pocket Outlook**
%CE10%	**\Program Files\Office**

String	Replacement value
%CE11%	\Windows\Programs
%CE12%	\Windows\Programs\Accessories
%CE13%	\Windows\Programs\Communications
%CE14%	\Windows\Programs\Games
%CE15%	\Windows\Fonts
%CE16%	\Windows\Recent
%CE17%	\Windows\Favorites

The following code example shows a typical [DestinationDirs] section.

```
[DestinationDirs]
Files.Common    = 0,%CE1%\My Subdir    ;\Program Files\My Subdir
Files.Shared    = 0,%CE2%              ;\Windows
```

CopyFiles

The [Copyfiles] section, under the [DefaultInstall] section, is required and describes the default files to copy to the target device.

```
[copyfile_list_section]
destination_filename,[source_filename],[,flags]
[destination_filename,[source_filename],[,flags]]
```

The *source_filename* parameter is optional if it is the same as *destination_filename.*

flags

Numeric value that specifies an action to be done while copying files. The following table shows the values that are supported by Windows CE.

Flag	Value	Description
COPYFLG_WARN_IF_SKIP	**0x00000001**	Warn a user if an attempt is made to skip a file after an error has occurred.
COPYFLG_NOSKIP	**0x00000002**	Do not allow a user to skip copying a file.
COPYFLG_NO_OVERWRITE	**0x00000010**	Do not overwrite an existing file in the destination directory.

Flag	Value	Description
COPYFLG_REPLACEONLY	**0x00000400**	Copy the source file to the destination directory only if the file is already in the destination directory.
CE_COPYFLG_NO_DATE_DIALOG	**0x20000000**	Do not copy files if the target file is newer.
CE_COPYFLG_NODATECHECK	**0x40000000**	Ignore date while overwriting the target file.
CE_COPYFLG_SHARED	**0x80000000**	Create a reference when a shared DLL is counted.

The following example is a typical [CopyFiles] section.

```
[DefaultInstall.SH3]
CopyFiles = Files.Common, Files.SH3

[DefaultInstall.MIPS]
CopyFiles = Files.Common, Files.MIPS
```

AddReg

The [AddReg] section, under the [DefaultInstall] section, is required and describes the keys and values that the .cab file adds to the device registry.

```
[add_registry_section]
registry_root_string , subkey,[value_name], flags, value[,value]
[registry_root_string, subkey,[value_name], flags, value[,value]]
```

registry_root_strings

String that specifies the registry root location. The following table shows the values that are supported by Windows CE.

Root string	Description
HKCR	The same as **HKEY_CLASSES_ROOT**
HKCU	The same as **HKEY_CURRENT_USER**
HKLM	The same as **HKEY_LOCAL_MACHINE**

value_name

Registry value name. If empty, the **"(default)"** registry value name is used.

flags

Numeric value that specifies information about the registry key. The following table shows the values that are supported by Window CE.

Flag	Value	Description
FLG_ADDREG_NOCLOBBER	**0x00000002**	If the registry key exists, do not overwrite it. This flag can be used in combination with any of the other flags in this table.
FLG_ADDREG_TYPE_SZ	**0x00000000**	The **REG_SZ** registry data type.
FLG_ADDREG_TYPE_MULTI_SZ	**0x00010000**	The **REG_MULTI_SZ** registry data type. The value field that follows can be a list of strings separated by commas.
FLG_ADDREG_TYPE_BINARY	**0x00000001**	The **REG_BINARY** registry data type. The value field that follows must be a list of numeric values separated by commas, one byte per field, and must not use the 0x hexadecimal prefix.
FLG_ADDREG_TYPE_DWORD	**0x00010001**	The **REG_DWORD** data type. Only the non-compatible format in the Win32 Setup .inf documentation is supported.

The following code example shows a typical [AddReg] section.

```
AddReg = RegSettings.All

[RegSettings.All]
HKLM,%reg_path%,,0x00000000,alpha         ; <default> = "alpha"
HKLM,%reg_path%,test,0x00010001,3         ; Test = 3
HKLM,%reg_path%\new,another,0x00010001,6 ; New\another = 6
```

CEShortcuts

The [CEShortcuts] section, a Windows CE–specific section under the [DefaultInstall] section, is optional and describes the shortcuts that the installation application creates on the device.

```
[shortcut_list_section]
shortcut_filename,shortcut_type_flag,target_file/path[,standard_destinat
ion_path]
[shortcut_filename,shortcut_type_flag,target_file/path[,standard_destina
tion_path]]
```

shortcut_filename
String that identifies the shortcut name. It does not require the .lnk extension.

shortcut_type_flag
Numeric value. Zero or empty represents a shortcut to a file; any nonzero numeric value represents a shortcut to a folder.

target_file/path
String value that specifies the destination location. For a file, use the target file name—for example, MyApp.exe—that must be defined in a file copy list. For a path, use a *file_list_section* name defined in the [DestinationDirs] section—for example, **DefaultDestDir**—or the *%InstallDir%* string.

standard_destination_path
Optional string value. A standard **%CE***x***%** path or *%InstallDir%*. If no value is specified, the *shortcut_list_section* name of the current section or the **DefaultDestDir** value from the [DestinationDirs] section is used.

The following code example shows a typical [CEShortcuts] section.

```
CEShortcuts = Shortcuts.All

[Shortcuts.All]
Sample App,0,sample.exe              ; Uses the path in DestinationDirs.
Sample App,0,sample.exe,%InstallDir% ; The path is explicitly specified.
```

Sample .inf File

The following code example shows a typical .inf file.

```
[Version]                  ; Required section
Signature = "$Windows NT$"
Provider = "Microsoft"
CESignature = "$Windows CE$"

[CEDevice.SH3]
ProcessorType = 10003      ; SH3 microprocessor
```

```
[CEDevice.MIPS]
ProcessorType = 4000        ; MIPS microprocessor

[DefaultInstall]            ; Required section
AddReg = RegSettings.All
CEShortcuts = Shortcuts.All

[DefaultInstall.SH3]
CopyFiles = Files.Common, Files.SH3

[DefaultInstall.MIPS]
CopyFiles = Files.Common, Files.MIPS

[SourceDisksNames]                  ; Required section
1 = ,"Common files",,C:\app\common   ; Using an absolute path

[SourceDisksNames.SH3]
2 = ,"SH3 files",,sh3       ; Using a relative path

[SourceDisksNames.MIPS]
2 = ,"MIPS files",,mips     ; Using a relative path

[SourceDisksFiles]          ; Required section
begin.wav = 1
end.wav = 1
sample.hlp = 1

[SourceDisksFiles.SH3]
sample.exe = 2              ; Uses the SourceDisksNames.SH3
                            ; identification of 2.

[SourceDisksFiles.MIPS]
sample.exe = 2              ; Uses the SourceDisksNames.MIPS
                            ; identification of 2.

[DestinationDirs]           ; Required section
Shortcuts.All = 0,%CE3%     ; \Windows\Desktop
Files.Common = 0,%CE2%      ; \Windows
Files.SH3 = 0,%InstallDir%
Files.MIPS = 0,%InstallDir%
DefaultDestDir = 0,%InstallDir%

[CEStrings]                 ; Required section
AppName = My Test App
InstallDir = %CE1%\%AppName%

[Strings]                   ; Optional section
reg_path = Software\Microsoft\My Test App
```

```
[Shortcuts.All]
Sample App,0,sample.exe              ; Uses the path in DestinationDirs.
Sample App,0,sample.exe,%InstallDir% ; The path is explicitly specified.

[Files.Common]
begin.wav,,,0
end.wav,,,0
Sample Help File.hlp,sample.hlp,,0   ; Rename the destination file.

[Files.SH3]
sample.exe,,,0

[Files.MIPS]
sample.exe,,,0

[RegSettings.All]
HKLM,%reg_path%,,0x00000000,alpha    ; <default> = "alpha"
HKLM,%reg_path%,test,0x00010001,3    ; test = 3
HKLM,%reg_path%\new,another,0x00010001,6 ; new\another = 6
```

Using Installation Functions in Setup.dll

Setup.dll is an optional file that enables you to perform custom operations during installation and removal of your application. The following table shows the functions that are exported by Setup.dll.

Function	Description
Install_Init	Called before installation begins. Use this function to check the application version when reinstalling an application and to determine if a dependent application is present.
Install_Exit	Called after installation is complete. Use this function to handle errors that occur during application installation.
Uninstall_Init	Called before the removal process begins. Use this function to close the application, if the application is running.
Uninstall_Exit	Called after the removal process is complete. Use this function to save database information to a file and delete the database and to tell the user where the user data files are stored and how to reinstall the application.

Note Use the [CESelfRegister] section in the .inf file to point to Setup.dll.

Using CAB Wizard to Create a .cab File

After you create the .inf file and the optional Setup.dll file, use the CAB Wizard to create the .cab file. The command-line syntax for the CAB Wizard is as follows:

```
cabwiz.exe "inf_file" [/dest dest_directory] [/err error_file]
[/cpu cpu_type [cpu_type]]
```

inf_file
Setup .inf file path.

dest_directory
Destination directory for the .cab files. If no directory is specified, the .cab files are created in the *inf_file* directory.

error_file
File name for a log file that contains all warnings and errors that are encountered when the .cab files are compiled. If no file name is specified, errors are displayed in message boxes. If a file name is used, the CAB Wizard runs without the user interface (UI); this is useful for automated builds.

cpu_type
Creates a .cab file for each microprocessor tag that you specify. A microprocessor tag is a label that is used in the Win32 setup .inf file to differentiate between different microprocessor types. The **/cpu** parameter, followed by multiple *cpu_type* values, must be the last qualifier in the command line.

The following example creates .cab files for the SH3 and MIPS microprocessors, assuming that the Win32 setup .inf file contains the SH3 and MIPS tags:

```
cabwiz.exe  "c:\myfile.inf"  /err myfile.err  /cpu  sh3  mips
```

Note The following Windows CE files must be installed in the same directory on the desktop computer: Cabwiz.exe, Makecab.exe, and Cabwiz.ddf. Cabwiz.exe must be called with its full path in order to run correctly.

Troubleshooting the CAB Wizard

To identify and avoid problems that might occur when using the CAB Wizard, follow these guidelines:

- Use %% for a percent sign (%) character when using this character in an .inf file string, as specified in the Win32 documentation. This will not work under the [Strings] section.
- Do not use .inf or .cab files that were created for Windows CE to install applications on Windows-based desktop platforms.

- Ensure that the Makecab.exe and Cabwiz.ddf files, included with Windows CE, are in the same directory as Cabwiz.exe.
- Use the full path to call Cabwiz.exe.
- Do not create a .cab file with the Makecab.exe file that is included with Windows CE. You must use Cabwiz.exe, which uses Makecab.exe to generate the .cab files for Windows CE.
- Do not set the read-only attribute for .cab files.

Using the Application Manager

The Application Manager application, CeAppMgr.exe, resides on a user's desktop computer. CeAppMgr adds and removes applications on a Windows CE–based device, and it deletes the application files from the desktop computer.

▶ **To install an application on a Windows CE–based device**

1. Create a single Application Manager initialization (.ini) file to provide the Application Manager with information about the application.
2. Install the application automatically or manually:
   - To install the application automatically, create a desktop setup application with any available third-party desktop setup application.

     This application copies the multiple device-specific .cab files to the desktop computer. The application then launches the Application Manager, with the Application Manager .ini file as a parameter.

     –Or–
   - To install the application manually, register the application with the Application Manager.

     Registering the application manually requires that you copy the .cab file and the .ini file for the Application Manager to the desktop computer and run the Application Manager with the .ini file as the parameter.

Creating an .ini File for the Application Manager

The .ini file contains information that registers an application with the Application Manager. The .ini file has the following format:

```
[CEAppManager]
Version      = version_number
Component    = component_name

[component_name]
Description  = descriptive_name
[Uninstall   = uninstall_name]
```

```
[InstallDir  = install_directory]
[IconFile    = icon_filename]
[IconIndex   = icon_index]
[DeviceFile  = device_filename]
CabFiles     = cab_filename [,cab_filename]
```

version_number
Numeric version of the Application Manager, which is 1.0.

component_name
String that identifies the name of the section for the application.

descriptive_name
String that will appear in the **Description** field of the Application Manager when a user chooses the application.

uninstall_name
String that identifies the application's Windows **Uninstall** registry key name. This name must match the application's registered Windows **Uninstall** key name, which is found in the **HKLM\Software\Microsoft\Windows \CurrentVersion\Uninstall** registry key. Providing this name enables the Application Manager to automatically remove the application from the desktop computer and the device when a user clicks the **Remove** button in the Application Manager UI.

install_directory
String that identifies the desktop installation directory containing the location of the .cab files. If this key is nonexistent, which is recommended, the path of the .inf file is used for the installation directory.

icon_filename
String that identifies the relative path from *install_directory* to the desktop icon file. This string is used to display the *device_filename* when the file name is viewed in Windows CE Services.

icon_index
Numeric index into *icon_filename*. The value is used to display the *device_filename* when it is viewed in Windows CE Services. If this key is nonexistent, the first icon in *icon_filename* is used.

device_filename
File name on the device that will display the icon specified by *icon_filename* and *icon_index* when the *device_filename* is viewed in Windows CE Services.

cab_filename
File name of the available .cab files, relative to *install_directory*. Use commas to separate multiple *cab_filenames*. Do not include unnecessary spaces in this list of file names.

Sample .ini File

The following code example shows a typical.ini file.

```
[CEAppManager]
Version      = 1.0
Component    = Games

[Games]
Description  = Game Pack for your Windows CE-based device
Uninstall    = Game Pack

;Do not specify the "InstallDir" key so that CEAppMgr will use
;the directory of this .ini file as the installation directory.

IconFile     = gamepack.ico
IconIndex    = 0
DeviceFile   = gamepack.exe

;Because there are multiple .cab files specific to a CPU type,
;these files are relative to the installation directory.
CabFiles= SH3\gamepack.cab,MIPS\gamepack.cab
```

Installing an Application Automatically

You can use a third-party desktop computer installation application to copy a file from the installation site to a user's desktop computer. With this approach, the Application Manager automatically installs the application on the Windows CE–based device. If the Windows CE–based device is not connected, the Application Manager notes that the application has not been installed. When the device is subsequently connected, the Application Manager automatically completes the installation.

You can extract the full file name and path of the Application Manager from the default registry value of the **HKLM\Software\Microsoft\Windows\CurrentVersion\App Paths\CEAppMgr.exe** registry key. Because the returned value is the full file name and path of CEAppMgr.exe, you can remove the CEAppMgr.exe file name to get the desktop installation directory of Windows CE Services. You can use the desktop installation directory to copy files to the desktop computer. The location for your files will be the installation directory with your application's subdirectory appended.

Installing and Removing an Application Manually

To install an application from a desktop computer to a Windows CE–based device, on the desktop computer, call the Application Manager with the application's .ini file as a parameter. The command-line syntax for the Application Manager CeAppMgr.exe is as follows:

```
CEAppMgr.exe [/report]  "CEAppMgr_ini_filename"
["CEAppMgr_ini_filename"]
```

CEAppMgr_ini_filename
Full file name and path of the CEAppMgr .ini file for a single application. If the application has multiple components, you can run the Application Manager once with multiple .ini files, one file for each component.

report
Optional parameter that you can use to test the installation process if problems occur. This parameter should not be included in the final setup application.

One you call CEAppMgr.exe, the Application Manager completes the installation process. To add additional capabilities to the installation process, use the **Install_Init** and **Install_Exit** functions. Because the installation procedure registers the application's .cab files with the Application Manager, a user can reinstall the application on the device at a later time or install the application on another device.

The full application name displayed in the **CEAppMgr** dialog box is extracted from the CAB Wizard .inf file. The extracted name is "*provider appname*". The value for *provider* is from [Version] Provider, while *appname* is from the [CEStrngs] AppName.

To remove an application from a Windows CE–based device, the user calls the Application Manager from the desktop computer. The Application Manager is usually located in the Control Panel. The Application Manager uses the information registered from the *uninstall_name* parameter of the .ini file to delete the application from the desktop computer and the Windows CE–based device. To add additional capabilities to the removal process, use the **Uninstall_Init** and **Uninstall_Exit** functions.

Troubleshooting the Application Manager

To identify and avoid problems that might occur when you install or remove an application on a Windows CE–based device, follow these guidelines:

- Use the full path for the location of the CeAppMgr .ini file when you call Ceappmgr.exe to register an application.
- Use the */report* parameter in debug versions to verify that CeAppMgr is using the correct information for the .cab files.
- In the CeAppMgr .ini file, verify the following:
  - The string list in the CabFiles key contains no unnecessary spaces and matches the actual .cab file name and relative path.
  - The string value in the Component key exists elsewhere in the .ini file.
- Verify that the desktop computer's setup application is calling the correct CeAppMgr .ini file, using the full path.

There are various third-party desktop setup applications that do not correctly update the actual file sizes when overwriting existing files. Because the Application Manager verifies the actual file size with the embedded file size of the .cab file, be sure that the installed .cab file sizes are correct. To ensure that this happens for future upgrade scenarios, delete the known existing .cab files when you reinstall an application.

Adding Custom Menus to Windows CE Explorer

You can add additional menu items to the **Tools** menu in the Windows CE Explorer window in two different ways. The method described here uses code to directly place values in the proper registry locations. You can also use the CEUTIL utility DLL, described in "Using the CEUTIL Helper DLL for Windows CE Services," to create custom menus and perform other tasks.

To add a custom menu, create a subkey and add several values under **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\ CustomMenus** as follows.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WINDOWS CE Services\CustomMenus]
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WINDOWS
CE Services\CustomMenus\subkey]
"DisplayName"="displayName"
"Command"="myApp.exe"
"StatusHelp"="StatusHelpText"
"Version"=version_number
```

subkey
String that identifies the subkey to be created on the **Tools** menu.

displayName
String that identifies the display name of the menu item. An ampersand (&) specifies a hot key.

myApp.exe
String that identifies the command that will be executed by **WinExec** when a user chooses the menu item.

StatusHelpText
String that identifies the status and Help text that appears in the status bar when a user browses the menu item.

version_number
Application version. This value should be **0x00020000**.

The following sample registry file adds a calculator menu item.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WINDOWS CE Services\CustomMenus]
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WINDOWS
CE Services\CustomMenus\MyApp]
"DisplayName"="&My Calculator"
"Command"="calc.exe"
"StatusHelp"="Brings up the calculator"
"Version"=dword:00020000
```

PART 3

Writing Applications for a Global Market

This part contains the following chapters:

- **Programming and Designing a Global Application**
- **Programming with Unicode and NLS**
- **Working with the Input Method Editor**

CHAPTER 11

Programming and Designing a Global Application

To compete successfully in international markets, your software must easily accommodate differences in language, culture, and hardware. The most effective way to accomplish this is to take international considerations into account at the beginning of the product cycle and throughout development. By planning ahead, you can create software in a single effort that accommodates multiple languages, instead of just one.

The process of developing an application whose features and code designs do not make assumptions based on a single language or locale and whose source code simplifies the creation of different language editions of an application is known as *globalization*. The process of creating globalized software is divided into two areas—internationalization, which covers generic coding and design issues, and *localization*, which involves translating and customizing a product for a specific market.

This section focuses on internationalization and, specifically, the issues that you must consider when designing the code and user interface for a global application.

Internationalizing Software

The goal of internationalization is to present users with a consistent look, feel, and functionality across different language editions of a product. Users expect localized software to support the same basic set of features that the original language edition of the product does, and they expect it to achieve the same level of quality. They also expect different language editions to interact smoothly with one another.

Internationalizing software involves designing a user interface and a code base that are generic enough to work for most of the product's intended language editions. Of course, some customization may be necessary, but the fewer changes needed for international editions, the faster you can release the product.

A key prerequisite to creating an internationalized code base for your application is that all language editions share the same source files. Maintaining separate source files for different language editions of the same product is error-prone, a waste of time and disk space, and unnecessary for code that is properly internationalized.

The process of coding an application that supports multiple language editions requires you to perform three tasks:

- Create a consistent user interface that accommodates language changes.
- Support international characters, as well as international date, time, currency, and numeric formatting and sorting conventions in your code.
- Implement smart coding practices that save time and money during the localization process.

Creating an International User Interface

A major aspect of creating an international user interface involves translating the text used in title bars, menus, other controls, messages, and registry entries. To make this process easier, store interface text as resources in your application's resource file, rather than including it in the source code of the application. Also translate any menu commands that your application stores for its file types in the system registry.

When translating text, remember that each language has its own syntax and grammar. The following are some general guidelines to keep in mind when translating text:

- Avoid using vague words that can have several meanings in different contexts.
- Avoid colloquialisms, jargon, acronyms, and abbreviations.
- Use good grammar. Translation is a difficult task even when a translator does not have to deal with poor grammar.
- Avoid dynamic, or run-time, concatenation of different strings to form new strings—for example, composing messages by combining frequently used strings. An exception is the construction of file names and names of paths.
- Avoid hard-coding file names in a binary file. File names may need to be translated.
- Avoid including text in images and icons. Doing so requires that these also be translated.

Translation of interface text often increases the length of text by 30 percent or more. In some extreme cases, the character count can increase by more than 100 percent; for example, the English word "move" becomes "verschieben" in German. Accordingly, if the amount of space for displaying text is strictly limited, as in a status bar, restrict the length of the interface text to approximately one-half of the available space. In contexts that allow more flexibility, such as dialog boxes and property sheets, allow 30 percent for text expansion in the interface design. Text in message boxes, however, should allow for text expansion of about 100 percent. Avoid having your software rely on the position of text in a control or window because translation may require movement of the text.

Expansion due to translation affects other aspects of your product. A localized version is likely to affect file sizes, which potentially can change the layout of your installation disks and setup software.

Additionally, translation is not always a one-to-one correspondence. A single word can have multiple translations in another language. Adjectives and articles sometimes change their spelling according to the gender of the nouns they modify. Therefore, be careful when reusing a string in multiple places. Similarly, several words may have only a single meaning in another language. This is particularly important when creating keywords for the Help index for your software.

Adhering to International Conventions

When you are internationalizing a user interface, language is not the only factor to consider. Several countries can share a common language but have different conventions for expressing information. In addition, some countries can share a language but use different keyboard conventions.

International keyboards differ. Avoid using punctuation character keys as shortcut keys because they are not always found on international keyboards or easily produced by the user. What seems like an effective shortcut because of its mnemonic association—for example, CTRL+B for Bold—may need to be changed to fit a particular language. Similarly, macros or other utilities that invoke menus or commands based on access keys are not likely to work in an international version because the command names on which the access keys are based differ.

Additionally, keys do not always occupy the same positions on all international keyboards. Even when they do, the interpretation of the unmodified keystroke can be different. For example, on US keyboards, SHIFT+8 results in an asterisk character. However, on French keyboards, it generates the number 8. Similarly, avoid using CTRL+ALT combinations, because the system interprets this combination for some language versions as the ALTGR key, which generates alphanumeric characters. Similarly, avoid using the ALT key as a modifier because it is the primary keyboard interface for accessing menus and controls. In addition, the system uses many specialized versions for special input. For example, ALT+~ invokes special input editors in Asian versions of Windows. For text fields, pressing ALT+number enters characters in the upper range of a character set. Similarly, avoid using the following characters when assigning shortcut keys.

@ £ $ { } [ ] \ ~ | ^ ' < >

Preparing for Cultural Differences

A more subtle factor to consider when you are preparing software for international markets is cultural differences. For example, users in the US may recognize a rounded mail box with a flag on the side as an icon for a mail program, but this image may not be recognized by users in other countries. Sounds and their associated meanings may also vary from country to country.

It is best to review the proposed graphics for international applicability early in your design cycle. Localizing graphics can be a time-consuming process.

Although graphics communicate more universally than text, graphical aspects of your software—especially icons and toolbar button images—may also need to be revised to address an international audience. For example, a toolbar image that includes a magic wand to represent access to a wizard interface does not have meaning in many countries and requires a different image.

When possible, choose generic images and glyphs. Even if you can create custom designs for each language, having different images for different languages can confuse users who work with more than one language version.

Many symbols with a strong meaning in one culture do not have any meaning in another. For example, many symbols for holidays and seasons are not shared around the world. Importantly, some symbols can be offensive in some cultures; for example, the open palm commonly used at US crosswalk signals is offensive in some countries. Some metaphors also may not apply in all languages.

Supporting International Characters and Formatting

Character encoding is the most basic foundation for any form of text processing; if it is handled poorly, the software is difficult to localize or internationalize. A program also requires functionality that observes language rules and cultural conventions. Windows CE provides support for numerous character codes as well as linguistic and cultural conventions through Unicode and national language support (NLS). Unicode is a universal character encoding system, while NLS carries information on date, time, calendar, number, and currency formats. NLS also provides sorting and character-type information for all the locales supported by the operating system. For more information on how to use Unicode and NLS when creating a global application, see "Programming with Unicode and NLS."

Coding for Internationalization

When you are coding your application, several coding practices can make the internationalization process easier. A few of these practices are the following:

- Do not hard-code localizable elements.

  Hard-coded strings, characters, constants, screen positions, file names, and file paths are difficult to track down and localize. Isolate all localizable items into resource files, and minimize compile dependencies.

- Do not make buffers too small to handle localized text.

  Buffers that are declared to be the exact size of a word or a sentence will probably overflow when text is translated. Consider the following example. Your application declares a 2-byte buffer size for the word "OK." In Spanish, however, when it refers to the text in an OK button, the same word is translated as "Aceptar," which would cause your application to overflow.

- Do not perform string composition.

  For example, translating "wrong file" and "wrong directory" to Italian results in "file errat*o*" and "cartella erratt*a*," respectively. If you try to perform string composition using the syntax "wrong%s", it does not work.

Another potential problem involves declaring a single string and displaying it in a number of different contexts: on a menu, in a dialog box, and perhaps in several messages. The problem with using all-purpose strings is that in European languages, adjectives and some nouns have from 4 to 14 different forms, such as masculine, feminine, and neuter singular; and masculine, feminine, and neuter plural, that must match the nouns they modify. A single string displayed in different contexts is correct in gender and number in some cases but incorrect in others.

One way to ensure that your coding practices works in an international market is to substitute your language strings with a pseudolanguage, and then test your code. Any potential problems should surface immediately.

CHAPTER 12

Programming with Unicode and NLS

Windows CE includes national language support (NLS) APIs, as well as Unicode, to assist you in creating global applications.

NLS functions help Windows CE–based applications to support the differing language-specific and location-specific needs of users around the world. These functions enable you to specify a locale so that you can correctly display times, dates, and other language-specific and location-specific information in your application. NLS also includes support for different keyboard layouts and language-specific fonts.

Unicode is a worldwide character-encoding standard that treats all characters as having a fixed width of 2 bytes. It can represent all the world's characters in modern computer use, including technical symbols and special characters used in publishing. Because Unicode characters are 2 bytes, Unicode-enabled functions are often referred to as wide-character functions.

Unicode defines semantics for each character, standardizes script behavior, provides a standard algorithm for bidirectional text, and defines cross-mappings to other standards. Because each Unicode character is 16-bits wide, it is possible to have separate values for up to 65,536 characters.

Windows CE uses Unicode exclusively at the system level for character and string manipulation. By implementing Unicode in your applications, you can provide your application with universal data exchange capabilities for global marketing, using a single binary file for every possible character code. This simplifies localization of software and improves multilingual text processing.

The following sections provide an overview of the Unicode standard, and then explain how to use NLS functions in Windows CE–based applications.

Understanding the Unicode Standard

The Unicode standard defines codes for characters in most major languages written today. Scripts include Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian, Tibetan, Japanese Kana, the complete set of modern Korean Hangul, and a unified set of Chinese/Japanese/Korean (CJK) ideographs. There are also several other scripts that have recently been added, including Ethiopic, Canadian Syllabics, Cherokee, Sinhala, Syriac, Burmese, Khmer, and Braille.

The Unicode standard also includes punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, and dingbats. It supports diacritics, which are character marks such as the tilde (~). Diacritics are used in conjunction with base characters to encode accented or vocalized letters; for example, ñ. In all, the Unicode standard provides codes for nearly 39,000 characters from the world's alphabets, ideograph sets, and symbol collections.

In addition, there are approximately 18,000 unused code values that have been reserved for future use. The Unicode standard also contains 6,400 code values that software and hardware developers can assign internally for their own characters and symbols.

Defining a Character Set

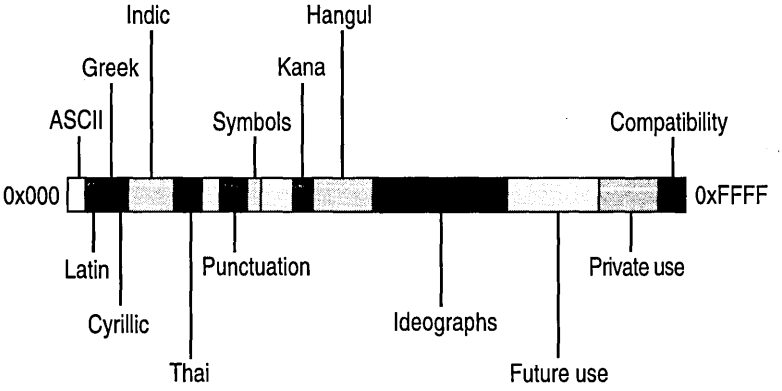
Written languages are represented by textual elements—called *code elements* or characters—that are used to create words and sentences. These elements can be letters such as *s* or *V,* characters such as those used in Japanese Hiragana to represent syllables, or ideographs such as those used in Chinese to represent full words or concepts.

A code element is an abstract concept, defined as the smallest component of a written language that has semantic value. A single 16-bit number is assigned to each code element defined by the Unicode standard. Each of these 16-bit numbers is called a *code value* and, when referred to in text, is listed in hexadecimal form following the prefix *U.* For example, the code value U+0041 is the hexadecimal number 0041, which is equal to the decimal number 65. It represents the character A in Unicode.

Each code element is also assigned a unique name that specifies it and no other. For example, U+0041 is assigned the character name LATIN CAPITAL LETTER A. U+0A1B is assigned the character name GURMUKHI LETTER CHA.

Code elements are grouped logically throughout the range of code values, which is called the *codespace*. The coding begins at U+0000 with standard ASCII characters, and then continues with Greek, Cyrillic, Hebrew, Arabic, Indic, and other scripts. Then symbols and punctuation are inserted, followed by Hiragana, Katakana, and Bopomofo. The complete set of modern Hangul appears next, followed by the unified ideographs. The end of the codespace contains code values that are reserved for further expansion, private use, and a range of compatibility characters.

The following illustration shows Unicode's encoding layout.



The Unicode standard defines how characters are interpreted. It is not responsible for rendering characters on screen or paper. The software or hardware is responsible for the appearance of the characters on the screen or in print. For example, the character identified by a Unicode code value as BENGALI DIGIT 5 is an abstract entity. The mark made on the screen or paper—called a glyph—is a visual representation of the character. The Unicode standard does not define the glyph image. It does not specify the size, shape, or orientation of the character. It simply defines how the character is interpreted by the software or device.

Occasionally, you may choose to render multiple characters together. This is referred to as creating a composite character. For example, "â" is a composite character created by rendering "a" and "^" together. A composite character is typically made up of a base letter, which occupies a single space, and one or more non-spacing marks, which are rendered in the same space as the base letter.

The Unicode standard specifies the order of characters used to create a composite character. The base character comes first, followed by one or more non-spacing marks. If a code element is encoded with more than one non-spacing mark, you can render the non-spacing marks in any order as long as the marks do not interact typographically. If they do interact, the order must be considered. The Unicode standard specifies how competing non-spacing characters are applied to a base character.

As an alternative to rendering your own composite characters, the Unicode standard offers precomposed characters to retain compatibility with established standards such as Latin 1, which includes many precomposed characters such as "ü" and "ñ". Each precomposed character is represented by a single code value, rather than two or more code values which may combine during rendering. For example, the character "ü" can be encoded as the single code value U+00FC "ü" or as the base character U+0075 "u" followed by the non-spacing character U+0308 "¨".

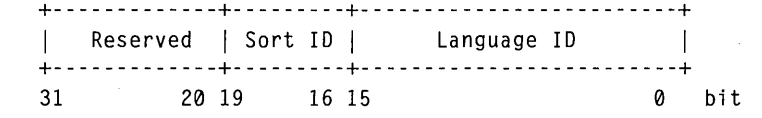
Precomposed characters may also be decomposed. For example, an application importing a text file containing the precomposed character "ü" may decompose that character into a "u" followed by the non-spacing character "¨". This allows easy alphabetical sorting for languages where character modifiers do not affect alphabetical order. The Unicode standard defines decomposition for all precomposed characters.

Specifying Locales with NLS

The national language support functions enable you to target an application for a specific locale. A locale is a collection of language-related user preference information represented as a list of values. Each system has at least one installed locale and often has many locales from which the user can choose.

The system assigns a locale to each thread. Initially, the system assigns the system default locale to the thread. This default locale is set by the user when the system is installed or through the Regional Settings program in Control Panel. If a thread is run in a process belonging to a user, the system assigns the user-default locale to the thread. An application can override either default by using the **SetThreadLocale** function to explicitly set the locale for a thread.

Each locale has a unique locale identifier (LCID), which is a 32-bit value that consists of a language identifier and a sort identifier. The LCID is constructed using the **MAKELCID** macro. The following illustration shows the format of the bits in an LCID.

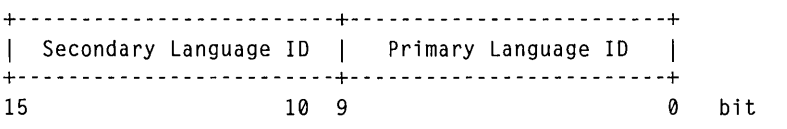


The following locale identifiers are predefined:

- LOCALE_SYSTEM_DEFAULT, which identifies the system default locale.
- LOCALE_USER_DEFAULT, which identifies the locale of the current user.
- LOCALE_NEUTRAL, which identifies the default language-neutral locale.

The LOCALE_NEUTRAL identifier is the same as LOCALE_USER_DEFAULT. An application can retrieve the current locale identifiers by using the **GetSystemDefaultLCID** and **GetUserDefaultLCID** functions.

A language identifier is a standard international numeric abbreviation for a country or geographical region. Each language has a unique language identifier (LANGID), which is a 16-bit value that consists of a primary language identifier and a secondary language identifier. The LANGID is constructed using the **MAKELANGID** macro. The following illustration shows the format of the bits in a LANGID.



The following language identifiers are predefined:

- LANG_SYSTEM_DEFAULT, which identifies the system default language.
- LANG_USER_DEFAULT, which identifies the language of the current user.

An application can retrieve the current language identifiers by using the **GetSystemDefaultLangID** and **GetUserDefaultLangID** functions.

It is often necessary to get specific information on available languages and locales in order to handle strings appropriately. Each element of locale information has a corresponding LCTYPE constant. To get the locale information, call the **GetLocaleInfo** function with the constant that corresponds to the information that is needed.

Most LCTYPE constants are mutually exclusive, so usually only one type of information can be retrieved at a time. The exceptions to this are LOCALE_NOUSEROVERRIDE, LOCALE_USE_CP_ACP, and LOCALE_RETURN_NUMBER, which can combined with other LCTYPE constants by using the binary **OR** operator.

Locale information is always stored and manipulated as a null-terminated string. No binary data is allowed; any numeric values must be specified as text. Each type of information has a particular format. Also, several of the types are linked together, so that changing one changes the value of the other as well.

Although a specified locale identifier may be supported, it is not available for use by an application unless it is also installed.

Retrieving Time and Date Strings

One of the most useful sets of NLS calls is the **GetDateFormat**, **GetTimeFormat**, **EnumDateFormats**, **EnumTimeFormats**, and **EnumCalendarInfo** functions, which return formatted date and time picture strings. **EnumDateFormats** and **EnumTimeFormats** enumerate the date and time picture strings that the system carries for a particular locale. With these two functions, you can build a list of possible date and time strings to present to the user.

GetDateFormat and **GetTimeFormat** each return a single string. When you use **GetDateFormat**, your application can request a string containing the correct date in the default short date format (DATE_SHORTDATE) or the default long date format (DATE_LONGDATE) for a particular locale.

For **GetTimeFormat**, the time values in the **SYSTEMTIME** structure pointed to by *lpTime* must be valid. The function checks each of the time values to determine that it is within the appropriate range of values. If any of the time values are outside the correct range, the function fails and sets the last error to ERROR_INVALID_PARAMETER.

The function ignores the date portions of the **SYSTEMTIME** structure pointed to by *lpTime*. If a time marker exists and the TIME_NOTIMEMARKER flag is not set, the function localizes the time marker, based on the specified locale identifier. Examples of time markers are "AM" and "PM" for US English, and "de." and "du." for Mexican Spanish.

The function does not return an error for a bad format string. The function simply forms the best time string that it can. If more than two hour, minute, second, or time marker format pictures are passed in, the function defaults to two.

For **GetDateFormat**, the date values in the **SYSTEMTIME** structure pointed to by *lpDate* must be valid. The function checks each of the date values: year, month, day, and day of week. If the day of the week is incorrect, the function uses the correct value and returns no error. If any of the other date values are outside the correct range, the function fails and sets the last error to ERROR_INVALID_PARAMETER.

The day name, abbreviated day name, month name, and abbreviated month name are all localized based on the specified locale identifier. The function ignores the time portions of the **SYSTEMTIME** structure pointed to by *lpDate*.

To obtain the short and long date format for the default locale calendar, use the **GetLocaleInfo** function with the LOCALE_SSHORTDATE or the LOCALE_SLONGDATE flag. To get the date format for an alternate calendar, use **GetLocaleInfo** with the LOCALE_IOPTIONALCALENDAR flag. To get the date format for a particular calendar, use **GetCalendarInfo**. To return all the date formats for a particular calendar, use the **EnumCalendarInfo** or the **EnumDateFormatsEx** function.

To obtain the time format without performing any actual formatting, use the **GetLocaleInfo** function with the LOCALE_STIMEFORMAT flag set.

Defining Calendar Formats

Most locales use the standard Gregorian calendar and a set number of date formats. These default choices for date formats are available for display by using the **EnumDateFormats** function. Other locales require special considerations when you are creating a complete list of format choices. Some of these require you to insert text strings within the date format string; others require a completely different method of computation of the values. These special requirements are addressed by the addition of certain LCTYPE and CALTYPE values.

Each LCID has a default calendar type associated with it. A locale identifier can also have an alternate calendar type. To have an alternate calendar type for an LCID, you must set LOCALE_IOPTIONALCALENDAR to the alternate calendar type for this locale.

The **CALTYPE** constants are used in the **EnumCalendarInfo** and **GetCalendarInfo** functions to define particular pieces of calendar information. Some of these types are also used for the **SetCalendarInfo** function.

The following table shows **CALTYPE** constants that are mutually exclusive—that is, they cannot be used in combination with each other in a function call.

Type	Description
CAL_ICALINTVALUE	An integer value indicating the calendar type of the alternate calendar.
CAL_IYEAROFFSETRANGE	One or more null-terminated strings that specify the year offsets for each of the era ranges. The last string has an extra terminating null character.
CAL_SABBREVDAYNAME1	Abbreviated local name of the first day of the week.
CAL_SABBREVDAYNAME2	Abbreviated local name of the second day of the week.
CAL_SABBREVDAYNAME3	Abbreviated local name of the third day of the week.

Type	Description
CAL_SABBREVDAYNAME4	Abbreviated local name of the fourth day of the week.
CAL_SABBREVDAYNAME5	Abbreviated local name of the fifth day of the week.
CAL_SABBREVDAYNAME6	Abbreviated local name of the sixth day of the week.
CAL_SABBREVDAYNAME7	Abbreviated local name of the seventh day of the week.
CAL_SABBREVMONTHNAME1	Abbreviated local name of the first month of the year.
CAL_SABBREVMONTHNAME2	Abbreviated local name of the second month of the year.
CAL_SABBREVMONTHNAME3	Abbreviated local name of the third month of the year.
CAL_SABBREVMONTHNAME4	Abbreviated local name of the fourth month of the year.
CAL_SABBREVMONTHNAME5	Abbreviated local name of the fifth month of the year.
CAL_SABBREVMONTHNAME6	Abbreviated local name of the sixth month of the year.
CAL_SABBREVMONTHNAME7	Abbreviated local name of the seventh month of the year.
CAL_SABBREVMONTHNAME8	Abbreviated local name of the eighth month of the year.
CAL_SABBREVMONTHNAME9	Abbreviated local name of the ninth month of the year.
CAL_SABBREVMONTHNAME10	Abbreviated local name of the tenth month of the year.
CAL_SABBREVMONTHNAME11	Abbreviated local name of the eleventh month of the year.
CAL_SABBREVMONTHNAME12	Abbreviated local name of the twelfth month of the year.
CAL_SABBREVMONTHNAME13	Abbreviated local name of the thirteenth month of the year, if it exists.
CAL_SCALNAME	Local name of the alternate calendar.
CAL_SDAYNAME1	Local name of the first day of the week.
CAL_SDAYNAME2	Local name of the second day of the week.
CAL_SDAYNAME3	Local name of the third day of the week.

Type	Description
CAL_SDAYNAME4	Local name of the fourth day of the week.
CAL_SDAYNAME5	Local name of the fifth day of the week.
CAL_SDAYNAME6	Local name of the sixth day of the week.
CAL_SDAYNAME7	Local name of the seventh day of the week.
CAL_SERASTRING	One or more null-terminated strings that specify each of the Unicode codepoints specifying the era associated with the specified CAL_IYEAROFFSETRANGE. The last string has an extra terminating null character.
CAL_SLONGDATE	Long date formats for this calendar type.
CAL_SMONTHNAME1	Local name of the first month of the year.
CAL_SMONTHNAME2	Local name of the second month of the year.
CAL_SMONTHNAME3	Local name of the third month of the year.
CAL_SMONTHNAME4	Local name of the fourth month of the year.
CAL_SMONTHNAME5	Local name of the fifth month of the year.
CAL_SMONTHNAME6	Local name of the sixth month of the year.
CAL_SMONTHNAME7	Local name of the seventh month of the year.
CAL_SMONTHNAME8	Local name of the eighth month of the year.
CAL_SMONTHNAME9	Local name of the ninth month of the year.
CAL_SMONTHNAME10	Local name of the tenth month of the year.
CAL_SMONTHNAME11	Local name of the eleventh month of the year.
CAL_SMONTHNAME12	Local name of the twelfth month of the year.
CAL_SMONTHNAME13	Local name of the thirteenth month of the year, if it exists.
CAL_SSHORTDATE	Short date formats for this calendar type.

If the local name for the day of the week or for a month is an empty string, that name is identical to the name given in the corresponding locale information and therefore is not duplicated here.

The CAL_IYEAROFFSETRANGE and CAL_SERASTRING values vary in format, depending on the type of optional calendar. The following example shows the values for these types—for each supported alternate calendar type—along with the formula for how to use the CAL_IYEAROFFSETRANGE value to compute the correct year given the Gregorian current year value Y.

```
CAL_ICALINTVALUE = "1"
CAL_IYEAROFFSETRANGE = ""
CAL_SERASTRING = ""

CAL_ICALINTVALUE = "2"
CAL_IYEAROFFSETRANGE = ""
CAL_SERASTRING = ""

CAL_ICALINTVALUE = "3"
CAL_IYEAROFFSETRANGE = "1989\01926\01912\01868\0"
CAL_SERASTRING = "Ux337B\0Ux337C\0Ux337D\0Ux337E\0"
if (Y>=1989) { Y = (Y-1989)+1; }
if (Y>=1926 && Y<1989) { Y = (Y-1926)+1; }
if (Y>=1912 && Y<1926) { Y = (Y-1912)+1; }
if (Y>=1868 && Y<1912) { Y = (Y-1868)+1; }
if (Y<1868) { Y = Y; }

CAL_ICALINTVALUE = "4"
CAL_IYEAROFFSETRANGE = "1912\0"
CAL_SERASTRING = "Ux4E2D\0Ux83EF\0Ux6C11\0Ux570B\0"
if (Y>=1912) { Y = (Y-1912)+1; }
if (Y<1912) { Y = Y; }

CAL_ICALINTVALUE = "5"
CAL_IYEAROFFSETRANGE = "2333\0"
CAL_SERASTRING = ""
Y = Y+2333;
```

CHAPTER 13

Working with the Input Method Editor

The *Input Method Editor (IME)* in Windows CE simplifies the process of providing input for users. In particular, IMEs are required for many Asian languages in order to input characters from the keyboard. These languages are often made up of thousands of distinct characters, which makes it impossible to show all of the characters on a single keyboard. To facilitate composition, the IME converts the keystrokes into the characters of the target language as a user types. Depending on the IME, these characters may be further converted. The IME in Windows CE that is localized for Japanese, for example, converts Roman keystrokes entered by a user to Kana or Hiragana. Then, an additional conversion changes characters to Kanji.

The IME can also present a list of alternatives—called the candidate list—in situations in which the composition is ambiguous. A Windows CE application uses the Input Method Manager (IMM) to communicate with the IME.

Overview of the Input Method System

The main parts of the Input Method system are the following:

- The IME kernel contains the knowledge of the specific language that a user is inputting.
- The IME user interface (UI) consists of a Status window, a default Composition window, a Candidate window, and a Guideline window.
- The *Input Method Manager (IMM)* coordinates the interaction among the window system, the application, and the IME.
- An *input context* maintains the current state of user interaction with the IME. In particular, it maintains the composition string, which is the characters that a user is in the process of inputting.
- An IME Control window routes unhandled IME messages to the IME.

The basic operation is as follows.

A user presses keys on a keyboard. These keystrokes are routed to the IME by the IMM. The IME uses the keys either as commands or to generate characters in the composition string. As the IME carries out its operations, it sends notification messages to the window that currently has the focus. If this window does not process the messages, the messages are sent to the IME Control window, which then routes them back to the IME for default processing. Windows that do not process IME messages are called IME-unaware windows. The IME provides the entire user interface, and the window is completely unaware that an IME is operational. Windows that intercept and process IME messages are called IME-aware windows. By intercepting IME messages, these windows may provide their own UI. By using IMM functions, these windows communicate with the IME.

Overview of the IME User Interface

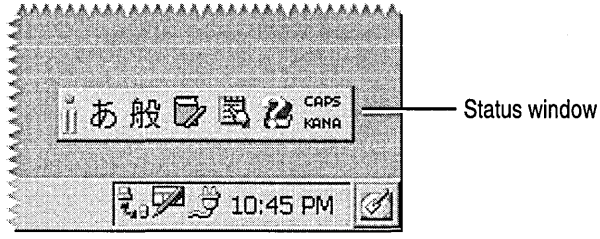
The IME-provided UI consists of the Status, Composition, Candidate, and Guideline windows. By default, the IME creates and manages these windows for all windows that require text input. For most applications, this default processing is sufficient. An application that relies entirely on the IME for its UI is considered IME-unaware because it is unaware that an IME is functioning in the system.

In contrast, an IME-aware application participates in the operation of the IME. Such an application can control the operation, position, and appearance of the IME, or can even provide its own view of the composition string and candidate list.

Working with the IME Status Window

The Status window provides information on the status of the IME and allows a user to set the IME conversion mode.

The following screen shot shows an active Status window.



IME hot keys turn the IME on or off or, in some situations, switch IMEs. An application can call the **ImmGetHotKey** function to retrieve, or the **ImmSetHotKey** function to set, the value of an IME hot key. This is not typically done by applications, however, because most users become accustomed to the particular hot key for the IME that they are using.

Applications cannot add hot keys to the system. Applications can initiate the same action as a hot key by using the **ImmSimulateHotKey** function.

Using the IME Composition Window

As a user types, characters are built into a composition string. The string is displayed in the default Composition window if the window that has the focus is IME-unaware. IME-aware windows display the composition string in their own window. The remainder of this section assumes that the string is displayed in the default Composition window. Depending on the conversion mode, the Composition window displays either converted or unconverted text.

An application can call the **ImmGetCompositionWindow** function to retrieve information about the Composition window contained in the **COMPOSITIONFORM** structure, or it can call the **ImmGetCompositionFont** function to get the logical font currently used to display characters in the Composition window.

Working with IME Composition Strings

The IME composition string is the current text in the Composition window. Each composition string consists of one or more clauses, where a clause is the smallest combination of characters that have a meaning in the language.

As a user enters text in the Composition window, the IME tracks the composition status, which includes attribute information, clause information, typing information, and cursor position.

Use the **ImmGetCompositionString** and **ImmSetCompositionString** functions to retrieve and set the composition status, or to retrieve and set the characters, attributes, and clauses of the composition string. **ImmSetCompositionString** also allows notification messages to be sent to the Composition window to ensure that this window updates its appearance based on the changes specified in this call. Applications that set a combination of composition status elements typically set the *fNotify* parameter to FALSE for all but the last call to this function so that only one notification message is generated for the Composition window.

The attribute information is an array of 8-bit values that specifies the status of characters in the composition string. In this array, all characters of one clause must have the same attribute. For each value in the array, bits 0 through 3 can be a combination of the values in the following table.

Value	Description
ATTR_INPUT	The character being entered by a user, not yet converted by the IME.
ATTR_INPUT_ERROR	The character is the error character and cannot be converted by the IME.
ATTR_TARGET_CONVERTED	The character converted by the IME. A user has selected this character, and the IME has converted it.
ATTR_CONVERTED	A converted character. The IME has already converted this character.
ATTR_TARGET_NOTCONVERTED	The character being converted. A user has selected this character, but the IME has not yet converted it.
ATTR_FIXEDCONVERTED	Characters that are not converted. The IME no longer converts these characters.

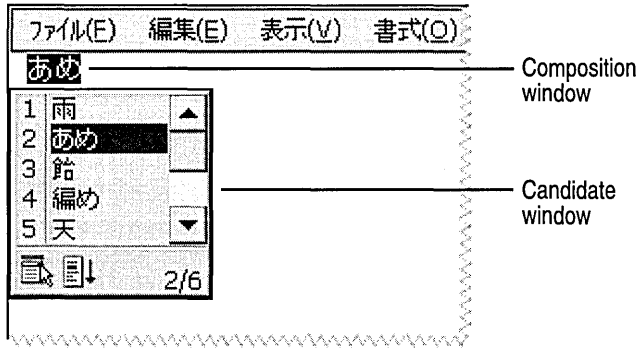
The clause information is an array of 32-bit values that specifies the positions of the clauses in the composition string. Each clause has one value in the array that specifies the beginning of the clause. These clause positions are followed by a final value that specifies the length of the full string. Each value in the array specifies the offset, in bytes, from the beginning of the composition string to the clause. The first value is always 0 because the first clause always starts at the beginning of the string. For example, if a string has two clauses, the clause information has three values: the first value is 0, the second value is the offset of the second clause, and the third value is the length of the string.

The typing information is a null-terminated character string that represents the characters entered at the keyboard.

The cursor position is a value indicating the position of the cursor relative to the characters in the composition string. This value is the offset, in bytes, from the beginning of the string. If this value is 0, the cursor is immediately before the first character in the string. If the value is equal to the length of the string, the cursor is immediately after the last character. If the value is –1, the cursor is not present.

Working with the IME Candidate Window

The Candidate window contains a list of candidate characters for the selected character in the Composition window. Users can scroll through this list and select the character that they want before composition of the character is completed in the Composition window. A user can compose the text that they want in this manner until the composition string is finalized and the Candidate window is closed. The following screen shot shows the Composition window with Hiragana text and the Candidate window with Kanji conversion options.



Once the character that a user wants has been determined, the IME sends this character to the application in the form of WM_IME_CHAR or WM_IME_COMPOSITION/GCS_RESULT messages. If the application does not process these messages, the **DefWindowProc** function translates them into one or more WM_CHAR messages.

The characters in the candidate list are arranged in an array of strings in the **CANDIDATELIST** structure. Use the **ImmGetCandidateList** function to retrieve a candidate list and copy the list to a buffer. Use the **ImmGetCandidateListCount** function to retrieve the total size, in bytes, of all the candidate lists. To get or set information about the Candidate window in the **CANDIDATEFORM** structure, use the **ImmGetCandidateWindow** and **ImmSetCandidateWindow** functions.

Handling IME Window Messages

The system sends IME window messages to the window that has focus when certain events occur. For example, the system sends the WM_IME_SETCONTEXT message when a window is activated. IME-unaware windows pass these messages to the **DefWindowProc** function, which sends them to the corresponding default IME window. IME-aware windows either process these messages or forward them to their own IME windows.

To have an IME window carry out an action, use the WM_IME_CONTROL message. To have the IME notify an application about changes to the composition string, use the WM_IME_COMPOSITION message. Use the WM_IME_NOTIFY message for general changes to the status of IME windows.

To process the WM_IME_COMPOSITION message, applications test the bits in the *lParam* parameter and call the **ImmGetCompositionString** function to retrieve the indicated string or data. The following code example checks for the result string, allocates sufficient memory for the string, and retrieves the result string from the IME.

```
HIMC hIMC;
HWND hWnd;
DWORD dwSize;
HGLOBAL hstr;
LPSTR lpstr;

case WM_IME_COMPOSITION:
    if (lParam & GCS_RESULTSTR)
    {
        hIMC = ImmGetContext(hWnd);

        If (!hIMC)
            MyError(ERROR_NULLCONTEXT);

        // Get the size of the result string.
        dwSize = ImmGetCompositionString(hIMC, GCS_RESULTSTR, NULL, 0);

        // Increase buffer size for NULL terminator;
        // maybe it is in unicode.
        dwSize += sizeof(WCHAR);

        lpstr = (LPSTR) LocalAlloc(LPTR, dwSize);
        if (lpstr == NULL)
             MyError(ERROR_GLOBALLOCK);

        // Get the result string that is generated by IME into lpstr.
        ImmGetCompositionString(hIMC, GCS_RESULTSTR, lpstr, dwSize);
        ImmReleaseContext(hWnd, hIMC);

        // Add this string into the text buffer of the application.

        LocalFree(lpstr);
    }
```

Working with Input Contexts

An *input context* stores information about the status of the IME. An input context is an internal structure maintained by the IME. The system creates and assigns a default input context to each thread. Within the thread, this default input context is a shared resource and is associated with each newly created window. Input contexts require considerable system resources, so use them conservatively.

To retrieve or set information in the IME, an application must first obtain a handle to the input context associated with a specified window. To obtain this handle, use the **ImmGetContext** function. You can use the returned handle in subsequent calls to the IMM functions to retrieve and set IME values, such as the Composition window style, the composition style, and the Status window position. The **ImmNotifyIME** function notifies the IME about changes to the input context. Once you are finished with the context, you must release it by using the **ImmReleaseContext** function.

Because the default input context is a shared resource, any changes that you make to it apply to all windows in the thread. However, you can override this default behavior by creating and associating your own input context for one or more windows in the thread. The changes that you make to your own input context apply only to the windows with which it is associated.

To create an input context, use the **ImmCreateContext** function. To assign an input context to a window, use the **ImmAssociateContext** function. **ImmAssociateContext** returns the handle of the previously associated input context. If you have not previously associated an input context with a window, the returned handle is for the default input context. When an application has finished with an input context that it has created, it calls **ImmDestroyContext** to free the memory that was allocated. It is the responsibility of the application to ensure that the input context being destroyed is not associated with a window. **ImmAssociateContext** is used to restore the original context of a window.

To create a new component that is a member of an input context structure, use the **ImmCreateIMCC** function. This function returns an input context component handle and initializes the component with 0s. To retrieve or change the size of the input context component, use the **ImmGetIMCCSize** and **ImmReSizeIMCC** functions. Use **ImmDestroyIMCC** to free the memory of an input context component.

To obtain a handle to an input context, an IME calls the **ImmLockIMC** function. To obtain a handle to an input context component that is a member of an input context, an IME calls the **ImmLockIMCC** function. With each call to these functions, the total number of handles, or the lock count, is incremented for the corresponding input context or input context component. To get the lock count of an input context or input context component, use the **ImmGetIMCLockCount** and **ImmGetIMCCLockCount** functions. To release a handle and decrement the lock count, use the **ImmUnlockIMC** or **ImmUnlockIMCC** function.

Once an input context is associated with a window, the system automatically selects that context when the window receives the input focus. The style and other information in the input context affects subsequent keyboard input for that window, and determines the appearance and operation of the IME.

Windows CE Glossary

A

ACCEL data structure A data structure that defines an accelerator key used in an accelerator table.

accelerator table An array of ACCEL data structures, each of which defines an accelerator.

Action button A hardware navigation control on a Palm-size PC that functions like the ENTER key on a keyboard.

Active Channel A Web site that has been enabled for Webcasting to information-receiving applications.

Active Desktop A technology delivered in Microsoft Internet Explorer and Microsoft Pocket Internet Explorer that allows you to include HTML documents, ActiveX controls, and Java applets on your desktop.

active notification The state of a user notification from the time the user is notified until the user handles the event. *See also* **user notification.**

Active Server Pages (ASP) An open application environment in which HTML pages, scripts, and ActiveX components are combined to create Web-based applications.

Active Template Library (ATL) A C++ template library used to create ActiveX servers and other Component Object Model (COM) objects. ActiveX controls created with ATL are generally smaller and faster than those created with the Microsoft Foundation Classes.

active window In an environment capable of displaying multiple on-screen windows, the window containing the display or document that will be affected by current cursor movements, commands, and text entry. *See also* **graphical user interface.**

ActiveX A set of technologies that enable software components to interact with one another in a networked environment, regardless of the language in which the components were created. ActiveX, which was developed as a proposed standard by Microsoft in the mid 1990s and is currently administered by the Open Group, is built on Microsoft's Component Object Model (COM). Currently, ActiveX is used primarily to develop interactive content for the World Wide Web, although it can be used in desktop applications and other applications. ActiveX controls can be embedded in Web pages to produce animation and other multimedia effects, interactive objects, and sophisticated applications. *See also* **COM.**

ActiveX client An application or tool that calls an ActiveX object.

ActiveX object An exposed object of the Component Object Model (COM).

ADC *See* **analog-to-digital converter.**

address card The fundamental unit of record in the Contacts database. Each address card contains information about an individual, such as name and address.

address mask A number that, when compared by the computer with a network address number, will block out, or mask, all but the necessary data. For example, bits in the address corresponding to one in the mask are used, but bits corresponding to zero are ignored.

Address Resolution Protocol (ARP) A TCP/IP protocol for determining the hardware address, or physical address, of a node on a local

area network connected to the Internet, when only the IP address, or logical address, is known. An ARP request is sent to the network, and the node that has the IP address responds with its hardware address. Although ARP technically refers only to finding the hardware address, and RARP, Reversed ARP, refers to the reverse procedure, ARP is commonly used for both senses. *See also* **IP address, TCP/IP.**

Advanced Technology Attachment (ATA) ANSI group X3T10's official name for the disk drive interface standard commonly known as Integrated Drive Electronics (IDE). *Also called* AT Attachment.

American National Standards Institute (ANSI) A voluntary, nonprofit organization of U.S. business and industry groups formed in 1918 for the development of trade and communication standards. ANSI is the American representative of the International Standards Organization and has developed recommendations for the use of programming languages including FORTRAN, C, and COBOL.

American Standard Code for Information Exchange (ASCII) A coding scheme using 7 or 8 bits that assigns numeric values to up to 256 characters, including letters, numerals, punctuation marks, control characters, and other symbols. ASCII was developed in 1968 to standardize data transmission among disparate hardware and software systems and is built into most minicomputers and all personal computers.

analog-to-digital converter (ADC) A device that converts an analog signal, such as sound or voltage, to binary code for use by a computer.

annunciator An icon placed onto the taskbar to indicate that a user notification is active. Although taskbars can contain multiple annunciator icons for different applications, only one instance of an icon for any given application is displayed at one time.

ANSI *See* **American National Standards Institute.**

apartment threading model A threading model that can be used only on the thread that created it. *See also* **free threading model** and **single threading model.**

API *See* **application programming interface.**

apogee The point within the orbit of a satellite where the satellite is farthest from the earth.

application-defined message A message created by an application to be used by its own windows or to communicate with windows in other processes. If an application creates its own message, the window procedure that receives the message must interpret it and provide the appropriate processing.

application notification An application notification starts an application at a specified time or when a system event occurs. When an application starts as the result of a notification, the system specifies a command-line parameter that identifies the event that has occurred.

application programming interface (API) A set of routines used by an application to direct the performance of procedures by a computer's operating system. For computers running a graphical user interface, an API manages an application's windows, icons, menus, and dialog boxes.

application-specific integrated circuit (ASIC) An integrated circuit designed to perform a particular function by defining the interconnection of a set of basic circuit-building blocks drawn from a library provided by the circuit manufacturer.

application switch A hardware navigation control intended to launch or reactivate software applications.

argument of the perigee The angle, as measured from the center of the earth, between the perigee and the ascending node of orbit of a satellite. The argument of the perigee must be a value between 0 and 2 pi radians. *See also* **ascending node** and **perigee.**

ARP *See* **Address Resolution Protocol.**

ascending node The point within the orbit of a satellite where the satellite crosses the earth's equatorial plane while travelling from south to north.

ASCII *See* **American Standard Code for Information Interchange.**

ASIC *See* **application-specific integrated circuit.**

ASP *See* **Active Server Pages.**

asynchronous operation **1.** A process in a multitasking system whose execution can proceed independently, or in the background. Other processes may be started before the asynchronous process has finished. **2.** A data transmission method that allows characters to be sent at irregular intervals over a line by preceding each character with a start bit and following it with a stop bit. *Compare* **synchronous operation.**

ATA *See* **Advanced Technology Attachment.**

ATAPI The interface used by the IBM PC AT system for accessing CD-ROM devices.

ATL *See* **Active Template Library.**

ATL for Windows CE The Active Template Library for Windows CE. *See* **Active Template Library.**

audio driver model The basic interface layer between the audio device drivers and the upper-layer application programming interfaces (APIs) and applications.

authentication **1.** The process of verifying that a message comes from its stated source. **2.** The process of verifying the identity or access level of a user, computer, or application.

Automation A technology based on the Component Object Model (COM), that enables interoperability among ActiveX components, including OLE components. Formerly referred to as OLE Automation. *See also* **OLE.**

B

background audio source An audio source that is not controlled by the operating system, such as a radio, a CD player, or an auxiliary input device. Background audio sources may continue playing in the background when foreground audio sources are active. *See also* **foreground audio source.**

background graphics mode Defines how background colors are mixed with window or screen colors for text and bitmap operations. *See also* **drawing mode.**

backlight A light source for a backlit display.

backup authority A trusted application running on a secure computer used as a storage medium.

bandwidth **1.** The difference between the highest and lowest frequencies that an analog communications system can pass. For example, a telephone accommodates a bandwidth of 3,000 Hz, which is the difference between the lowest (300 Hz) and highest (3,300 Hz) frequencies it can carry. **2.** The data transfer capacity of a digital communications system.

base font The font glyphs that you obtain from another font when performing font linking. *See also* **linked font.**

bidirectional parallel port An interface that supports two-way parallel communications between a device and a computer.

binary image builder file (.bib) A file used by the Windows CE ROM image builder tool to determine which modules and files to combine when forming the ROM image, and where to place the modules in memory.

binary large object (BLOB) **1.** A large piece of data, such as a bitmap, characterized by large field values, an unpredictable table size, and data that is formless from the perspective of an application. **2.** A keyword that designates the BLOB structure that contains information about a block of data.

bit block transfer (blit) The process of copying the bits that constitute a bitmap from one device context to another. For example, a bit block transfer can be used to move a bitmap stored in memory to the screen for display. The bits can also be altered during a bit block transfer. As a result, light and dark portions of an image can be reversed. Successive displays can thus be used to change the appearance of an image or to move it around on the screen.

bitmap A data structure in memory that represents information as a collection of individual bits. A bitmap represents a bit image. A bit map is also used in some systems to represent the blocks of storage on a disk, indicating whether each block is free (0) or in use (1).

blink time The elapsed time, in milliseconds, required to invert the caret display. This value is half of the flash time.

blit *See* **bit block transfer.**

BLOB *See* **binary large object.**

block cipher mode An encryption scheme in which data is encrypted one block at a time. Compare **stream cipher mode.**

block mode A synchronous method of calling the **CeRapiInvoke** function by storing input parameters and output data in a single buffer.

boot loader An application that is automatically run when a computer is switched on (booted). After first performing a few basic hardware tests, the boot loader loads and passes control to a larger loader application, which then typically loads the operating system. The boot loader normally resides in the computer's read-only memory (ROM).

bootstrap loader *See* **boot loader.**

bounding rectangle The smallest rectangle that completely surrounds an ellipse.

brush A tool used in painting applications to sketch or fill in areas of a drawing with the color and pattern currently in use. Painting applications that offer a variety of brush shapes can produce brushstrokes of varying width and, in some cases, shadowing or calligraphic effects.

brush origin The coordinates of a mapped pixel.

build environment The state of the development workstation and the directory structure when an application build begins.

build window *See* **command prompt build window.**

built-in device driver *See* **native device driver.**

C

CA *See* **Certification Authority.**

cabinet file A self-contained file with a .cab extension used for application installation and setup. In a cabinet file, multiple files are compressed into one file. They are commonly found on Microsoft software distribution disks.

cache A special memory subsystem in which frequently used data values are duplicated for quick access. A memory cache stores the contents

of frequently accessed RAM locations and the addresses where this data is stored. When the processor references an address in memory, the cache checks to see whether it holds that address. If it does hold the address, the data is returned to the processor; if it does not hold the address, a regular memory access occurs. A cache is useful when RAM accesses are slow compared with the microprocessor speed, because cache memory is always faster than main RAM memory.

callback function A function that receives messages from the operating system. Callback functions are application-defined.

caret A flashing line, block, or bitmap that marks the location of the insertion point in a window's client area.

cascading menu A hierarchical graphical menu system in which a side menu of subcategories is displayed when the pointer is placed on the main category.

CDF *See* **Channel Definition Format.**

central processing unit (CPU) The computational and control unit of a computer. The CPU is the device that interprets and executes instructions. It has the ability to fetch, decode, and execute instructions and to transfer information to and from other resources over the computer's main data-transfer path, the bus. By definition, the CPU is the chip that functions as the "brain" of a computer. In some instances, however, the term encompasses both the processor and the computer's memory or, even more broadly, the main computer console, as opposed to peripheral equipment.

certificate A packet of data containing a public key and identification information. Every certificate is created and signed by Certification Authorities.

Certification Authority (CA) An entity that attests to the identity of a person or an organization. The Certification Authority's primary function is to verify the identity of entities and issue digital certificates attesting to that identity.

Cesh The Windows CE Debug Shell Tool (Cesh.exe). This shell enables you to transfer an operating system image from the development workstation to the target platform and provides you with a set of commands to assist in debugging processes running on the target platform.

channel A subscription to a Web site that conforms to the Channel Definition Format.

Channel Definition Format (CDF) A specification developed by Microsoft and presented to the World Wide Web Consortium (W3C) that allows applications to send Web pages to users. Once a user subscribes to a CDF channel, any software that supports the CDF format automatically receives any new content posted on the channel's Web server. The default client subscription application for Internet channel broadcasting in Broadcast Architecture stores subscription information as .cdf files.

channel script An application written in HTML that uses Visual Basic Script, JScript, Java Script, and other scripting languages to specify the layout and behavior of a channel.

channel synchronization The process of first downloading Mobile Channels content into a cache using the standard Internet Explorer 4.0 channel retrieval mechanism and then transferring it onto a Windows CE–based device. Channel synchronization makes it possible for users to access Mobile Channels using either a Windows CE–based device without a radio module or a Windows-based desktop computer when the device is not readily available. *See also* **Mobile Channels.**

check box An interactive control found in graphical user interfaces. Check boxes are used to enable or disable one or more features or options from a set. When an option is selected, an x or a check mark appears in the box.

child window A window that has the WS_CHILD style. A child window always appears within the client area of its parent window.

chord A combination of navigation controls used to perform a defined function. A chord is functionally similar to a keyboard accelerator. For example, on some Palm-size PCs, pressing and holding an Action button and then pressing an Exit button toggles the backlight.

CIFS *See* **Common Internet File System.**

CIFS redirector A module through which one computer gains access to another. Its function is to reestablish disrupted connections and to package and send remote file-system requests to host targets.

cipher mode A method used to encrypt data.

ciphertext Data that has been encrypted.

class identifier (CLSID) A universally unique identifier (UUID) that identifies a type of Component Object Model (COM) object. Each type of COM object item has its CLSID in the registry so that it can be loaded and used by other applications. For example, a spreadsheet may create worksheet items, chart items, and macrosheet items. Each of these item types has its own CLSID that uniquely identifies it to the system.

client **1.** In object-oriented programming, a member of a class (group) that uses the services of another class to which it is not related. **2.** A process, such as an application or task, that requests a service provided by another application. For example, a word processor that calls on a sort routine built into another application. The client process uses the requested service without having to know any working details about the other application or the service itself. **3.** On a local area network or the Internet, a computer that accesses shared network resources provided by another computer, called a *server.*

client area The client area is the portion of a window where the application displays output, such as text or graphics. *Also called* a client rectangle.

client coordinate A coordinate that is relative to the upper-left corner of a window's client area.

clipping region A subregion of the client area to which output is restricted. Clipping is used in Windows CE in a variety of ways. For example, word processing and spreadsheet applications clip keyboard input to keep it from appearing in the margins of a page or spreadsheet.

CLSID *See* **class identifier.**

code element The smallest component of a written language that has semantic value. *See also* **codespace** and **code value.**

codespace The logical grouping of code elements throughout the range of supported Unicode code values. *See also* **code element** and **code value.**

code value A single 16-bit number assigned to each code element that is defined by the Unicode standard. When referred to in text, each code value is listed in hexadecimal form following the prefix *U. See also* **code element** and **codespace.**

cold boot A startup process that begins with turning on the computer's power. Typically, a cold boot involves some basic hardware checking by the system, after which the operating system is loaded from disk into memory. *Compare* **warm boot.**

COM *See* **Component Object Model.**

COM class The definition of an object in code. In COM, class refers to the general object definition, whereas in C++, the class of an object is a data type.

COM object A programming structure that includes both data and functionality. A COM object is defined and allocated as a single unit. The only public access to a COM object is through the programming structure's interfaces. At a minimum, a COM object must support the **IUnknown** interface, which maintains the object's existence while it is being used and provides access to the object's other interfaces.

COM port Short for communications port, the logical address assigned by MS-DOS versions 3.3 and later, and Microsoft Windows to each of the four serial ports on an IBM personal computer or an IBM PC–compatible computer. COM ports also have come to be known as the actual serial ports on a computer's CPU where peripherals, such as printers, scanners, and external modems, are plugged in.

combo box A control that combines an edit control with a list box. This allows the user to type in an entry or choose one from the list.

command band A rebar control with a fixed band at the top that contains a toolbar with a **Close** (**X**) button, an **OK** button, and optionally, a **Help** (**?**) button in the upper-right corner.

command bar A control window that can contain buttons, combo boxes, and menu bars. Windows CE–based applications can use a command bar rather than a separate menu and toolbar to efficiently utilize available screen space.

command prompt build window A development workstation command prompt window from which the Platform Builder user has run the Wince.bat tool. *Also called* build window.

common control A standardized child window that an application uses in conjunction with another window to perform input/output tasks. A common control enables users to view and organize information and to set or change attributes and properties. Most common controls send the WM_NOTIFY message.

Common Internet File System (CIFS) A standard proposed by Microsoft that would compete directly with Sun Microsystems' Web Network File System. A system of file sharing of Internet or intranet files.

Compact Flash A group of related technologies for providing long-term storage through various types of nonvolatile memory.

component A subset of the Windows CE operating system. Windows CE is structured as a collection of modules that are subdivided into smaller components. Each module and component is a self-contained subset of the Windows CE operating system that can be used to construct a customized operating system for a particular device.

Component Object Model (COM) An open architecture for cross-platform development of client/server applications. It is based on object-oriented technology as agreed upon by Digital Equipment Corporation and Microsoft Corporation. COM defines the interface, similar to an abstract base class, **IUnknown**, from which all COM-compatible classes are derived.

compound file A number of individual files bound together in one physical file where each individual file can be accessed as if it were a single physical file.

connection-based session A communications session that requires a connection to be established between hosts prior to an exchange of data.

connectionless session A communications session that does not require a connection to be established between hosts prior to an exchange of data.

Contacts database A collection of names, addresses, telephone numbers, and other information stored on a Windows CE device by the Contacts application. The database is divided into a set of records called address cards. The database contains any number of address cards, limited only by the amount of memory available on the device.

container A network resource that contains other resources.

continuous resistive touch panel
See **touch screen.**

control A standardized child window on the screen that can be manipulated by the user to perform an action or display information. The most common controls are buttons, which allow the user to select options, and scroll bars, which allow the user to move through a document or position text in a window.

control identifier A value that uniquely identifies a control.

control style A value, similar to a window style, that specifies the appearance and behavior of a control. The window procedure for the control uses the style to determine how to draw the control and process input.

cookie A block of data a Web server stores on a client system. When a Web client user returns to the Web server site, the browser sends a copy of the cookie to the server. Cookies identify users, instruct the server to send a customized version of the requested Web page, submit account data for the user, and fulfill other administrative purposes.

CPU *See* **central processing unit.**

credential Data used by a principal to establish the identity of the principal, such as a password or user name.

critical section An object that protects a section of code from being accessed by more than one thread. A critical section is limited to only one process or dynamic-link library (DLL) and cannot be shared with other processes.

cryptographic service provider (CSP)
An independent module that performs cryptographic operations, such as creating and destroying keys. A cryptographic service provider consists of, at a minimum, a dynamic-link library (DLL) and a signature file.

CSP *See* **cryptographic service provider.**

cursor A small bitmap whose location on the screen is controlled by a pointing device, such as a mouse, pen, or trackball. Some Windows CE–based platforms support only the wait cursor—the spinning hourglass.

custom command A speech command that is trained by the user. A custom command has a speaker-dependent template.

D

database synchronization The process of bringing two separate copies of a database into agreement.

database system application programming interface
A set of functions that enable you to create and manipulate Windows CE databases. Each database consists of an arbitrary number of records, and each record consists of at least one *property*.

database type identifier A user-specified token, or number, that is attached to a database. The token can be used to identify related databases by associating the same value, or related values, with each database.

datagram A data packet, containing sufficient delivery information, that can be routed through a packet-switching network without reliance on exchanges between the source and destination computer.

data link A connection between any two devices capable of sending and receiving information, such as a computer and a printer or a main computer and a terminal. Sometimes the term is extended to include equipment, such as a modem, that enables transmission and receiving. Such devices follow protocols that govern data transmission.

date and time picker control (DTP) A control that displays information about dates and times, and provides users with an easy way to modify this information.

datum A frame of reference for coordinates used to describe horizontal or vertical position on or near the surface of the earth.

DCC *See* **direct cable connection.**

DDB *See* **device-dependent bitmap.**

DDE *See* **dynamic data exchange.**

DDI *See* **device driver interface.**

DDK *See* **Device Driver Kit.**

DDTK *See* **Device Driver Test Kit.**

dead key A key used with another key to create an accented character. A dead key, when pressed, produces no visible character, but indicates that the accent mark it represents is to be combined with the character produced by the next letter key pressed.

decryption The process of returning encrypted data to its original form.

de-emphasis Resets the signal magnitude on an audio channel. *Compare* **pre-emphasis.**

derived session key A session key created by an application as needed. Before creating a derived session key, an application prompts the user for a password.

deserialize The process of converting a series of bytes back into an object. *Compare* **serialize.**

desktop An on-screen work area that uses icons and menus to simulate the top of a desk. Its intent is to make a computer easier to use by enabling users to move pictures of objects and to start and stop tasks in much the same way as they would if they were working on a physical desktop.

desktop connectivity The services required to connect a Windows CE–based device to a desktop computer.

desktop provider module One of two DLLs that comprise a service provider. The desktop provider module handles the bulk of communication with the service manager and implements two COM interfaces. *See also* **service provider** and **service manager.**

development workstation The PC-based computer on which you install Windows CE development toolkits and develop software for your Windows CE–based platform.

device **1.** A generic term for a computer subsystem. Printers, serial ports, and disk drives are often referred to as devices; such subsystems frequently require their own controlling software, called device drivers. **2.** A hardware feature that can—or must—be part of the target platform. For example, a built-in device could be a low-battery notification LED, while a PC Card modem is an installable device. *See also* **device driver.**

device context A GDI structure containing information that governs the display of text and graphics on a particular output device. A device context stores, retrieves, and modifies the attributes of graphic objects and specifies graphic modes. The graphic objects stored in a device

context include a pen for line drawing, a brush for painting and filling, a font for text output, a bitmap for copying or scrolling, a palette for defining the available colors, and a region for clipping.

device-dependent bitmap (DDB) An array of bits that can only be used with a particular display or printer.

device driver A software component that permits a computer system to communicate with a device. In most cases, the driver also manipulates the hardware in order to transmit the data to the device. However, device drivers associated with application packages typically perform only the data translation; these higher-level drivers then rely on lower-level drivers to actually send the data to the device. Many devices will not work properly—if at all—without the correct device drivers installed in the system.

device driver interface (DDI) **1.** The interface between applications and the device drivers. **2.** A set of functions implemented in the model device driver and called by the Graphics, Windowing, and Events Subsystem (GWES).

Device Driver Kit (DDK) A set of tools and libraries that enable programmers to write Windows-based software used to run hardware devices such as printers.

Device Driver Test Kit (DDTK) A set of tools and libraries that enable you to test the porting of your device drivers to the Windows CE operating system.

device-independent bitmap (DIB) An array of bits combined with several structures that specify the width and height of the bitmap image (in pixels), the color format of the device with which the image was created, and the resolution of the device used to create that image. A DIB generally has its own color table, and can therefore be displayed on a variety of devices.

device manager An application, included on all Windows CE–based platforms, that manages stream interface device drivers. The device manager handles loading and unloading stream interface device drivers, identifying the correct driver for plug-and-play devices, managing running device drivers, and notifying stream interface device drivers of power-up and power-down events.

device partnership A device partnership is a registry key on a Windows CE device that a desktop computer uses to identify a Windows CE device to which a desktop computer is connected. The key defines values for synchronization, file conversions, and backup and restore information, which enable multiple Windows CE devices to connect to the same desktop computer. A device partnership is created the first time you connect a Windows CE device to a desktop computer.

device provider module One of two DLLs that comprise a service provider. The device provider module handles communication between the service manager and the device. *See also* **service provider** and **service manager**.

DHCP *See* **Dynamic Host Configuration Protocol.**

dialog box A temporary window that contains controls. You can use it to display status information and to get user input.

dialog box procedure An application-defined callback function that the system calls when it has input for a dialog box or has tasks for a dialog box to carry out.

dialog box template A binary description of a dialog box and the controls it contains. You can create this template as a resource to be loaded from the application's executable file, or created in memory while the application runs.

DIB *See* **device-independent bitmap.**

digital signature Binary data attached to a message that uniquely identifies a sender. A digital signature can be used with hash values to ensure that a transmitted message has not been tampered with.

direct cable connection (DCC) A RAS networking connection between two computers, or between a computer and a Windows CE–based device, which uses a serial or parallel cable directly connected between the systems instead of a modem and a phone line.

direct memory access (DMA) Memory access that does not involve the microprocessor and is frequently used for data transfer directly between memory and an "intelligent" peripheral device, such as a disk drive.

discrete speech recognition Speech recognition that recognizes words that are delineated by pauses.

DLL *See* **dynamic-link library.**

DMA *See* **direct memory access.**

DNS *See* **Domain Name System.**

Domain Name System (DNS) A name service that resolves system names to current IP addresses and uses a tiered or hierarchical model to pass name resolutions between domains.

dotted decimal notation The process of formatting an Internet Protocol (IP) address as a 32-bit identifier made up of four groups of numbers, with each group separated by a period. For example, 123.432.154.12.

drag-and-drop A technique for moving or copying data between applications, between windows within an application, or within a single window in an application. The user selects the data to be transferred and drags the data to the desired destination. Windows CE supports drag-and-drop operations. However, nondefault drag-and-drop operations, equivalent to right mouse button drag-and-drop operations, are not supported.

drawing mode Defines how foreground colors are mixed with window or screen colors for pen, brush, bitmap, and text operations. *See also* **background graphics mode.**

drop-down menu A menu that drops from the menu bar when requested and remains open without further action until the user closes it or chooses a menu item.

DTP control *See* **date and time picker control.**

dummy file filter A means for transferring files of nonstandard or possibly unknown extensions for which no translation is necessary. Passing the file through the dummy filter keeps the **No Convertor Selected** dialog box from appearing.

dynamic data exchange (DDE) An interprocess communication method that allows two or more applications running simultaneously to exchange data and commands.

Dynamic Host Configuration Protocol (DHCP) A TCP/IP protocol that enables a network connected to the Internet to automatically assign a temporary Internet protocol (IP) address to a host when the host connects to the network.

dynamic-link library (DLL) A set of autonomous functions that any application can use. DLLs are a set of source code modules with each module containing a set of functions.

E

eccentricity A value that specifies how much the shape of the elliptical orbit of a satellite deviates from a circular path. Eccentricity equals the distance between the foci of the orbital ellipse divided by one-half the length of the major axis of

the orbital ellipse. The eccentricity must be between 0 and 1. Values closer to 0 indicate that the orbit is closer to a circular path. *See also* **focus** and **major axis.**

edit control A rectangular window in which a user can enter and edit text from the keyboard. An edit control is also referred to as a text box.

embedded Software code or commands built into their carriers. For example, applications insert embedded printing commands into a document to control printing and formatting. Low-level assembly is embedded in higher-level languages, such as C, to provide more capabilities or better efficiency.

encryption The process of transforming data into a form unreadable by anyone without a secret key.

engine **1.** A section of an application that determines how that application manages and manipulates a type of data. **2.** An application or module with an open API to which an application passes data in order to access the engine's processing capabilities.

environment variable An element of the operating system environment, such as a path, a directory name, or a configuration string. Environment variables are typically set within batch files.

Ethernet A widely used LAN developed by Xerox, Digital, and Intel. Ethernet networks connect up to 1,024 nodes at 10 megabits per second over twisted pair, coaxial cable, and optical fiber.

event An event is an occurrence that triggers a notification. Windows CE supports timer and system events.

event-driven operating system An operating system that constantly evaluates and responds to sets of events, such as key presses or mouse movements.

event object A synchronization object that enables one thread to notify another that an event has occurred. Event objects are useful when a thread needs to know when to perform its task. For example, a thread that copies data to an archive needs to be notified when new data is available. By using an event object to notify the copying thread of the availability of new data, the thread can perform its task as soon as possible.

exception handling The process of dealing with exceptions, or errors, as they arise during application execution. Exceptions occur when an application executes abnormally due to conditions outside the application's control. Windows CE does not support C++ exception handling.

execute in place (XIP) The process of executing code directly from read-only memory (ROM), rather than loading it from random access memory (RAM) first. Executing the code in place, instead of copying the code into RAM for execution, saves system resources. Applications in other file systems, such as on a PC Card storage device, cannot be executed in this way.

Exit button A hardware navigation control that functions as the ESC key on a keyboard.

extension key An entry in the registry, corresponding to the extension of a given file, that specifies which file filter will handle conversions for that file type.

F

FAT *See* **file allocation table.**

file allocation table (FAT) A table or list maintained by some operating systems to manage disk space used for file storage. Files on a disk are stored, as space allows, in fixed-size groups of bytes (characters) rather than from beginning to end as contiguous strings of text or numbers. A single file can thus be scattered in pieces over many separate storage areas. A file allocation table maps available disk storage space so that it

can mark flawed segments that should not be used and can find and link the pieces of a file. In MS-DOS, the file allocation table is commonly known as the FAT.

file filter A Windows CE dynamic-link library (DLL) that controls the transfer of data between a desktop computer and a Windows CE–based device.

file handle A token, or number, that the operating system uses to identify or refer to an open file or, sometimes, to a device.

file pointer The offset into an open file that the operating system maintains internally. It points to the starting location where data is read from or written to when a read or write operation is performed on an open file. The file pointer can be moved to any location within the file by a seek operation.

file system In an operating system, the overall structure in which files are named, stored, and organized. A file system consists of files, directories, and the information needed to locate and access these items. The term can also refer to the portion of an operating system that translates requests for file operations from an application into low-level, sector-oriented tasks that can be understood by the drivers controlling the disk drives.

file system application interface A subset of the standard Win32 file system functions. These functions let you create directories and data files, read and write file data, and retrieve file and directory information.

file system driver (FSD) A user-written DLL that the operating system loads to interface to a user-created installable file system. The functions that access an installable file system are implemented in the DLL. See also **installable file system**.

File Transfer Protocol (FTP) The protocol used for copying files to and from remote computer systems on a network using a Transmission Control Protocol/Internet Protocol (TCP/IP), such as the Internet. This protocol also allows users to use FTP commands to work with files, such as listing files and directories on the remote system.

firmware Software routines stored in read-only memory (ROM). Unlike random access memory (RAM), read-only memory stays intact even in the absence of electrical power. Startup routines and low-level input/output instructions are stored in firmware. It falls between software and hardware in terms of ease of modification. See also **RAM** and **ROM**.

first order clock correction A measurement of how much the atomic clock on a Global Positioning System (GPS) satellite drifts over time.

flash card See **PC Card storage device**.

flash memory Semiconductor memory that can operate as ROM but, on an activating signal, can rewrite its contents as though it were RAM.

flash time The elapsed time, in milliseconds, required to display, invert, and restore the caret display. This value is twice as much as the blink time.

focus One of the two points that determine the shape of an ellipse. The sum of the distances between any point on the ellipse and each of the foci is constant.

focus window The window that is currently receiving keyboard input. The focus window is always the active window, a descendant of the active window, or NULL.

font mapping The process of matching an application-defined description of a font with a font that is physically stored on a device or in an operating system. An application-defined font is called a logical font and a font on a device or in

an operating system is called a physical font. *See also* **logical font** and **physical font.**

foreground audio source An audio source that is controlled entirely by the operating system, such as speech recognition tones and text-to-speech (TTS). Depending on user settings, a foreground audio source can partially or fully mute background audio sources while they play. *See also* **background audio source.**

foreground thread The thread used to create the window with which the user is currently working.

foreground window The window with which the user is currently working. The system assigns a slightly higher priority to the thread used to create the foreground window than it does to other threads.

form An ActiveX control container that you customize to create a user interface for your application. A form contains a collection of controls, such as speech controls, power list boxes, audio controls, and tabber controls. A form displays information on a screen.

Forms Manager An ActiveX control container that manages forms, providing focus management, menus, help, and event sinks.

fragmentation The process of separating a datagram into smaller pieces for routing between networks.

free threading model A model in which an object can be used on any thread at any time. *See also* **apartment threading model** and **single threading model.**

FSD *See* **file system driver.**

FTP *See* **File Transfer Protocol.**

G

gateway A device that connects networks using different communications protocols.

GDI *See* **graphics device interface.**

GDOP *See* **geometric dilution of precision.**

geoid A model for the surface of the earth based on gravitational factors. A geoid approximates a sea-level surface and is the reference for most land elevations and ocean depths that appear on maps and charts.

geometric dilution of precision (GDOP) A numerical factor that specifies the portion of the error in a Global Positioning System (GPS) position, and time measurement that results from the geometry of the GPS satellites used to make the measurement.

Global Positioning System (GPS) A space-based radio navigation system that consists of 24 satellites and ground support. This system provides a user with accurate information about vehicle position and velocity.

globalization The process of developing an application core whose feature and code designs do not make assumptions based on a single language or locale, and whose source code simplifies the creation of different language editions of an application.

global variable A variable whose value can be accessed and modified by any statement in an application, and not merely within a single routine in which it is defined.

GPS *See* **Global Positioning System.**

graphics device interface (GDI) The Windows CE subsystem responsible for displaying text and images on display devices and printers. The GDI processes graphical function calls from a Windows-based application. It then passes those calls to the appropriate device driver, which generates the output on the display hardware. By acting as a buffer between applications and output

devices, the GDI presents a device-independent view of the world for the application while interacting in a device-dependent format with the device. Because of the smaller memory footprint of Windows CE–based devices, Windows CE supports only a subset of the standard Win32 GDI.

graphics object The pen, brush, bitmap, palette, region, font, and path associated with a device context. Windows CE does not support paths.

Graphics, Windowing, and Events Subsystem (GWES)
The Windows CE module that contains the graphics and windowing functionality needed to display text and images and to receive user input. It includes all the functionality needed to create and manage windows, controls, dialog boxes, and resources such as icons and menus. It also processes all user input. GWES includes the graphics device interface, which displays text and images on display devices and printers.

grayscale A sequence of shades ranging from black through white, used in computer graphics to add detail to images or to represent a color image on a monochrome output device. Like the number of colors in a color image, the number of shades of gray depends on the number of bits stored per pixel. Grays may be represented by actual gray shades, by halftone dots, or by dithering.

gripper bar A gripper bar is a tall, thin rectangle with a dark stripe running through it that appears on a rebar or a command band control. By touching and dragging a gripper bar with a stylus, a user can reposition a rebar or command bar. Gripper bars are especially useful for bringing off-screen rebar or command bar controls into view.

group box A rectangular area within a dialog box in which you can group together other controls that are semantically related. The controls are grouped by drawing a rectangular border around them. Any text associated with the group box is displayed in its upper-left corner.

GUID A globally unique identifier. *See* **universally unique identifier.**

GWES *See* **Graphics, Windowing, and Events Subsystem.**

H

HAL *See* **hardware abstraction layer.**

handle **1.** A pointer to a pointer; that is, a variable that contains the address of another variable, which in turn contains the address of the desired object. In certain operating systems, the handle points to a pointer stored in a fixed location in memory, whereas that pointer points to a movable block. If applications start from the handle whenever they access the block, the operating system can perform memory management tasks such as garbage collection without affecting the applications. **2.** Any token that an application can use to identify and access an object such as a device, a file, a window, or a dialog box. **3.** One of several small squares displayed around a graphical object in a drawing application. The user can move or reshape the object by clicking on a handle and dragging.

hardware abstraction layer (HAL)
Provides high-level access and control of various audio devices. HAL allows an application to communicate with various audio devices in a standard way.

hash value A value used in creating digital signatures. This value is generated by imposing a hashing algorithm onto a message. This value is then transformed, or signed, by a private key to produce a digital signature. *Also called* message digest.

hashing algorithm A formula used to generate hash values and digital signatures.

header control A horizontal window that is usually positioned above columns of data. It is divided into partitions that correspond to the columns, and each partition contains the title for the column below it.

heap A portion of memory reserved for an application to use for the temporary storage of data structures whose existence or size cannot be determined until the application is running. The application can request free memory from the heap to hold such elements, use it as necessary, and later free the memory.

hibernation The way in which a Windows CE–based device manages a memory shortage by requesting applications to free memory that is not currently needed.

hibernation threshold The point at which the system enters a limited-memory state.

high-resolution performance counter Hardware that provides high-resolution timing, which is useful in improving the performance of applications.

High Sierra specification An industrywide format specification for the logical structure, file structure, and record structure on a compact disc.

H/PC A Handheld PC.

hook A location in a routine or application in which the programmer can connect or insert other routines for the purpose of debugging or enhancing functionality.Windows CE does not support hooking.

host A device on a TCP/IP network that can be identified by a logical IP address.

host identifier An address that identifies a workstation, server, router, or other TCP/IP host within a network. Each host address must be unique to the network identifier.

hot key A keystroke or combination of keystrokes that switches the user to a different application, often a terminate-and-stay-resident (TSR) application or the operating system user interface. Hot keys generate a WM_HOTKEY message.

hot spot The pixel in a cursor that marks the exact screen location affected by a mouse or pen action, such as a button click. Messages include the coordinates of a hot spot.

HTML *See* **Hypertext Markup Language.**

HTTP *See* **Hypertext Transfer Protocol.**

Hypertext Markup Language (HTML) A markup language derived from the Standard Generalized Markup Language (SGML). Used to create a text document with formatting specifications that tells a software browser how to display the page or pages included in the document.

Hypertext Markup Language (HTML) viewer control **1.** A control that provides programmers with the ability to implement the Windows CE Pocket Internet Explorer and the Help engine. It also provides independent software vendors (ISVs) with the ability to implement additional viewers based on the HTML viewer control. **2.** A control that enables a user to render HTML text, display embedded images, and notify an application of user events.

Hypertext Transfer Protocol (HTTP) The client/server protocol used to access information on the Web.

I

IAS *See* **Information Access Service.**

ICMP *See* **Internet Control Message Protocol.**

icon A small image displayed on the screen to represent an object that can be manipulated by the

user. By serving as visual mnemonics and allowing the user to control certain computer actions without having to remember commands or type them at the keyboard, icons are a significant factor in the user-friendliness of graphical user interfaces.

IDE *See* **integrated development environment** and **Integrated Device Electronics.**

idle priority One of three thread priority groups. Idle priority indicates that a thread's processing can wait until all other threads have finished running. *See also* **interrupt priority** and **main priority.**

IEEE *See* **Institute of Electrical and Electronics Engineers.**

IHV *See* **independent hardware vendor.**

IIS *See* **Internet Information Server.**

IM *See* **input method.**

image list A collection of images that are all the same size, such as bitmaps or icons.

IME *See* **Input Method Editor.**

IMM *See* **Input Method Manager.**

Inbox A mail client application provided with Windows CE.

inclination A measurement that describes the angle formed between the plane defined by the orbit of a satellite and the equatorial plane of the earth. The value of the inclination must be between 0 and pi radians.

independent hardware vendor (IHV)
A company that manufactures devices that connect to Windows CE–based platforms, such as PC Card storage devices. IHVs must also produce stream interface device drivers for their devices. *See also* **stream interface device driver.**

.inf A CAB Wizard input file that specifies information about the application.

Information Access Service (IAS)
A part of an IrDA infrared communication protocol used so that devices can learn about the services offered by another device. *See also* **Infrared Data Association.**

infrared (IR) Of or relating to the range of invisible radiation wavelengths from about 750 nanometers, just longer than red in the visible spectrum, to 1 millimeter, on the border of the microwave region.

Infrared Data Association (IrDA) The industry organization of computer, component, and telecommunications vendors who have established the standards for infrared communication between computers and peripheral devices such as printers. Windows CE supports the IrDA standard through the Winsock application programming interface (API). Windows CE–based applications that communicate over serial cables using the Winsock API communicate over IrDA-compliant IR links with only minimal reprogramming.

Infrared Link Access Protocol (IrLAP)
A protocol, based on the High-level Data Link Control (HDLC) protocol, designed to control an infrared link. IrLAP provides for discovery of devices, their connection over an infrared link, and reliable data delivery between devices.

Infrared Link Management Protocol (IrLMP)
A service multiplexing protocol that provides for multiple sessions over a single point-to-point link.

Infrared Sockets (IrSock) An implementation of the Winsock protocol.

.ini An initialization file that registers an application with an application manager. It contains information such as the location of .cab files, icon files, and the installation directory.

initialization vector A random number used as a starting point when encrypting data. Two identical packets of data encrypted with the same key can result in two different packets of ciphertext, if each packet of data is encrypted with different initialization vectors.

input context An internal structure, maintained by the IME, that contains information about the status of the IME and is used by IME windows. By default, the system creates and assigns an input context to each thread. Within the thread, this default input context is a shared resource and is associated with each newly created window.

input method (IM) A COM component that allows the user to input text using a touch screen. For example, a Palm-size PC supports input methods for a graphical representation of a keyboard and a character recognizer.

Input Method Editor (IME) An engine that converts keystrokes into phonetic and ideograph characters, along with a dictionary of ideograph words, for conversion of characters into non-Roman, particularly Asian, characters.

Input Method Manager (IMM) A component supported in Windows CE version 2.10 and later that handles communication between IMEs and applications.

input panel A window control that supports various input methods, such as writing or drawing.

installable device driver *See* **stream interface device driver.**

installable file system A file system that is accessed through a file system driver (FSD) that is loaded onto a device by the user, rather than accessed by the built-in file system. An installable file system may be implemented differently from the built-in file system. For example, an installable file system may prevent a user from deleting files, provide automatic compression of files, or use a structure for internal information different from that used by the built-in file system. *See also* **file system driver.**

installable file system driver *See* **file system driver.**

Institute of Electrical and Electronics Engineers (IEEE)
An organization of professional electrical and electronics engineers that is notable for developing standards for hardware and software.

integrated development environment (IDE)
In the Microsoft Developer Studio, an integrated set of Windows-based tools for building, testing, and refining an application. The IDE includes a variety of editors, project build facilities, compilers, an incremental linker (for C++), a class viewer, and an integrated debugger. The IDE enables you to create, test, and refine your applications and Web sites all in one place.

Integrated Device Electronics (IDE)
A type of disk-drive interface in which the controller electronics reside on the drive itself, eliminating the need for a separate adapter card.

interface **1.** The point at which a connection is made between two elements so that they can work with one another. **2.** Software that enables an application to work with the user (the user interface, which can be a command-line interface, menu-driven, or a graphical user interface), with another application, such as the operating system, or with the computer's hardware. **3.** A card, plug, or other device that connects pieces of hardware with the computer so that information can be moved from place to place. **4.** A networking or communications standard that defines ways for different systems to connect and communicate.

Internet Control Message Protocol (ICMP)
A network-layer Internet protocol that provides error correction and other information relevant to Internet Protocol (IP) packet processing, such as testing whether a particular computer is connected

to the Internet (pinging) by sending a packet to its IP address and waiting for a response. For example, it can let the IP software on one machine inform another machine about an unreachable destination. *See also* **ping**.

Internet Information Server (IIS) Microsoft's brand of Web server software, using Hypertext Transfer Protocol (HTTP) to deliver World Wide Web documents. It incorporates various functions for security, allows for CGI applications, and also provides for Gopher and File Transfer Protocol (FTP) servers.

Internet Protocol (IP) Provides the protocol for connecting hosts over a network, breaking messages into packets, addressing the packets, routing them from the sender to the destination network, and reassembling the packets into the original message at the destination. IP corresponds to the network layer in the International Organization for Standardization Open Systems Interconnection (ISO/OSI) model. *See also* **ISO/OSI model**.

Internet Protocol (IP) address A 32-bit (4-byte) binary number that uniquely identifies a host computer connected to the Internet to other Internet hosts, for the purposes of communication through the transfer of packets. An IP address is expressed in "dotted quad" format, consisting of the decimal values of its four bytes, separated with periods; for example, 127.0.0.1. The first one, two, or three bytes of the IP address, assigned by InterNIC Registration Services, identify the network the host is connected to; the remaining bits identify the host itself.

internetwork Of or pertaining to communications between connected networks. Often used to refer to communication between one local area network and another over the Internet or another wide-area network.

interrupt A request for attention from the processor. When the processor receives an interrupt, it suspends its current operations, saves the status of its work, and transfers control to a special routine known as an interrupt handler, which contains the instructions for dealing with the particular situation that caused the interrupt. Interrupts can be generated by various hardware devices to request service or report problems, or by the processor itself in response to application errors or requests for operating-system services. Interrupts are the processor's way of communicating with the other elements that make up a computer system. A hierarchy of interrupt priorities determines which interrupt request will be handled first if more than one request is made. An application can temporarily disable some interrupts if it needs the full attention of the processor to complete a particular task.

interrupt identifier (interrupt ID) A unique value used by the kernel to identify the device that raised the interrupt and that requires more processing. The kernel then uses the interrupt identifier to indicate whether all handling is complete, or whether to launch an interrupt service thread that handles further processing by the device driver.

interrupt priority One of three thread priority groups. Interrupt priority is reserved for operating system threads. *See also* **idle priority** and **main priority.**

interrupt request line (IRQ) A hardware line over which a device, such as an I/O port, keyboard, or disk drive, can send interrupt requests to the CPU. Interrupt request lines are built into the computer's internal hardware and are assigned different levels of priority so that the CPU can determine the sources and relative importance of incoming service requests.

interrupt service routine (ISR) A small subroutine that resides in the OEM adaptation layer (OAL). The ISR executes in kernel mode and has direct access to the hardware registers. Its sole job is to determine what interrupt identifier to return to the

interrupt support handler. Essentially, ISRs map physical interrupts onto logical interrupts.

interrupt service thread (IST) A thread created by a device driver to wait on an event.

interrupt support handler A routine that registers a driver so that it can handle a particular interrupt and deregister it later. It also enables communication between the interrupt service routine, the interrupt service thread, and subroutines within the OEM adaptation layer (OAL).

I/O Input/output.

IP *See* **Internet Protocol.**

IR *See* **infrared.**

IrCOMM An infrared implementation of the serial line communication driver. IrCOMM is supported by Windows CE.

IrDA *See* **Infrared Data Association.**

IrLAP *See* **Infrared Link Access Protocol.**

IrLMP *See* **Infrared Link Management Protocol.**

IrLPT A protocol for printing through a serial infrared connection.

IRQ *See* **interrupt request line.**

IrSock *See* **Infrared Sockets.**

ISO/OSI model A layered architecture that standardizes levels of service and types of interaction for computers exchanging data through a communications network. The ISO/OSI model separates computer-to-computer communications into seven layers.

ISR *See* **interrupt service routine.**

IST *See* **interrupt service thread.**

ISV Independent software vendor.

item script An application written in HTML and Visual Basic Script, JScript, Java Script, or other scripting languages that specifies the behavior of an item within a channel.

K

kernel The main module of the Windows CE operating system. The kernel provides system services for managing threads, memory, and resources.

key A field or expression used to identify a record; often used as the index field for a database table.

key binary large object (key BLOB)
A key BLOB provides a way to store keys outside of the cryptographic service provider (CSP) and are used to transfer keys securely from one CSP to another. A key BLOB consists of a standard header followed by data representing the key.

key BLOB *See* **key binary large object.**

key container A place where cryptographic key pairs are stored. Each key container stores all of the key pairs belonging to a specific user.

keyboard accelerator **1.** In applications, a key or key combination used to perform a defined function. *Also called* shortcut key. **2.** In hardware, a device that speeds or enhances the operation of one or more subsystems, leading to improved application performance.

L

LAN *See* **local area network.**

launch entry A registry entry that specifies the order in which applications launch.

layered device driver A sample device driver that comes with the Platform Builder. It contains two

layers: a model device driver (MDD) layer and a platform-dependent driver (PDD) layer. *See also* **model device driver** and **platform-dependent driver.**

LBA *See* **logical block address.**

lead-in On an audio compact disc, the lead-in contains a table of contents for the track layout.

lead-out On an audio compact disc, the lead-out indicates the end of data.

linked font The font glyphs that you add to a base font when performing font linking. *See also* **base font.**

list box A control that enables the user to choose one option from a list of possibilities. The list box appears as a box, displaying the currently selected option, next to a button marked with a down arrow. When the user clicks or taps the button, the list appears. The list has a scroll bar if there are more options than the list has room to show.

list view A common control that displays a collection of items, such as files or folders. Each item has an icon and a label.

load file A file that contains a list of commands for the **Load** function to process. You use load file commands to direct Ppcload.dll to create directories on a Windows CE–based device, copy files into the directories, edit registry entries, execute applications on the Windows CE–based device, and add items to the unload script. The fully qualified path of the load file is given as a command-line argument to Load.

local address An address found on a local network.

local area network (LAN) A group of computers and other devices dispersed over a relatively limited area and connected by a communications link that enables any device to interact with any other device on the network. LANs commonly include microcomputers and shared resources such as laser printers and large hard disks. The devices on a LAN are known as nodes, and the nodes are connected by cables through which messages are transmitted.

localization The process of adapting an application for a specific international market, which includes translating the user interface, resizing dialog boxes, customizing features if necessary, and testing results to ensure that the application still functions properly.

logical block address (LBA) A method of expressing a data address on a storage medium, such as a compact disc.

logical font An application-defined description of a font. *See also* **font mapping.**

logical palette An array of colors, or a color palette, that an application creates and associates with a device context and uses for graphics output.

Logical Service Access Point (LSAP)
The point of access to a service or application within Infrared Link Management Protocol (IrLMP). This access point is referenced with an LSAP selector (LSAP-SEL).

Logical Service Access Point Selector (LSAP-SEL)
A 1-byte number that corresponds to a Logical Service Access Point (LSAP). This byte is broken into ranges for the Information Access Service (IAS) server, legal connections, connectionless service, and future use.

LSAP *See* **Logical Service Access Point.**

LSAP-SEL *See* **Logical Service Access Point Selector.**

M

main priority One of three thread priority groups. Main is the default priority. *See also* **idle priority** and **interrupt priority.**

main window The window that serves as the primary interface between a user and an application.

major axis A line whose length is one of the parameters used to describe the shape of an ellipse. The major axis has endpoints on the ellipse and passes through the two foci of the ellipse. *See also* **focus.**

MDD *See* **model device driver.**

MDI *See* **multiple-document interface.**

mean anomaly An angular measurement that specifies the position of a satellite within the orbit of that satellite. The time that the satellite takes to complete one orbit maps to 2 pi radians of mean anomaly. Zero radians corresponds to the perigee and pi radians corresponds to the apogee. The mean anomaly of any other point in the orbit is proportional to the amount of time that the satellite takes to travel from the perigee to that point.

menu A list of options from which a user can make a selection in order to perform a selected action, such as choosing a command or applying a particular format to part of a document. Many applications, especially those that offer a graphical interface, use menus as a means of providing a user with an easily learned, easy-to-use alternative to memorizing commands and their appropriate usage.

menu handle A unique value of type **HMENU** used to identify a menu.

menu item A string or bitmap displayed in a menu. Choosing a menu item either sends a command message or activates a pop-up menu.

menu template A menu template defines a menu, including the items on a menu bar and all submenus.

message A structure or set of parameters used for communicating information or a request. Messages can be passed between the operating system and an application, different applications, threads within an application, and windows within an application.

message box A secondary window that is displayed to inform a user about a particular condition.

message digest *See* **hash value.**

message handler A Component Object Model (COM) object that implements the **ITranslate** interface in an in-process COM object.

message identifier A unique value that identifies a message. System-defined messages use named constants, such as WM_PAINT, as message identifiers. Windows CE reserves message-identifier values in the range 0x0400 through 0x7FFF for application-defined messages.

message queue An ordered list of messages awaiting transmission, from which they are taken up on a first-in, first-out (FIFO) basis.

message sink A callback function that receives messages for a form or a control. Forms or controls that need to be notified of messages implement a message sink.

message store The database in the object store for storing mail messages.

MFC *See* **Microsoft Foundation Classes.**

Microsoft Foundation Classes (MFC)
The C++ class library that Microsoft provides with its C++ compiler to assist programmers in creating Windows-based applications. MFC hides the fundamental Windows API in class hierarchies so that programmers can write a Windows-based application without needing to know the details of the native Windows API.

Mobile Channels A Windows CE technology that represents a fourth type of Internet Explorer 4.0 (IE4) channel to allow the user to access the Web with great mobility.

modal dialog box A modal dialog box requires the user to supply information or close the dialog box before allowing the application to continue.

model device driver (MDD) The platform-neutral layer of a native device driver supplied by Microsoft. *See also* **native device driver.**

modeless dialog box A dialog box that allows the user to supply information and return to a previous task without closing the dialog box.

module A subset of the Windows CE operating system. Windows CE is structured as a collection of modules. Each module is a self-contained subset of the Windows CE operating system that can be used to construct a customized operating system for a particular device.

monolithic device driver A sample device driver that comes with the Microsoft Windows CE Platform Builder.

month calendar control A child window that displays a monthly calendar. The calendar can display one or more months at a time.

mounted file system A file system located on a removable medium, such as a PC Card storage device. The operating system loads, or mounts, the file system when the medium is inserted into the device. It unloads, or unmounts, the file system either when the medium is removed or when the user issues a command to do so.

MSF A data address format that displays a data address in minutes, seconds, and frames.

multicast A communication between a single sender and multiple receivers on a network.

multicast group Collectively, the hosts listening to a specific Internet Protocol (IP) multicast address.

multimedia driver A device driver that uses the Windows CE subset of the Win32 WDM device driver model.

multiple-document interface (MDI) A user interface in an application that allows a user to have more than one document open at the same time. MDI is not supported by Windows CE.

mutex object An interprocess synchronization object whose state is set to signaled when it is not owned by any thread, and nonsignaled when it is owned. Only one thread at a time can own a mutex object.

N

NaN Not a number.

national language support (NLS) A function that enables you to specify system and user locale information.

native device driver A software component that enables a computer system to communicate with a device. In Windows CE, a native device driver is linked with the GWES component. The driver consists of a model device driver (MDD) layer and a platform-dependent driver (PDD) layer. Together, these layers make it possible for applications to access physically different, but functionally equivalent, hardware resources in the same way on all Windows CE–based platforms. *Also called* a built-in device driver.

navigation control On a device, a moveable piece, such as a wheel or key, that sends virtual key codes. These virtual key codes are often used to move the cursor, alter the view of the current document, or launch a new application. A navigation control usually exists on a device that does not require a full hardware keyboard, such as a Palm-size PC.

NDIS *See* **Network Driver Interface Specification.**

NDIS driver A device driver for NDIS network adapters.

Network Driver Interface Specification (NDIS) A programming interface that allows different protocols to share the same network hardware.

network identifier An identifier for systems located on the same physical network.

network stack An operating system component responsible for processing data that is transmitted or received over a network.

NLS *See* **national language support.**

node **1.** In local area networks, a device that is connected to the network and is capable of communicating with other network devices. **2.** In tree structures, a location on the tree that can have links to one or more nodes below it. Some authors make a distinction between node and element, with an element being a given data type and a node comprising one or more elements as well as any supporting data structures.

nonclient area The parts of a window that are not a part of the client area. A window's nonclient area consists of the border, menu bar, title bar, and scroll bar.

nonsignaled *See* **synchronization object.**

notification A signal from the operating system that an event has occurred. This could be a timer event or a system event such as establishing a network connection. An application registers a notification for an event and the system generates a notification when the event occurs. Windows CE provides an application programming interface (API) that can be used to register events and select options that determine the type of notification.

notification function A Windows CE function that allows an application to register its name and an event with the system. When the event occurs, the kernel automatically starts the named application.

notification message A message that a control sends to its parent window when events, such as input from the user, occur.

O

OAL *See* **OEM adaptation layer.**

object A file, directory, database, or database record that resides in an object store.

object ID *See* **object identifier.**

object identifier **1.** A unique value that identifies each object in the object store. **2.** In reference to the Contacts database, an object identifier is a unique value that the system assigns to each address card when it is added. An application uses the object identifier when querying an address card's properties or when modifying or deleting an address card.

Object Linking and Embedding *See* **OLE.**

object store The persistent storage that Windows CE makes available to applications. For example, Windows CE reserves part of its available random access memory (RAM) for the operating system and uses the rest for the object store. This data can be stored in files, registry entries, or in Windows CE databases.

object type A name for a particular group of objects that are contained in a folder. For example, appointment is an object type naming all appointments in a Microsoft Schedule+ folder.

OEM *See* **original equipment manufacturer.**

OEM adaptation layer (OAL) That portion of Windows CE that must be provided by the hardware manufacture to adapt Windows CE to their platform.

OLE Object Linking and Embedding. A technology for transferring and sharing information among applications. When an object, such as an image file created with a painting application, is linked to a compound document, such as a spreadsheet or a document created with a word processing application, the document contains only a reference to the object; any changes made to the contents of a linked object are seen in the compound document. When an object is embedded in a compound document, the document contains a copy of the object; any changes made to the contents of the original object are not seen in the compound document unless the embedded object is updated. *See also* **Automation**.

option button In graphical user interfaces, a means of selecting one of several options, usually within a dialog box. An option button, also known as a radio button, appears as a small circle that, when selected, has a smaller, filled circle inside it. Option buttons act like the station selector buttons on a car radio. Selecting one button in a set deselects the previously selected button, so one and only one of the options in the set can be selected at any given time. In contrast, check boxes are used when more than one option in the set can be selected at the same time.

original equipment manufacturer (OEM) For Windows CE, an OEM is a company that manufacturers a hardware platform and ports Windows CE to that platform.

overlapped communication operation The performance of two distinct communication operations simultaneously; for example, a simultaneous read/write operation. Windows CE does not support overlapped communication operation, but does support multiple read/writes pending on a device.

overlapped window A window with the WS_OVERLAPPED style. Overlapped windows are top-level windows designed to serve as an application's main window.

P

packet A unit of information transmitted as a whole from one device to another on a network.

paint cycle The process of a control painting or erasing itself in response to messages received from the operating system.

palette A collection of colors that can be displayed on an output device.

parallel port The input/output connector for a parallel interface device.

parent window A window that has one or more child windows.

parser An application that breaks data into smaller chunks so that an application can act upon the information. For example, Mobile Channels use a Channel Definition Format parser to parse a channel.

pASP *See* **pocket Active Server Pages.**

path **1.** In communications, a link between two nodes in a network. **2.** A route through a structured collection of information, as in a database, an application, or files stored on disk. **3.** In programming, the sequence of instructions that a computer carries out in executing a routine. **4.** In file storage, the route followed by the operating system in finding, sorting, and retrieving files on a disk. **5.** In graphics, an accumulation of line segments or curves to be filled or overwritten with text.

PC Card storage device A trademark of the Personal Computer Memory Card International Association (PCMCIA) that is used to describe add-in cards that conform to the PCMCIA specification. A PC Card storage device is a

removable device approximately the same size as a credit card that is designed to plug into a PCMCIA slot. Type I cards are primarily used as memory-related peripherals. Type II cards accommodate devices such as modem, fax, and network cards. Type III cards accommodate devices that require more space, such as wireless communications devices and rotating storage media, including hard disks. **Flash card** is a general term for a PC Card storage device.

PCT *See* **program comprehension tool.**

PDD *See* **platform-dependent driver.**

peer Any of the devices on a layered communications network that operate on the same protocol level.

pen A drawing tool used to draw lines and curves.

perigee The point within the orbit of a satellite where the satellite is closest to the earth.

persistent object A COM object that adheres to standards through which clients can request objects to be initialized, loaded, and saved to and from a data store, such as a flat file, structured storage, or memory.

personal information manager (PIM) An application that usually includes an address book and organizes unrelated information, such as notes, appointments, and names, in a useful way.

phone book Entries in the Remote Access Service (RAS) phone book contain the information necessary to establish a RAS connection. Unlike Windows NT, which keeps the phone book entries in a file, Windows CE stores these entries in the registry.

physical font The font that is stored on a device or in an operating system. *See also* **font mapping.**

PIM *See* **personal information manager.**

ping A protocol for testing whether a particular computer is connected to the Internet by sending a packet to its Internet Protocol (IP) address and waiting for a response.

plaintext Data that has not been encrypted.

platform **1.** The foundation technology of a computer system. Because computers are layered devices composed of a chip-level hardware layer, a firmware and operating-system layer, and an applications layer, the bottommost layer of a machine is often called a platform. **2.** In everyday usage, the type of computer or operating system being used. **3.** The hardware upon which an implementation of Windows CE runs. **4.** The directory structure containing the hardware-specific files needed to build an implementation of Windows CE.

platform-dependent driver (PDD) The platform-specific layer of a native device driver supplied by an original equipment manufacturer. *See also* **native device driver.**

platform directory The root of the directory structure where platform-specific files are stored. Each subdirectory in the platform directory specifies the name of a development workstation.

pocket Active Server Pages (pASP) A scaled-down version of the Active Server Pages optimized for server-side Mobile Channels scripting.

Point-to-Point Protocol (PPP) An advanced serial packet protocol commonly used for dial-up connections.

POP3 *See* **Post Office Protocol 3.**

pop-up menu A menu that appears on the screen when a user selects a certain item. Pop-up menus can appear anywhere on the screen, and they

generally disappear when the user selects an item in the menu.

pop-up window A special type of overlapped window typically used for dialog boxes, message boxes, and other temporary windows that appear outside an application's main window.

Portable Operating System Interface for Computer Environments (POSIX)
An IEEE standard that defines the open systems environment standards for system interfaces, shells, tools, testing, verification, real-time processing, security, system administration, networking, and transaction processing. The standard is based on UNIX system services, but it allows implementation on other operating systems.

position index An identifier associated with each address card in the Contacts database. The position index indicates the address card's position relative to the other address cards in the database. A position index is distinct from an object identifier.

POSIX *See* **Portable Operating System Interface for Computer Environments.**

Post Office Protocol 3 (POP3) A standard protocol for transferring mail messages on demand from a mail server.

PPP *See* **Point-to-Point Protocol.**

predefined control A control belonging to a window class supplied by Windows CE.

pre-emphasis Increases the magnitude of the higher signal frequencies on an audio channel, improving the signal-to-noise ratio. *Compare* **de-emphasis.**

preemptive multitasking A form of multitasking in which the operating system periodically interrupts the execution of an application and passes control of the system to another waiting application. Preemptive multitasking prevents any one application from monopolizing the system.

prerecorded speech template A recording of a speech command in your application that is used for short voice Help.

principal An entity recognized by a security system. A principal can be a human user or an autonomous process.

priority class A range of thread priority levels. Whereas Win32 utilizes four priority classes with seven base priority levels per class, Windows CE has only eight base priority levels. Hence, for processes running under Windows CE, preemption is based solely on the thread's priority.

priority inheritance A process by which a thread that is blocking a shared resource needed by a higher-priority thread inherits the priority of that higher-priority thread in order to free the resource for use by the higher-priority thread, thus preventing *priority inversion.*

priority inversion Priority inversion is a situation in which higher-priority thread A spawns lower-priority thread B to access a shared resource that is already in use by lower-priority thread C with greater priority than thread B, blocking higher-priority thread A. This situation can be averted by a process of *priority inheritance.*

process A running application that consists of a private virtual address space, code, data, and other operating-system resources, such as files, pipes, and synchronization objects that are visible to the process. A process also contains one or more threads that run in the context of the process.

program button On a device, a navigation control that is pressed to launch an application. The program button can also be programmed for additional features, such as creating a new document in an application.

program comprehension tool (PCT)
A software engineering tool that facilitates the process of understanding the structure and/or functionality of computer applications.

program memory Program memory is used for stack and heap storage for both system and non-system applications. Non-system applications are taken from storage memory, uncompressed, and loaded into program memory for execution.

progress bar A common control that indicates the progress of a lengthy operation by displaying a colored bar inside a horizontal rectangle. The length of the bar in relation to the length of the rectangle corresponds to the percentage of the operation that is complete.

project **1.** The implementation of an instance of Windows CE. **2.** The directory structure, under Public, containing files that define which components will be included in an implementation of Windows CE.

property With respect to the database application programming interface, a property refers to a data item that consists of a property identifier, data type, and value. Windows CE supports several data types such as integer, string, time, and binary large object (BLOB).

property sheet A type of dialog box that lists the attributes or settings of an object, such as a file, application, or hardware device. A property sheet presents the user with a tabbed, index card–like selection of property pages, each of which features standard dialog box–style controls for customizing parameters.

protocol stack Collectively, the layers of communications software in the ISO/OSI model.

public-key encryption An asymmetric scheme that uses a pair of keys for encryption: The public key encrypts data, and a corresponding secret key decrypts it. For digital signatures, the process is reversed: The sender uses the secret key to create a unique electronic number that can be read by anyone possessing the corresponding public key, which verifies that the message is truly from the sender.

pull-down menu A menu containing commands that are accessed from a command or menu bar. A pull-down menu usually provides access to a small number of items with content that rarely changes.

push button A small rectangular control that a user can turn on or off. A push button, also known as a command button, has a raised appearance in its default off state and a depressed appearance when it is turned on.

Q

queued message A message in a message queue.

QWERTY keyboard A keyboard layout named for the six leftmost characters in the top row of alphabetic characters on most keyboards—the standard layout of most typewriters and computer keyboards.

R

radio button *See* **option button.**

RAM *See* **random access memory.**

random access memory (RAM) Semiconductor-based memory that can be read and written by the CPU or other hardware devices.

RAPI *See* **remote application programming interface.**

RAS *See* **remote access server** and **Remote Access Service.**

raster font A font in which each glyph—a character or symbol—is of a particular size and style, designed for a specific resolution of device and described as a unique bitmap. There are seven

system raster fonts available in several sizes stored in the Windows CE read-only memory (ROM). The built-in fonts are built into the Windows CE operating system. *Also called* bitmap fonts and non-scalable fonts.

raw infrared (raw IR) A method of receiving data through an infrared transceiver. Raw IR treats the IR transceiver like a serial cable and does not process data in any way. The application is responsible for handling collision detection and other potential problems.

raw IR *See* **raw infrared.**

read-only memory (ROM) Any semiconductor circuit serving as a memory that contains instructions or data that can be read but not modified, regardless of whether it was placed there by a manufacturer or by a programming process.

rebar control A rebar control acts as a container for child windows. A rebar control contains one or more bands. Each band can contain one child window, which can be a toolbar or any other control.

record A data structure that is a collection of elements, each with its own name and type. The elements of a record represent different types of information and are accessed by name. A record can be accessed as a collective unit of elements, or the elements can be accessed individually. A collection of records is a database. A Windows CE database consists of an arbitrary number of records, where each record consists of one or more properties. Each of the records in a specific database typically contain a similar set of properties. A Windows CE database should not be confused with a full-fledged relational database. It is simply a general-purpose, flexible, structured collection of data.

rectangle A function that draws a rectangular image.

Red Book audio A data format standard for an audio compact disc.

redirector A module through which one computer accesses another.

reentrant code Code written so that it can be shared by several applications or threads within a single process simultaneously. When code is reentrant, one thread can safely interrupt the execution of another thread, execute its own code, and then return control to the first thread in such a way that the first thread does not fail or behave in an unexpected way.

region A rectangle, polygon, ellipse, or a combination of two or more of these shapes used by Windows-based applications to define a part of the client area to be painted, inverted, filled with output, framed, or used for hit testing.

registered notification The state of a user notification from the time **CeSetUserNotification** is called until the time the user is notified.

registry A central hierarchical database used to store information necessary to configure the system for applications and hardware devices. The registry contains information—such as the applications installed on the computer and the types of documents each can create, property sheet settings for folders and application icons, what hardware exists on the system, and which ports are being used—that the operating system continually references during operation.

remote access server (RAS) A Windows NT feature by which a single serial connection provides a remote workstation with host connectivity, NT file services, or Novell file and printing services (NWLink). Windows CE supports the standard Win32 RAS functions; however, it allows only one connection at a time. RAS functions can be implemented for direct serial connections or dial-up modem connections. *See also* **Remote Access Service.**

Remote Access Service (RAS) Windows software that allows a user to gain remote access to the network server by means of a modem. *See also* **remote access server.**

remote address An address not found on a local network.

remote application programming interface (RAPI) Enables an application running on a desktop computer to make function calls on a Windows CE–based device. The desktop computer is known as the RAPI client and the Windows CE device is known as the RAPI server. RAPI runs over Winsock and TCP/IP.

Request for Comments (RFC) A document in which a standard, a protocol, or other information pertaining to the operation of the Internet is published.

resistive touch panel A transparent, touch-sensitive surface implemented to detect user input. *See* **touch screen.**

resource **1.** Any part of a computer system or a network, such as a disk drive, a printer, or memory, that can be allotted to an application or a process while it is running. **2.** An item of data or code that can be used by more than one application or in more than one place in an application, such as a dialog box, a sound effect, or a font in a windowing environment. Many features in an application can be altered by adding or replacing resources without the necessity of recompiling the application from source code. Resources can also be copied and pasted from one application into another, typically by a specialized utility application called a *resource editor.*

RFC See **Request for Comments.**

Rich Ink The underlying technology that enables a user to write and draw on a touch-sensitive screen by using a stylus.

right ascension The angle as measured from the center of the earth between a satellite and the vernal equinox. The right ascension must have a value between 0 and 2 pi radians. *See also* **vernal equinox.**

RLE *See* **run-length encoding.**

rocker switch A hardware navigation control designed to perform spatial navigation, much like the UP ARROW key and the DOWN ARROW key.

ROM *See* **read-only memory.**

ROM image Files and binaries as they appear in physical memory as defined by the binary image builder (.bib) file.

router An intermediary device on a communications network that expedites message delivery. On a single network linking many computers through a mesh of possible connections, a router receives transmitted messages and forwards them to their correct destinations over the most efficient available route. On an interconnected set of local area networks using the same communications protocols, a router serves the somewhat different function of acting as a link between these local area networks, enabling messages to be sent from one network to another.

run-length encoding (RLE) A simple compression method that replaces a contiguous series (run) of identical values in a data stream with a pair of values that represent the length of the series and the value itself. For example, a data stream that contains 57 consecutive entries with the value 10 could replace them all with the much shorter pair of values 57, 10.

S

salt value Random data used to supplement encryption schemes. A salt value allows two identical packets of data to be encrypted into two

different packets of ciphertext using the same key by changing the salt value with each packet.

satellite azimuth An angular measure of the horizontal direction of a satellite relative to an observer on Earth. The value of the satellite azimuth must be between 0 and 2 pi radians.

satellite elevation The angular position of a satellite above the plane that is tangent to the earth at the position of the observer. The value of the satellite elevation must be between 0 and one-half pi radians.

scan code A code number transmitted to a computer whenever a key is pressed or released. Each key on the keyboard has a unique scan code. This code is not the same as the ASCII code for the letter, number, or symbol shown on the key; it is a special identifier for the key itself and is always the same for a particular key. When a key is pressed, the scan code is transmitted to the computer, where a portion of the read-only memory basic input/output system (ROM BIOS) dedicated to the keyboard translates the scan code into its ASCII equivalent. Because a single key can generate more than one character—lowercase "a" and uppercase "A," for example—the ROM BIOS also keeps track of the status of keys that change the keyboard state, such as the SHIFT key, and takes them into account when translating a scan code.

score When referring to a spelling checker, a score is a number that indicates how much a replacement word differs from the original misspelled word. A low score indicates that the misspelled word was changed slightly, while a high score indicates that the word was changed a great deal.

script An application consisting of a set of instructions to an application or utility application. The instructions usually use the rules and syntax of the application or utility.

scripting language A simple programming language designed to perform special or limited tasks, sometimes associated with a particular application or function. An example of a scripting language is Visual Basic Script.

scroll bar In some graphical user interfaces, a vertical or horizontal bar at the side or bottom of a display area that can be used with a mouse for moving around in that area. Scroll bars often have four active areas: two scroll arrows for moving line by line, a sliding scroll box for moving to an arbitrary location in the display area, and the gray areas in the scroll bar for moving in one-window increments.

scrolling menu A menu with top arrows used to scroll menu items up and down.

secure socket layer (SSL) A proposed open standard developed by Netscape Communications for establishing a secure communication channel to prevent the interception of critical information, such as credit card numbers. The primary purpose of the SSL is to enable secure electronic financial transactions on the Web, although it is designed to work with other Internet services as well.

security context The security data relevant to a connection. A security context contains information such as a session key and the duration of a session. Both the client and server in a communication link must cooperate to create a security context.

security package A security solution that maps Security Support Provider Interface (SSPI) functions to the security protocols specified in a package.

Security Support Provider (SSP) A dynamic-link library (DLL) containing common authentication and cryptographic data schemes.

separator A blank space used to divide toolbar elements into groups or to reserve space in a command bar.

serial cable A cable that connects to a serial port. It is used to transfer information between two devices. *See also* **serial port.**

Serial Infrared (SIR) Part of the basic Infrared Data Association (IrDA) communication protocol, a Serial Infrared physical layer provides for serial infrared links.

serial I/O A communications channel that transmits data one bit at a time.

serialize The process of converting an object to a series of bytes for transmission to another device. *Compare* **deserialize.**

Serial Line Internet Protocol (SLIP) A data link protocol that allows transmission of Internet Protocol (IP) data packets over dial-up telephone connections, thus enabling a computer or a local area network to be connected to the Internet or some other network.

serial port An input/output location (channel) that sends and receives data to and from a computer's central processing unit or a communications device one bit at a time. Serial ports are used for serial data communication and as interfaces to peripheral devices, such as mouse devices and printers.

server **1.** On a local area network (LAN), a computer running administrative software that controls access to the network and its resources, such as printers and disk drives, and provides resources to computers functioning as workstations on the network. **2.** An application that responds to requests from another application or task. *See also* **client.**

service identifier An identifier used by a service to uniquely identify messages. This value should be changed only by the service library.

service manager A synchronization engine that resides on both the desktop computer and the device. The service manager performs many common synchronization tasks, which include providing connectivity, detecting changes in data, and resolving data conflicts, as well as mapping and transferring data objects.

service provider When referring to ActiveSync technology, a service provider is a pair of DLLs that a developer must implement in an application in order to perform synchronization tasks. One module, called the desktop provider module, resides on the desktop computer and the other module, called the device provider module, resides on the device. *See also* **desktop provider module** and **device provider module.**

session identifier An identifier generated by a mail transport service. Each time a Post Office Protocol 3 (POP3) connection is made to the server, the server looks at all of the currently stored messages and assigns a session identifier to each message, numbered 1 through the total number of messages. This makes it easier to reference a particular message without having to use its long unique identifier. The session identifier can be trusted only during a single connection to the mail server.

session key A key used in symmetric encryption schemes where a single key is used to both encrypt and decrypt data.

SGML *See* **Standard Generalized Markup Language.**

shared directory On a local area network, a directory on a disk that is located on a computer other than the one a user is operating. A shared directory differs from a network drive in that a user has access to only that directory.

shared library Any code module that can be accessed and used by many applications. Shared libraries are used primarily for sharing common code between different executable files or for breaking an application into separate components, thus allowing easy upgrades. In Windows CE,

shared libraries are usually referred to as dynamic-link libraries (DLLs).

shell An application that enables the user to connect with the kernel and, thus the system, usually providing some basic services in addition to facilitating the loading and executing of applications.

sibling window A child window that has the same parent window as one or more other child windows.

signaled *See* **synchronization object.**

signature file A file that ensures that a cryptographic service provider (CSP) will be recognized by the operating system.

silkscreen button A section of a resistive touch panel with a painted icon. An OEM provides a driver that lets this section of the panel send virtual-key messages. A silkscreen button is considered a navigation control.

silkscreen region A section of a resistive touch panel that contains several silkscreen buttons.

Simple Mail Transfer Protocol (SMTP) A TCP/IP protocol for sending messages from one computer to another on a network. This protocol is used on the Internet to route e-mail. *See also* **TCP/IP.**

single threading model A model in which all objects are executed on a single thread. *Contrast* **multithreaded application**; *see also* **free threading model, apartment threading model.**

SIR *See* **Serial Infrared.**

SLIP *See* **Serial Line Internet Protocol.**

SMTP *See* **Simple Mail Transfer Protocol.**

socket An object that represents an endpoint for communication between processes across a network transport. Sockets have a datagram or stream type and can be bound to a specific network address. Windows Sockets provides an application programming interface (API) for handling all types of socket connections in Windows.

sort order The order in which a set of records or other data objects are to be sorted, or the function that defines this order. Possible sort orders for an array of strings could include alphabetical order or ascending order by length, for example.

sound scheme A collection of audio effects, such as clicks and beeps, associated with system and application key events.

speaker-dependent template A recording of a speech command created by the user of a speech recognition system to train the system to recognize the command. Using a speaker-dependent template, a speech recognition system recognizes only the user who trained the word. *See also* **speaker-independent template.**

speaker-independent template A synthesis of many speakers' recorded pronunciation of a word or phrase. Using a speaker-independent template, a speech recognition system recognizes most speakers. *See also* **speaker-dependent template.**

speech recognition The ability of a computer to understand the spoken word for the purpose of receiving commands and data input from the speaker.

spelling session The resources that a spelling checker uses for a particular application, including dictionaries and created structures.

spin button control A control containing a pair of arrow buttons that a user can tap with the stylus to increment or decrement a value. A spin button control is most often used with a companion control, called a buddy window, in which a current value is displayed. *See also* **up-down control.**

SSL *See* **secure socket layer.**

SSP *See* **Security Support Provider.**

stack A region of reserved memory in which applications store status data such as procedure and function call addresses, passed parameters, and sometimes local variables.

Standard Generalized Markup Language (SGML) An information-management standard adopted by the International Organization for Standardization (ISO) in 1986 as a means of providing platform-independent and application-independent documents that retain formatting, indexing, and linked information. SGML provides a grammar-like mechanism for users to define the structure of their documents and the tags they will use to denote the structure in individual documents.

static control A control used to display text, to draw frames or lines separating other controls, or to display icons. A static control does not accept user input.

status bar A space at the bottom of many application windows that contains a short text message about the current condition of the application. Some applications also display an explanation of the currently selected menu command in the status bar.

storage memory Storage memory is similar to a RAM disk on a desktop computer. It is used to store data and nonsystem applications.

stream cipher mode A method of encryption where data is encrypted one bit at a time. *Compare* **block cipher mode.**

stream interface device driver A user-level DLL that controls devices connected to a Windows CE–based platform. A stream interface device driver presents the services of a hardware device to applications by exposing Win32 stream interface functions. Stream interface drivers also can control devices built into a Windows CE–based platform, depending on the software architecture for the drivers. *Also called* installable device driver.

stream mode An asynchronous method of calling **CeRapiInvoke** by using an **IStream** type interface to exchange arbitrary-size data in any order and direction.

stylus A pointing device used on a touch-sensitive surface.

subfolder A directory, or logical grouping of related files, within another directory.

submenu A menu that appears as the result of the selection of an item on a higher-level menu.

subnet mask *See* **address mask.**

subnetwork An identifiable and separate part of an organization's network identified through Internet Protocol (IP) addresssing.

symbol A name that represents a register, an absolute value, or a memory address (relative or absolute).

symmetric encryption A type of encryption where the same key is used to encrypt and decrypt data.

synchronization The process of updating information between the desktop computer and a Windows CE–based device to ensure that data is the same on both computers.

synchronization object An object whose handle can be specified in one of the wait functions to coordinate the execution of multiple threads. A synchronization object will be a member of one of the synchronization classes. Synchronization classes are used when access to a resource must be controlled to ensure integrity of the resource. The state of a synchronization object is either signaled, which can allow the wait function to return, or nonsignaled, which can prevent the function from returning. More than one process

can have a handle of the same synchronization object, making interprocess synchronization possible. There are four types of synchronization objects: mutex, semaphore, event, and critical section. Of these, Windows CE supports mutex, event, and critical section.

synchronous operation **1.** Two or more processes that depend upon the occurrences of specific events such as common timing signals. **2.** Data transmission method in which there is constant time between successive bits, characters, or events. The timing is achieved by the sharing of a single clock. Each end of the transmission synchronizes itself with the use of clocks and information sent along with the transmitted data. Characters are spaced by time, and not by start and stop bits. **3.** A function call that blocks execution of a process until it returns. *Compare* **asynchronous operation.**

sysgen phase Refers to the process of defining and building the selected modules and components, as governed by the Makefile located in the directory %_PUBLICROOT%\Common\Cesysgen.

system-defined message A message that the system uses to control the operations of an application and to provide input and other information for an application to process. An application can also send or post a system-defined message. An application generally uses this message to control the operation of control windows created by using preregistered window classes.

system registry functions The functions used to manipulate keys and values in the registry. A Windows CE–based application uses the standard Win32 registry functions.

T

tab control A control that is analogous to a set of dividers in a notebook or labels in a file cabinet. A tab control is used in a property sheet to provide a way for a user to move from one property page to another.

TAPI *See* **Telephony Application Programming Interface.**

target platform The system for which Windows CE is being adapted.

TCP/IP *See* **Transmission Control Protocol/Internet Protocol.**

telephony Telephone technology; the conversion of sound into electrical signals, its transmission to another location, and its reconversion to sound, with or without the use of connecting wires.

Telephony Application Programming Interface (TAPI)
A set of functions in the Win32 API that lets a computer communicate directly with telephone systems. Windows CE supports TAPI version 1.5. It provides the basic functions, structures, and messages for establishing outgoing calls and controlling modems from a Windows CE–based device.

Telephony Service Provider (TSP)
A modem driver that enables access to vendor-specific equipment through a standard device driver interface. *See also* **Telephony Service Provider Interface (TSPI).**

Telephony Service Provider Interface (TSPI)
The external interface of a service provider to be implemented by vendors of telephony equipment. A telephony service provider accesses vendor-specific equipment through a standard device driver interface. Installing a service provider allows Windows CE–based applications that use elements of telephony to access the corresponding telephony equipment. *See also* **Telephony Service Provider (TSPI).**

TEXT A Win32 macro that exists so that code can be compiled either as American Standard Code for Information Interchange (ASCII) text or as

Unicode. For Windows CE, which supports only Unicode, the macro forces the compiler to convert ASCII characters to Unicode characters. For example, passing the ASCII string "Hello Windows CE!'' through the **TEXT** macro converts all characters in the string to 16-bit wide characters.

text normalization Changing how words are pronounced based on their context.

text-to-speech (TTS) The conversion of text-based data into voice output by speech synthesis devices. Text-to-speech allows users to gain access to information audibly.

thread A process that is part of a larger process or application. A thread can execute any part of an application's code, including code that is currently being executed by another thread. All threads share the virtual address space, global variables, and operating-system resources of their respective processes.

thread identifier The unique identifier associated with a specific thread. Note that thread identification numbers are reused; they identify a thread only for the lifetime of that thread.

thread local storage (TLS) A Win32 mechanism that allows multiple threads of a process to store data that is unique for each thread. For example, a spreadsheet application can create a new instance of the same thread each time the user opens a new spreadsheet. A dynamic-link library that provides the functions for various spreadsheet operations can use thread local storage to save information about the current state of each spreadsheet.

thread synchronization The method used to coordinate the execution of two or more threads. There are two states in synchronization, signaled and nonsignaled. Threads can either modify the state of the synchronization object or wait for the object to reach a signaled state.

time-out value A specified time interval used by a timer. Each time the time-out value elapses, Windows CE sends a WM_TIMER message to the window associated with the timer.

timer An internal routine that causes the system to send a WM_TIMER message whenever a specified interval elapses.

timestamping The process of attaching the date and time to a message.

Time to Live A header field for a packet sent over the Internet indicating how long the packet should be held.

TLB *See* **translation look-aside buffer.**

TLS *See* **thread local storage.**

toolbar A row, column, or block of on-screen buttons or icons. When these buttons or icons are depressed, macros or certain functions of the application are activated.

ToolTip A small rectangular pop-up window that displays a brief description of a command bar button's purpose.

top-level window A window that has no parent window.

topmost window A window with the WS_EX_TOPMOST style. A topmost window overlaps all other non-topmost windows.

touchpad An input device that functions like a mouse to control cursor movements.

touch panel *See* **touch screen.**

touch screen A computer screen on which the user selects options, such as from a menu, by touching the screen. The touch screen is composed of an LCD and a resistive touch panel.

trackbar A common control, also known as a slider control, that consists of a bar with tick marks on it and a slider, also known as a thumb.

When a user drags the slider or clicks on either side of it, the slider moves in the appropriate direction, tick by tick.

traffic The load carried by a communications link or channel.

translation look-aside buffer (TLB)
A table used in a virtual memory system that lists the physical address page number associated with each virtual address page number. A TLB is used in conjunction with a cache whose tags are based on virtual addresses. The virtual address is presented simultaneously to the TLB and to the cache so that cache access and virtual-to-physical address translation can occur simultaneously.

Transmission Control Protocol/Internet Protocol (TCP/IP)
A protocol developed by the Department of Defense for communications between computers. It is built into the UNIX system and has become the de facto standard for data transmission over networks, including the Internet. TCP and IP are transport and address protocols; TCP is used to establish a connection for data transmission, and IP defines the method for sending the data in packets.

transport functions A set of functions, exported by a mail transport service dynamic-link library, that are used to transfer mail messages from one location to another.

tree-view control A hierarchical display of labeled items. The top item in the hierarchy is called the root. If an item has other items below it in the hierarchy, it is also referred to as a parent. Items subordinate to parents are called children. Child items, when displayed, are indented below their parent item. The hierarchy may be expanded or collapsed at any level to display or hide child items.

TrueType A scalable outline font whose glyphs are stored as a collection of line and curve commands, plus a collection of hints.

TSP *See* **Telephony Service Provider.**

TSPI *See* **Telephony Service Provider Interface.**

TTS *See* **text-to-speech.**

U

UNC *See* **Universal Naming Convention.**

unicast A communication between a single sender and a single receiver on a network.

Unicode A 16-bit character set capable of encoding almost all known characters and used as a worldwide character-encoding standard. Windows CE uses Unicode exclusively at the system level.

Uniform Resource Identifier (URI)
See **Uniform Resource Locator.**

Uniform Resource Locator (URL)
The address of a resource on the Internet. URL syntax is in the form *protocol://host/localinfo*, where protocol specifies the means of returning the object, such as Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). Host specifies the remote location where the object resides, and localinfo is a string—often a file name—passed to the protocol handler at the remote location. *Also called* Uniform Resource Identifier (URI).

Unimodem **1.** The universal modem driver, provided with Windows CE, that translates Telephony Service Provider Interface (TSPI) calls into AT commands, and sends the commands to a virtual device driver that talks to the modem. **2.** A universal modem that supports standard modem AT commands. Windows CE currently supports only PCMCIA modems.

universally unique identifier (UUID)
A 128-bit value that uniquely identifies objects such as OLE servers, interfaces, manager entry-point vectors, and client objects. Universally

unique identifiers are used in cross-process communication, such as remote procedure calling (RPC) and OLE. *Also called* globally unique identifier (GUID).

Universal Naming Convention (UNC) The system of naming files among computers on a network so that a file on a given computer will have the same path when it is accessed from any of the other computers on the network. For example, if the directory c:\path1\path2\...pathn on computer servern is shared under the name pathdirs, a user on another computer would open \\servern\pathdirs\filename.ext to access the file c:\path1\path2\...pathn\filename.ext on servern. *See also* **Uniform Resource Locator.**

universal serial bus (USB) A serial bus with a bandwidth of 1.5 megabits per second (Mbps) for connecting peripherals to a microcomputer. USB can connect up to 127 peripherals, such as external CD-ROM drives, printers, modems, mouse devices, and keyboards, to the system through a single, general-purpose port. This is accomplished by daisy-chaining peripherals together. USB supports hot plugging and multiple data streams. Developed by Intel, USB competes with DEC's ACCESS.bus for lower-speed applications.

up-down control A control containing a pair of arrow buttons that a user can click to increment or decrement a value, such as a scroll position. When used with an edit control or other type of companion control, an up-down control is referred to as a spin button. *See* **spin button control.**

URI *See* **Uniform Resource Locator.**

URL *See* **Uniform Resource Locator.**

USB *See* **universal serial bus.**

USB driver A device driver for USB-compatible devices.

user level driver *See* **stream interface device driver.**

user notification A warning to the user that a timer event has occurred. The notification may require the user to perform some action to handle the notification or may generate a sound to alert a user. For example, the system may display a dialog box and play a sound or display an icon before a scheduled appointment. The user would tap the dialog box **OK** button to acknowledge the appointment. User notifications are always associated with an application.

UUID *See* **universally unique identifier.**

V

vernal equinox A point that represents the apparent ascending node of the sun in the apparent orbit of the sun around the earth. For modeling purposes, pretending that the sun orbits the earth, rather than that the earth orbits the sun, simplifies the descriptions of satellite orbits. *See also* **ascending node.**

vehicle bus bridge A hardware interface that connects between an Auto PC's universal serial bus (USB) port and an automobile's on-board diagnostic level II (OBD II) port.

virtual-key code A device-independent value that identifies the purpose of a keystroke as interpreted by the Windows keyboard device driver.

W

wait function Allows a thread to block its own execution. Wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters an efficient wait state, consuming very little processor time while waiting for the criteria to be met. Windows CE supports only single object wait functions.

warm boot Restarting a running computer without first turning off the power. *Also called* soft boot, warm start. *Compare* **cold boot.**

.wav The file extension that identifies sound files stored in waveform (WAV) audio format.

Web Browser ActiveX control An ActiveX control that programmers can use to add Internet browsing capabilities to applications.

Win32 The application programming interface in Windows 95, Windows NT, and Windows CE that enables applications to use the 32-bit instructions available on 80386 and higher processors.

window A rectangular area on the screen where an application displays output and receives user input. On a Windows CE–based device that supports a graphical display, a window—rather than the screen itself—is the primary output device. Windows are also the means by which applications send and receive messages to the operating system. Therefore, all Windows CE–based applications—even those that lack a visual interface—need to create and manage windows.

window class A set of attributes that Windows CE uses as a template to create a window. Each window class has a window procedure that processes messages for all windows of that class. Every window in a Windows CE–based application is a member of a window class.

window control A predefined child window used in conjunction with another application window to provide a standardized way for users to make selections, carry out commands, and perform input and output tasks. Windows controls typically send WM_COMMAND messages.

window coordinate The position of a window in relation to the upper-left corner of the screen or, for a child window, the upper-left corner of the parent window's client area.

window handle A 32-bit value, assigned by Windows CE, that uniquely identifies a window.

window procedure A function, called by the operating system, that controls the appearance and behavior of its associated windows. The procedure receives and processes all messages to these windows.

Windows CE Services A set of technologies that makes Windows CE–based devices Web-enabled. Its architecture is based on a multilayered client/server model that provides the functionality to deliver Web content information to Windows CE–based devices from a wireless network or by desktop synchronization.

Windows Internet Naming Service (WINS) A distributed database for registering and querying dynamic name-to-IP address mappings in a routed network environment. When dynamic addressing through DHCP results in new IP addresses for computers that move between subnets, the changes are automatically updated in the WINS database.

Windows Sockets (Winsock) A programming interface used to provide a protocol-independent transport interface. Windows CE supports most of the common Winsock functions.

window style A named constant that defines an aspect of the window's appearance and behavior not specified by the window's class.

WinInet An API that provides Internet access to applications using Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP).

WINS *See* **Windows Internet Naming Service.**

Winsock *See* **Windows Sockets.**

Wireless Push Server (WPS) *See* **Wireless Services server component.**

Wireless Services client component
Decodes and processes messages from the Wireless Push Server (WPS). The client architecture allows for new decoding components to be installed at any time.

Wireless Services server component
Allows a content provider or carrier to configure and schedule any number of information acquisition/encoding/transmission components to create a data stream to be transmitted by a carrier to the device. The server component builds on an open architecture to allow new server components to be installed in any part of the stream at any time.

WPS *See* **Wireless Services server component.**

wrapper function A function that provides a simplified interface to another function, for example by changing the order of some parameters or by interpreting the return code.

X

X.509 An international message-handling standard for message authentication and encryption. X.509 is published by the Internal Telecommunications Union (ITU), formerly the International Telegraph and Telephone Consultative Committee (CCITT) standards body.

XIP *See* **execute in place.**

Z

z-order A stack of overlapping windows. Each window has a unique position in the z-order.

zeroth order clock correction A rough numerical measure of the bias in the time reported by a Global Positioning System (GPS) satellite.

Index

D

E

F

G

H

I

R

S

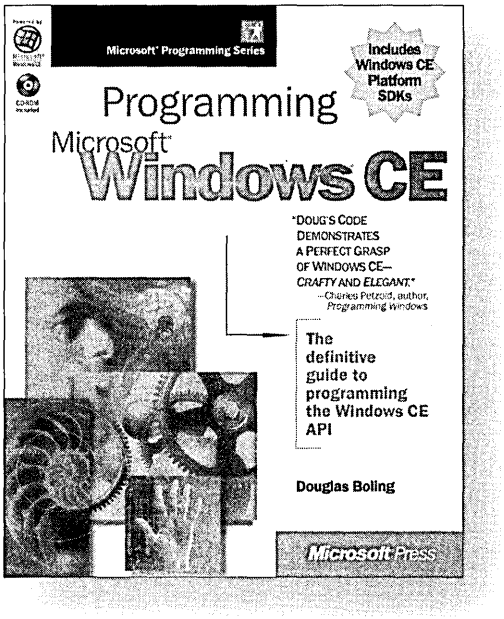
T

U

V

W

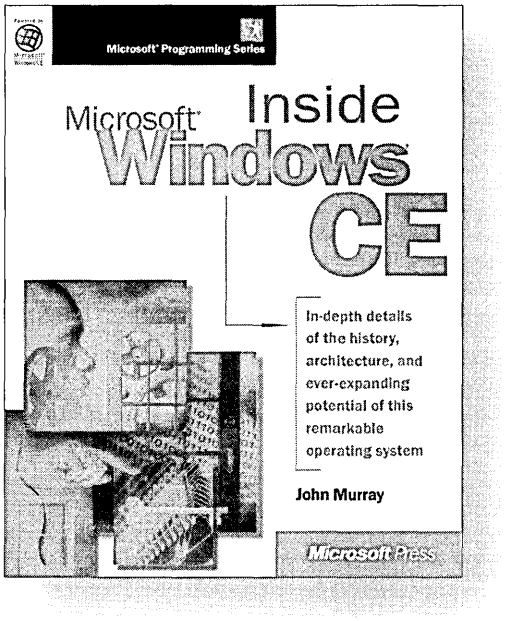
The **definitive guide** to programming the **Windows CE API**



U.S.A. **$49.99**
U.K. £46.99 [V.A.T. included]
Canada $71.99
ISBN 1-57231-856-2

Design sleek, high-performance applications for the newest generation of smart devices with PROGRAMMING MICROSOFT® WINDOWS® CE. This practical, authoritative reference explains how to extend your Windows or embedded programming skills to the Windows CE environment. You'll review the basics of event-driven development and then tackle the intricacies and idiosyncrasies of Windows CE's modular, compact architecture. With Doug Boling's expert guidance and the software development tools on CD-ROM, you'll have everything you need to mobilize your Win32® programming efforts for exciting new markets!

Get *moving* with **Windows CE.**



U.S.A. **$29.99**
U.K. £27.49 [V.A.T. included]
Canada $42.99
ISBN 1-57231-854-6

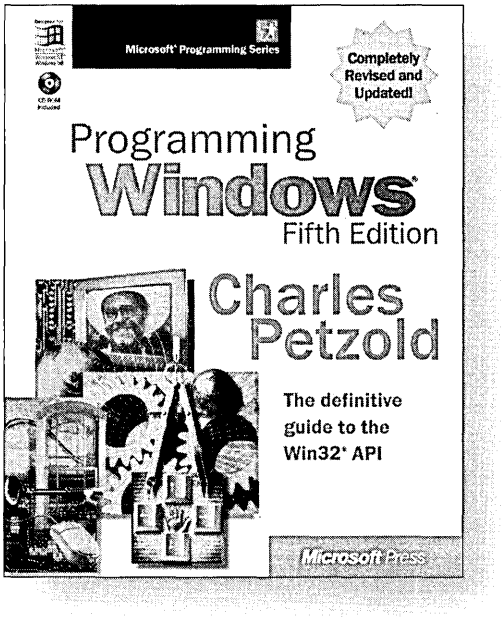
From roadside computing and pocket PCs to smart appliances and rich multimedia home theater, Microsoft® Windows® CE opens dynamic new development vistas for work, home, and everywhere in between. This modular, customizable operating system extends the Windows platform far beyond the desktop to the realm of smaller, mobile, and more specialized devices—while its Windows pedigree ensures compatibility and support for an expansive developer base. Find conceptual frameworks to help you understand your design options, and see real-world examples that demonstrate the flexibility and potential of this remarkable operating system. INSIDE MICROSOFT WINDOWS CE is the developer's key to understanding how Windows CE will spring new computing concepts into motion.

Microsoft Press® products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at mspress.microsoft.com. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the U.S. (in Canada, call 1-800-268-2222).

Prices and availability dates are subject to change.

Microsoft®
mspress.microsoft.com

The *definitive guide* to the **Win32 API**



"Look it up in Petzold" remains the decisive last word in answering questions about Microsoft® Windows® development. And in PROGRAMMING WINDOWS, Fifth Edition, the esteemed Windows Pioneer Award winner revises his classic text with authoritative coverage of the latest versions of the Windows operating system—once again drilling down to the essential API heart of Win32® programming. Packed as always with definitive examples, this newest Petzold delivers the ultimate sourcebook and tutorial for Windows programmers at all levels working with Windows 95, Windows 98, or Windows NT.® No aspiring or experienced developer can afford to be without it.

U.S.A. **$59.99**
U.K. £56.49 [V.A.T. included]
Canada $86.99
ISBN 1-57231-995-X

Microsoft Press® products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at mspress.microsoft.com. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the U.S. (in Canada, call 1-800-268-2222).

Prices and availability dates are subject to change.

Microsoft®
mspress.microsoft.com

Microsoft® Windows® CE Programmer's Guide

Your official guide to the Windows CE operating system—direct from Microsoft.

Design applications for the newest PC companions—or invent the future—with this authoritative guide to Windows CE, version 2.11. This from-the-source reference details the streamlined system architecture and programming interfaces that enable you to build sleek, high-performance Win32® applications for portable computing devices. This newly updated, second-edition guide also features expanded tutorial sections that come loaded with examples and sample code to help accelerate your productivity.

Get the definitive guide to programming the Windows CE API.

Programming Microsoft Windows CE
ISBN: 1-57231-856-2

Part No. 097-0002194

mspress.microsoft.com

Microsoft® Press