Powered by Microsoft® Windows® CE

MICROSOFT PROFESSIONAL EDITIONS

Microsoft® Press

The ultimate reference and toolkit for Windows CE

Microsoft® Windows® CE User Interface Services Guide

Microsoft®

Windows® CE User Interface Services 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. 097-0002195

Contents

Preface

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

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

  Introduces the architecture of the Windows CE operating system.

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

  Provides information on connecting a Windows CE–based device to a desktop computer, synchronizing data between a device and a desktop computer, 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 content described in the following table.

Content	Description
Windows CE 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 Classes (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 User Interface Services Guide*, contains the following chapters:

Introduction to User Interface Services on Windows CE

This chapter provides an overview of the Graphics, Windowing, and Events Subsystem (GWES) module and, specifically, Windows CE User Interface Services. It explains the specific components included in GWES and how those components work together to provide window management and event handling as well as power management features. GWES also supports controls, menus, dialog boxes, resources, user input, and the Graphics Device Interface (GDI).

Working with Windows and Messages

This chapter discusses Windows CE windowing and messaging features. Emphasis is placed on creating, sizing, positioning, and destroying windows as well as sending and responding to messages.

Using Resources

This chapter provides information about resources such as dialog and message boxes, menus, icons, carets, bitmaps, and timers.

Creating Controls

This chapter explains how to add standard controls to a UI.

Working with Graphics

This chapter provides an overview of the Windows CE GDI. It also explains how to work with graphics objects such as pens, brushes, shapes, fonts, and lines.

Working with Sound

This chapter explains how to use the Wave API to add sound to an application.

Receiving User Input

This chapter discusses various user input methods supported by Windows CE, such as keyboard, mouse, handwriting, and Speech API (SAPI). It also describes how to use virtual keys.

Designing a User Interface for Windows CE

This chapter provides a general overview of UI design concepts and presents specific suggestions for working with windows and dialog boxes, adding controls, creating icons, using color, and layout.

Window and Control Styles

This appendix lists window and control styles that are supported by Windows CE.

About the Code Samples Included in this Guide

The *Windows CE User Interface Services Guide* includes code samples developed with Microsoft® Visual C++® version 6.0 and the Microsoft® Windows® CE Toolkit for Visual C++® version 6.0. Concepts presented in sample applications are ported for an H/PC and for an H/PC running Microsoft® Windows® CE, Handheld PC Professional Edition software; however, the samples apply to all Windows CE–based platforms. The following table describes the code samples included in this guide.

Sample name	Description
CeGDI	Demonstrates how to create and use GDI objects such as pens, brushes, palettes, bitmaps, and regions. Shows how to enumerate fonts, select a font type, and display text in a selected font.
CePad	Shows how to register a window class and how to create a window, menu, dialog box, and edit control. Shows how to open and save a file and search for and replace a text string in an edit control.
CmdBand	Registers a toolbar and a rebar control class as well as creates a command band containing two toolbars.
Cmdbar	Shows how to add common buttons to a command bar and how to create, add bitmaps to, and destroy a command bar. Shows how to use the **InsertMenu** function to create a menu. A user can then use this menu to display up to three sets of buttons and two standard bitmaps on the command bar.
FileView	Shows how to register a list view and header control class, create a list view control, and change the window style of the list view control from list view to file view.
ListView	Shows how to register a list view and header control class, create a list view control, and change the window style of the list view control from file view to list view.
Rebar	Shows how to register a rebar and toolbar control class, create a rebar that contains a toolbar and a combo box, and move the rebar up and down.
Toolbar	Shows how to register a toolbar control class, create a toolbar, and add toolbar tips.

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 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 that are 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 User Interface Services on Windows CE

The Microsoft® Windows® CE operating system combines the Microsoft® Win32® application programming interface (API) user interface (UI) and Graphics Device Interface (GDI) libraries into the *Graphics, Windowing, and Events Subsystem* (GWES) module (Gwes.exe). GWES is the interface between the user, your application, and the operating system.

GWES supports all the windows, dialog boxes, controls, menus, and resources that make up the Windows CE UI. This UI enables users to control applications. GWES also provides information to the user in the form of bitmaps, carets, cursors, text, and icons.

Even Windows CE–based platforms that lack a graphical UI use GWES basic windowing and messaging capabilities and power management functions.

GWES Component Model

Because Windows CE is a modular operating system (OS), an OEM can design a platform-specific OS by selecting from a set of available software modules. An OEM can further customize an OS by selecting from a subset of available components for a particular module. Windows CE supplies several pre-tested component configurations that can be divided into three general categories: minimally featured configurations, moderately featured configurations, and full-featured configurations.

Minimally featured configurations build a basic version of Windows CE that includes the core OS or kernel (Core.dll) and selected GWES support, such as messaging, user input, and power management. Minimally featured configurations do not display a UI or contain window management features.

Moderately featured Windows CE configurations include the core OS as well as support for the following GWES features:

- Messaging and user input
- Power management
- Notification light-emitting diode (LED)
- GDI, including Microsoft® TrueType® font pack and raster fonts, text drawing, palette, and printing
- Customizable touch and calibration UI
- Network UI dialogs
- Wave API manager
- Input method manager (IMM)
- Window and dialog management
- Customizable UI

Some minimally featured configurations also provide console, notification, and common control support.

Full-featured configurations build a full version of Windows CE, including all GWES component support. A complete listing of the components contained in full-featured configurations can be found in the *Windows CE Library*.

Window Management and Event Handling

The central feature of any UI is the *window*. In Windows CE–based platforms with a graphical display, the window is the rectangular area of the screen where an application displays output and receives user input. However, even applications running on devices that lack a graphical display require windows to receive messages from the OS. To help manage the windows used in an application, Windows CE supports the Win32 API UI, tailored for the smaller display size of typical Windows CE–based devices. The following table shows the similarities and differences between the Win32 API UI and Windows CE.

Win32 API UI	Windows CE
Owned windows	Full dialog box support
Sends nonclient messages to applications	Not supported
Client area	Maximum area to support a command bar in client area and controlled by application
Multiple-document interface (MDI)	Not supported
Cascading menus and bitmap menus	Not supported
Owner-drawn menus	Not supported
Icons of any size	Supported

The OS controls how the nonclient area is drawn and managed; Windows CE does not send applications messages dealing with the nonclient area of the window.

Event Handling

Windows CE is an event-driven OS. Each message is passed in the **MSG** structure. The Windows CE **MSG** structure contains six members. Message hooks are not supported. The following table shows supported members.

Member	Description
hwnd	Handle to the window whose window procedure receives the message.
message	Specifies the message number.
wParam	Specifies additional message data. The exact meaning depends on the value of the message member.
lParam	Specifies additional message data. The exact meaning depends on the value of the message member.
time	Specifies the time that the message was posted.
pt	Specifies the coordinates of the last position touched on the screen, when the message was posted.

GDI Support

The GDI is the GWES subsystem that controls the display of text and graphics. Use GDI to draw lines, curves, closed figures, text, and bitmap images.

GDI uses a device context to store data that it requires to display text and graphics on a specified device. The graphics 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 clipping region. Windows CE supports printer device contexts for drawing on printers, display device contexts for drawing on video displays, and memory device contexts for drawing into memory.

The following table shows GDI features supported by Windows CE.

GDI feature	Description
Raster and TrueType fonts	TrueType fonts are scalable and rotatable. Seven rasterized system fonts are available in several sizes in ROM. You can also add your own raster fonts. Windows CE supports only one category of font, either raster or TrueType, on a specified system.
Custom color palettes and both palletized and nonpalletized color display devices	Supports color bit depths of 1, 2, 4, 8, 16, 24, and 32 bits per pixel (bpp). The 2-bpp color bit depth is unique to Windows CE.
Bit block transfer functions and raster operation codes	Enables you to transform and combine bitmaps.
Pens and brushes	Supports dashed, wide, and solid pens, and patterned brushes.
Printing	Supports graphics printing.
Cursors	Supports full use of cursors, including user-defined cursors, or just the wait cursor.
Shape drawing functions	Supports the ellipse, polygon, rectangle, and round rectangle shapes.

Windows CE GDI does not support the following features:

- Transformation functions of coordinate space, such as **SetMapMode**, **GetMapMode**, **SetViewportExt**, and **SetWindowExt**. Coordinate space is equivalent to device space.
- World Transform API.
- **MoveTo** and **LineTo** functions.
- Color cursors.
- Animated cursors.

User Input Support

You can configure Windows CE to meet the user input requirements of different platforms. Currently, the keyboard, input panel, voice, mouse, and stylus are the supported input methods (IMs) on Windows CE–based devices.

Keyboard features in Windows CE–based devices are similar to Windows-based desktop platforms. Like those platforms, Windows CE supports *hot keys.* A hot key is a keystroke or combination of keystrokes that shifts the user to a different application. Hot keys give the user high-priority system access for a specific purpose, such as canceling a time-consuming file transfer operation.

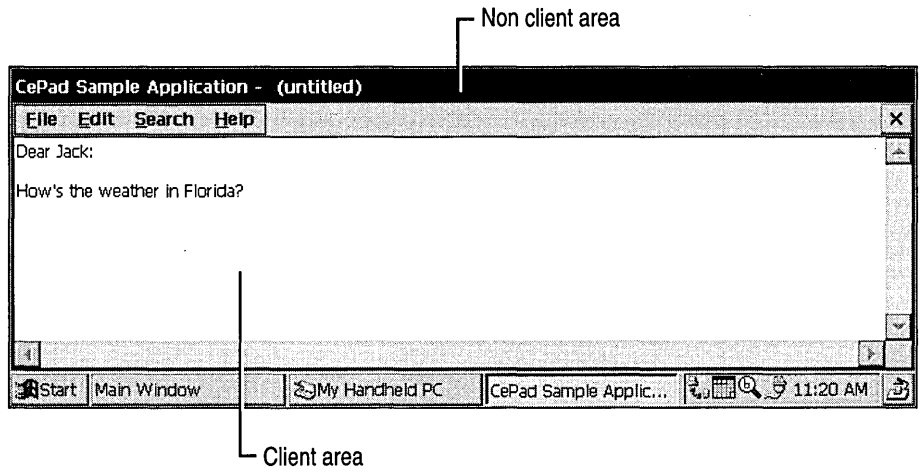
CHAPTER 2

Working with Windows and Messages

The appearance and behavior of a window is largely determined by its inherent attributes and relationship to other windows. You assign attributes to a window by setting window styles and extended styles and by calling functions that alter window attributes.

Windows are always rectangular. They are placed above and below each other along an imaginary line that runs perpendicular to the screen. This stack of windows is called the *z-order*. Each window has a unique position in the z-order. Windows that appear first in the z-order are considered to be in front of, or on top of, windows that appear later in the z-order. A window's position in the z-order affects its appearance. A window might partially or totally obscure another, depending on its location, size, and position in the z-order.

A window is divided into a *nonclient area*—occupied by borders, scroll bars, and various other controls—and a *client area*, the central space inside the nonclient area. You can draw in the client area, but not in the nonclient area. In the Microsoft Windows CE operating system (OS), the nonclient area of a window is controlled exclusively by the window manager. Windows CE does not send applications messages dealing with the nonclient area. The following screen shot shows the nonclient and client areas.



A window can be displayed or hidden, depending on whether its WS_VISIBLE style is turned on or off. A window that has the WS_VISIBLE style turned off will not be displayed on the screen. A window that has the WS_VISIBLE style turned on might or might not be displayed on the screen, depending on whether it is obscured by other windows. Covering or uncovering a window with another window does not change the WS_VISIBLE style.

Every window has a unique identifier called a *window handle*. When you create a window, you receive a window handle, which you can then use to call functions that use the window. Handles are useful in applications that create multiple child windows. You can change the window handle by calling the **SetWindowLong** function and retrieve the handle by calling the **GetWindowLong** function.

All applications have at least one window, regardless of whether they have a graphical user interface (UI). This is because Windows CE is a message-driven OS and a window is the means by which an application receives messages. Each window must be associated with a special function called a *window procedure* or **WinProc**. Windows CE calls this window procedure to pass a message to the application.

A message consists of a *message identifier* and optional parameters. A message identifier is a named constant that identifies a message. When a window procedure receives a message, it uses a message identifier to determine how to process the message. For example:

- The WM_CREATE message is sent to a window when it is created.
- The WM_DESTROY message is sent to a window when it is destroyed.
- The WM_PAINT message is sent to a window when the window client area has changed and must be repainted.

Message parameters contain data, or the location of data, that a window procedure uses to process messages. The meaning and value of the message parameters depend on the message identifier. A message parameter can contain an integer, packed bit flags, a pointer to a structure containing additional data, or other information. A window must check the message identifier to determine how to interpret the message parameters. The term "message" is used to mean either the message identifier or the identifier and the parameters together. The specific meaning is usually clear from the context.

The system sends a message to a window procedure by passing the message data as arguments to the procedure. The window procedure then performs an appropriate action for the message; it checks the message identifier and, while processing the message, uses data specified by the message parameters.

If a window procedure does not process a message, it should pass the message along for default processing. The window procedure does this by calling the **DefWindowProc** function, which performs a default action and returns a message result. The window procedure must then return this value as its own message result. Most window procedures process just a few messages and pass the others on to **DefWindowProc**.

Window procedures can be shared. The handle of the specific window receiving the message is available as an argument of the window procedure.

In addition to having a window procedure, every Windows CE–based application must have the **WinMain** function as its entry point function. **WinMain** performs a number of tasks, including registering the window class for the main window and creating the main window. **WinMain** registers the main window class by calling the **RegisterClass** function, and it creates the main window by calling the **CreateWindowEx** function. **WinMain** does not need to do these things itself; it can call other functions to perform any or all of these tasks. For a code example of a **WinMain** function, see "Creating a Sample Application" later in this chapter.

One task **WinMain** must perform itself is to establish a *message loop*. The message loop retrieves messages from a thread message queue and dispatches them to the appropriate window procedure. The message queue coordinates the transmission of messages for a specified thread. Each thread can have only one message queue. When a message is passed to a window, it is placed on the message queue of the window thread. The thread receives and dispatches the message. There are two ways to pass a message to a window: posting a message and sending a message.

Posting Messages

To post a message, call the **PostMessage** function. **PostMessage** combines the message identifier and parameters into a single message and places it on the receiving window message queue. Eventually, the receiving window message loop removes the message from the message queue and dispatches it to the appropriate window procedure.

PostMessage is an asynchronous function. Windows CE does not synchronize between the sending thread and the receiving thread for posted messages. When the call to **PostMessage** returns, there is no guarantee that the window procedure for the receiving window has processed the message. In fact, if the message was posted to the same thread, the window procedure has not processed the message.

You can post a message without specifying a window. If you supply a NULL window handle when you call the **PostMessage** function, the message is posted to the queue associated with the current thread; because no window handle is specified, you must process the message directly from the message loop. This creates a message that applies to the entire application instead of a specific window.

Sending Messages

To send messages to a window, call the **SendMessage** function. Unlike **PostMessage**, **SendMessage** is a synchronous function. It does not return until the window procedure of the receiver window has processed the message.

You typically send a message when you want a window procedure to perform a task immediately. The **SendMessage** function sends the message directly to the window procedure of the receiving window. The **SendMessage** function waits until the window procedure completes processing and then returns the message result. Parent and child windows often communicate by sending messages to each other. For example, a parent window that has an edit control as its child window can set the text of the control by sending a message to it. The control can notify the parent window of user-initiated changes to the text by sending messages back to the parent.

If the receiving thread is the same as the sending thread, **SendMessage** calls the window procedure directly. If the receiving thread is a different thread from the sending thread, the two message queues synchronize the message passing. The sending thread does not continue executing until the receiving thread processes the message. The receiving thread does not process the message if it is not executing a message loop. Consequently, if you send a message to a window in a thread that is not executing a message loop, the sending thread stops responding.

Receiving and Dispatching Messages

To receive messages, call the **GetMessage** function. When a thread calls **GetMessage**, Windows CE examines the thread message queue for incoming messages. Windows CE processes messages in the following order:

1. Windows CE checks for messages placed on the queue by the **SendMessage** function. After the system removes the message from the queue, it dispatches the message to the appropriate window procedure from within the **GetMessage** function. This guarantees that the sender and receiver message queues remain synchronized. The receiver must call **GetMessage** for the sent messages to be processed.
2. If no sent message is found, Windows CE checks for messages placed on the queue by a call to **PostMessage**.
3. If no posted message is found, Windows CE checks the queue for messages posted by the user input system.

   By processing user input messages at a lower priority, the system guarantees that each input message and any posted messages that result from it are processed completely before moving on to the next input message.
4. If no posted input messages are found, Windows CE checks the queue for WM_QUIT messages placed on the queue by a call to the **PostQuitMessage** function.
5. If no posted quit messages are found, Windows CE checks the queue for WM_PAINT messages placed on the queue by the windowing system.
6. If no paint messages are found, Windows CE checks the queue for WM_TIMER messages placed on the queue by the timer system.

When **GetMessage** receives any of the previous messages, it returns the message content. The thread must call the **DispatchMessage** function to dispatch the message to the correct window procedure. If the message is a WM_QUIT message, the return value of **GetMessage** is zero, which causes the thread to end its message loop.

The system dispatches messages in the **GetMessage** call of the message loop, and the application dispatches messages by calling the **DispatchMessage** function in the message loop.

You might need to process messages you receive from **GetMessage** before you send them out using **DispatchMessage**. The most common processing routines are the **TranslateMessage**, **TranslateAccelerator**, and **IsDialogMessage** functions. Some of these routines can dispatch messages internally because the application no longer needs to call **DispatchMessage** in the main message loop.

You usually call **TranslateMessage** before **DispatchMessage**. **TranslateMessage** determines which characters go with keyboard messages. **TranslateMessage** posts the characters to the message queue to be picked up on the next pass of the message loop.

To intercept keyboard messages and generate menu commands, call the **TranslateAccelerator** function. Call the **IsDialogMessage** function to ensure the proper operation of modeless dialog boxes.

You can remove a message from its queue with the **GetMessage** function. Call the **PeekMessage** function to examine a message without removing it from its queue. This function fills an **MSG** structure with information about the message. Use the **PeekMessage** function carefully because it does not block the waiting for a message event, which enables an application to continue processing regardless of messages in the queue. In a Windows CE–based application, if an application does not block the waiting for a message or some other event, the kernel cannot shift the CPU into low-power mode; this can quickly drain the device batteries.

When processing messages, Windows CE supports both system-defined messages and application-defined messages. System-defined messages have message identifiers ranging from 0 through 0x3ff. Messages with message identifiers ranging from 0x400 through 0x7fff are available for application-defined messages.

There are two types of system-defined messages: general window messages, which are used for all windows, and special purpose messages, which apply to a particular class of windows. General window messages cover a wide range of information and requests, including messages for input device and keyboard input, as well as window creation and management.

The prefix of the symbolic constant for the message generally identifies the category to which the message belongs. For example, general window messages all start with WM, whereas messages that apply only to button controls start with BM.

The following table shows Windows CE message types.

Message type	Description
BM	Button message
BN	Button notification
CB	Combo box message
CBN	Combo box notification
CDM	Common dialog box message
CDN	Common dialog box notification
CPL	Control panel message
DB	Object store message

Message type	Description
DM	Dialog box default push button message
DTM	Date time picker and Hypertext Markup Language (HTML) viewer messages
DTN	Date time picker notification
EM	Edit control message
EN	Edit control notification
HDM	Header control message
HDN	Header control notification
IMN	Input context message
LB	List box control message
LBN	List box notification
LINE	Line device message
LVM	List view message
LVN	List view notification
MCM	Month calendar message
MCN	Month calendar notification
NM	Messages sent by a variety of controls
PBM	Progress bar message
PSM	Property sheet message
PSN	Property sheet notification
RB	Rebar message
RBN	Rebar notification
SB	Status bar window message
SBM	Scroll bar message
STM	Static bar message
STN	Static bar notification
TB	Toolbar message
TBM	Trackbar message
TBN	Trackbar notification
TCM	Tab control message
TCN	Tab control notification
TVM	Tree view message
TVN	Tree view notification
UDM	Up-down control message
UDN	Up-down control notification
WM	General window messages

You can define messages for use by your own application's window. If you create messages, be sure that the window procedure that receives them interprets and processes them correctly. The operating system (OS) does not interpret application-defined messages.

In some situations, you need to use messages to communicate with windows that are controlled by other processes. In this situation, call the **RegisterWindowMessage** function to register a message identifier. The message number returned is guaranteed to be unique throughout the system. By using this function, you prevent the conflicts that can arise if different applications use the same message identifier for different purposes.

Windows CE does not support hooking messages because the extra processing required by hooks can impair the performance of Windows CE–based devices.

When handling messages in your application, be aware of the WM_HIBERNATE message. Windows CE defines a WM_HIBERNATE message to notify an application when system resources run low. When an application receives this message, it should attempt to release as many resources as possible. The system checks memory status at five-second intervals. Every Windows CE–based application that uses even moderate amounts of system resources should implement a handler for the WM_HIBERNATE message. If an application window is not visible, it cannot receive a WM_HIBERNATE message. This is because the WM_HIBERNATE message is sent only to applications that have a button on the taskbar, which only visible windows do. A hidden window will not get this message, even if it is a top-level, overlapped window.

Creating a Window

Every window is a member of a *window class*. A window class is a template for creating a window. When you write an application, you must register all window classes that are used to create windows. To simplify the process of creating windows, Windows CE includes several system-defined window classes; because Windows CE registers these classes automatically, you can immediately create windows with them.

You create windows with the **CreateWindow** or **CreateWindowEx** function. The only difference between these functions is that **CreateWindowEx** supports the extended style parameter, *dwExStyle*, while **CreateWindow** does not. These functions take a number of parameters that specify the attributes of the window being created. In Windows CE, **CreateWindow** is implemented as a macro that calls **CreateWindowEx**.

Windows CE includes additional functions, including **DialogBox**, **CreateDialog**, and **MessageBox**, for creating special-purpose windows such as dialog boxes and message boxes.

The **CreateWindowEx** function has the following syntax:

```
HWND
CreateWindowEx(
   DWORD       dwExStyle,       //Extended style parameter
   LPCWSTR     lpClassName,     //Class name parameter
   LPCWSTR     lpWindowName,    //Window name parameter
   DWORD       dwStyle,         //Style parameter
   int         X,               //Horizontal parameter
   int         Y,               //Vertical parameter
   int         nWidth,          //Width parameter
   int         nHeight,         //Height parameter
   HWND        hwndParent,      //Parent parameter
   HMENU       hMenu,           //Menu parameter
   HINSTANCE   hInstance,       //Instance handle parameter
   LPVOID      lpParam);        //Creation data parameter
```

The following table shows the window attributes in **CreateWindowEx**.

Window attribute	Description
Extended style	The dw*ExStyle* parameter specifies one or more window extended styles. These have their own set of WS_EX_* flags and should not be confused with the WS_* flags.
Class name	Every window belongs to a window class. Except for built-in classes, such as controls, an application must register a window class before creating any windows of that class. The *lpClassName* parameter specifies the name of the class used as a template for creating the window.
Window name	The window name, also called window text, is a text string associated with a window. The *lpWindowName* parameter specifies the window text for the newly created window. Windows use this text in different ways. A main window, dialog box, or message box typically displays its window text in its title bar. A button control, edit control, or static control displays its window text within the rectangle occupied by the control. A list box, combo box, or scroll bar control does not display its window name. All windows have the text attribute, even if they do not display the text.
Style	The *dwStyle* parameter specifies one or more window styles. A window style is a named constant that defines an aspect of the window's appearance and behavior. For example, a window with the WS_BORDER style has a border around it. Some window styles apply to all windows; others apply only to windows of specific window classes. For a list of all supported window styles, see Appendix A, "Window and Control Styles."

Window attribute	Description
Horizontal and vertical coordinates	The *x* and *y* parameters specify the horizontal and vertical screen coordinates, respectively, of the window's upper-left corner.
Width and height coordinates	The *nWidth* and *nHeight* parameters determine the width and height of the window in device units.
Parent	The *hwndParent* parameter specifies the parent or the owner of a window, depending on the style of the flags passed in. If neither the WS_POPUP nor WS_CHILD style is specified, the *hwndParent* parameter might be a valid window handle or NULL. If the parameter is NULL, the new window is a top-level window without a parent or owner. If it is non-NULL, the new window is created as a child of the specified parent window. If WS_CHILD is specified, the *hwndParent* parameter must be a valid window handle. The new window is created as a child of the parent window. If the WS_POPUP style is specified, the new window is created as a top-level window and the *hwndParent* parameter specifies the owner window. If WS_POPUP is specified and the parameter is NULL, the new window is partially owned by Windows CE. The WS_POPUP style overrides the WS_CHILD style.
Menu	Windows CE does not support menu bars. In Windows CE, you can use the *hMenu* parameter to identify a child window. Otherwise, it must be NULL.
Instance handle	The *hInstance* parameter identifies the handle of the specific instance of the application that creates the window.
Creation data	Every window receives a WM_CREATE message when it is created. The *lpParam* parameter is passed as one of the message parameters. Although it can be any value, it is most commonly a pointer to a structure containing data that is required to create a particular window.

The class name for a new window class has to be a Unicode string. You can use the **TEXT** macro to cast a string as Unicode, as in **TEXT**("classname").

The system does not automatically display the main window after creating it. Rather, the application's **WinMain** function uses the **ShowWindow** function to display the window. An application uses the **SetWindowText** function to change the window text after it creates the window. It uses the **GetWindowTextLength** and **GetWindowText** functions to retrieve the window text from a window.

For an example of how to call the **CreateWindowEx** function, see "Creating a Sample Application" later in this chapter.

Window Relationship Fundamentals

When you create a window, you need to specify a window style. The style you select determines the relationship that window has with other windows in your application. For example, you can designate a window as a child of another window by specifying the WS_CHILD style. A *child window* is a window that appears only within the client area of its parent window. A child window has only one parent, but a parent window can have any number of child windows and these, in turn, can have their own child windows.

A child window that can trace a relationship to a parent window through a chain of parent/child window relationships, however long, is said to be a *descendant* of the parent window. Likewise, a parent window that can trace a relationship to a child window through a chain of parent/child windows is said to be an *ancestor* window of that child window. To determine whether a window is a descendant window of a specified parent window, call the **IsChild** function.

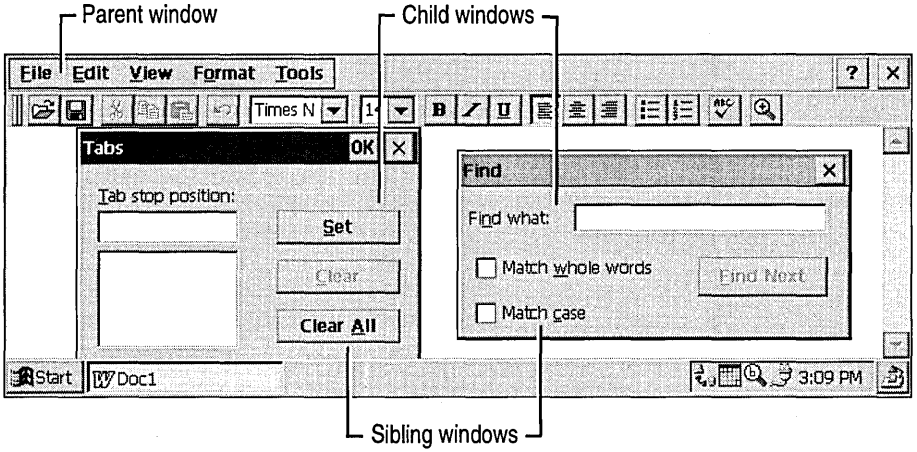
You can change the parent window of an existing child window by calling the **SetParent** function. When you do, the system removes the child window from the client area of the existing parent window and moves it to the client area of the new parent window. The **GetParent** function retrieves the handle to a window's parent window.

Windows CE has rules governing the display and behavior of parent and child windows. For example, a child window is positioned relative to the upper-left corner of its parent's client rectangle. Child windows are always placed directly in front of their parent windows and are always kept with their parents in the z-order. When the z-order of a parent window is changed, child windows automatically move with their parent.

Although you can place or size a child window outside of a parent window, a child window cannot draw any part of itself outside of its parent's client rectangle. In Windows CE, a parent window cannot draw on its children, and a window cannot draw on siblings in front of it. In other words, all windows behave as if they have the WS_CLIPCHILDREN and WS_CLIPSIBLING styles. You can avoid some of these restrictions by using the **GetDCEx** function. This function enables you to obtain a handle to a device context for the client area of a specified window and to control how, or whether, clipping occurs. For example, when used with either the WS_CLIPSIBLINGS or DCX_CACHE style, **GetDCEx** enables a child window to scroll with its parent.

A window that has no parent is called a *top-level window*. Windows that have the same parent are *sibling windows*. Even though they might be in different applications, all top-level windows are considered siblings. Top-level windows are parented to an invisible dummy root window.

The following screen shot shows the parent/child window relationship.



A window can also be defined as another window's owner; thus, there can be an *owner window* and an *owned window*. Although the relationship between an owner window and an owned window is similar to the relationship between a parent and child window, there are some differences. For example, the owner-owned relationship can exist only between top-level windows and, unlike child windows, owned windows can draw outside of their owners.

You can create an owner-owned relationship between top-level windows when you create a window with the WS_POPUP style. Because top-level windows do not have parents, the window you specify as the parent when you call the **CreateWindow** function becomes the owner of the new window. Owned windows can in turn own other windows. To return the owner of a specified window, call the **GetParent** function. When a window is destroyed, owned windows are also destroyed.

Owner-owned windows move as a group. If you move a window forward in the z-order, its owner window and owned windows move forward with it. Windows CE places owned windows in front of their owners. Although Windows CE does not prevent you from inserting a top-level window between an owner window and an owned window, it does keep owned groups of windows together when one is moved in the z-order. This means that when you change a window's placement in the z-order, Windows CE displaces any windows between the window and its owned or owner windows. Moving or sizing a window does not affect the location or size of its owner or owned windows.

You can create a WS_POPUP window with a NULL owner. When you do, the window becomes partially owned by the desktop. If Windows CE moves the desktop to the top of the z-order, these windows remain on top of the desktop. However, if you move the window to the top of the z-order, it does not pull the desktop with it. Threads in the system that do not usually have any kind of window interface use this style when they need to display a message to the user. Owned windows typically do not have taskbar buttons. However, if you create a WS_POPUP window with a NULL owner, a taskbar button will appear in the taskbar. Be sure when specifying the WS_POPUP style that you do not specify the WS_CHILD style as well. WS_POPUP and WS_CHILD windows are incompatible and should not be used together.

Displaying a Window

You can control a window's visibility by using the **ShowWindow** or **SetWindowPos** function or by turning its WS_VISIBLE style on or off. Think of the WS_VISIBLE style as a way to hide a window. If this style is turned off, neither the window nor its descendants will be drawn on the screen. Even though a child window is hidden when its parent is hidden, the child window's WS_VISIBLE style is not changed when its parent's style is changed. A child window might have the WS_VISIBLE style turned on and still not be visible if it has a parent or ancestor window with the WS_VISIBLE style turned off.

To determine if a window is visible, call the **IsWindowVisible** function. This function checks the window and its ancestors to determine if the window is visible. A window might be considered visible but might not appear on the screen if it is covered by other windows.

By default, the **CreateWindowEx** function creates a hidden window unless you specify the WS_VISIBLE style. Typically, an application sets the WS_VISIBLE style after it has created a window to keep details of the creation process hidden from the user. For example, an application might keep a new window hidden while it customizes the window appearance.

Changing the visibility of a window does not automatically change the visibility of windows that it owns. Also, if you create a dialog box whose parent window is not visible, the dialog box will be visible. To avoid this inconsistency, do not create a dialog box that is owned by an invisible window.

Sizing and Positioning a Window

A window's size and position are expressed as a bounding rectangle specified in coordinates relative to the screen or to the parent window. Typically, window dimensions and coordinates are measured in pixels.

When you create a window, you can set the initial size and position of the window directly or instruct the system to calculate the initial size and position by specifying CW_USEDEFAULT in the **CreateWindow** or **CreateWindowEx** function. After creating a window, set the window size or position by calling the **MoveWindow** or **SetWindowPos** function.

If you need to create a window with a client area of a specified size, call the **AdjustWindowRectEx** function to calculate the required size of a window based on the preferred size of the client area. Pass the resulting size values to the **CreateWindowEx** function.

Although you can create a window of any size, it should not exceed the screen size of the target device. Before setting a window size, check the width and height of the screen by using the **GetSystemMetrics** function with the SM_CXSCREEN and SM_CYSCREEN flags.

You can call the **GetWindowRect** function to retrieve the coordinates of a window's bounding rectangle. **GetWindowRect** fills a **RECT** structure with the coordinates of the window's upper-left and lower-right corners. The coordinates are relative to the upper-left corner of the screen, even for a child window. The **ScreenToClient** or **MapWindowPoints** function maps the screen coordinates of a child window's bounding rectangle to coordinates relative to the parent window's client area.

The **GetClientRect** function retrieves the position and size of a window client area; because coordinates are relative to the client area itself, the client area's upper-left corner is always at coordinates (0, 0) and the coordinates of the lower-right corner are the width and height of the client area. Furthermore, because the command bar is part of the client area in Windows CE, it is included in the dimensions that are returned by the **GetClientRect** function.

To retrieve the handle to the window that occupies a particular point on the screen, call the **WindowFromPoint** function. Call the **ChildWindowFromPoint** function to retrieve the handle to the child window that occupies a specified point in the parent window client area. To convert the client coordinates of a specified point to screen coordinates, call the **ClientToScreen** function. To convert the screen coordinates of a specified point into client coordinates, call the **ScreentoClient** function.

To change a window's position in the z-order, call the **SetWindowPos** function. This function is used to create a topmost window. A *topmost window* is a window that has the WS_EX_TOPMOST style. Do not confuse topmost with top-level. Top-level refers to whether or not a window has a parent, whereas topmost refers to a specific style that controls the z-order for the window. Topmost windows are above all non-topmost sibling windows in the z-order. To create a topmost window, specify the WS_EX_TOPMOST style when you create the window, or call **SetWindowPos** and set the *hWndInsertAfter* parameter to HWND_TOPMOST.

A window might lose its topmost style by calling **SetWindowPos** and setting the *hWndInsertAfter* parameter to HWND_NOTOPMOST. If a window is positioned directly after a non-topmost window, that window loses its WS_EX_TOPMOST style. You can set the **SetWindowLong** function to give a window the WS_EX_TOPMOST style; however, this function does not change the window's z-order.

The defer window API, which consists of **DeferWindowPos**, **BeginDeferWindowPos**, and **EndDeferWindowPos**, is an alternative to making repeated calls to **SetWindowPos**. These functions enable you to queue up several window position changes and execute them simultaneously.

Occasionally, topmost windows can disappear behind the desktop. This typically occurs when one of the following rules governing the use of topmost windows is broken:

- When creating an owned window, if the owner of the window is designated WS_EX_TOPMOST, the owned window must be a topmost window as well.
- When calling **SetWindowPos** to change the topmost attribute of a window, you must change the topmost attribute of all of its owned windows.
- When changing the topmost attribute of a window, do not call **SetWindowLong**.
- When changing the z-order of a member of an owned group of windows, you must respect the topmost attribute of other windows.

Destroying a Window

To destroy a window, call the **DestroyWindow** function. When a thread or process terminates, Windows CE removes all windows that are owned by that thread or process. Windows that are removed when a thread or process terminates do not always receive WM_DESTROY messages. For this reason, it is recommended that you manually destroy windows. When a window is destroyed, the system hides the window, sends a WM_DESTROY message to the window procedure of the window being destroyed, and removes any associated internal data. The window handle becomes invalid and can no longer be used by the application.

Destroying a window automatically destroys the window's descendant windows. The **DestroyWindow** function first sends a WM_DESTROY message to the initial window being destroyed and then to its descendant windows.

You should destroy any window that is no longer needed. Before destroying a window, save or remove any data associated with the window and release system resources allocated to the window.

Destroying a window does not affect the window class from which the window is created. You can still create new windows by using the class, and any existing windows of that class continue to operate.

Creating a Sample Application

This section contains a code example used to create a simple Windows CE–based application. This sample application demonstrates the basic framework common to all Windows CE–based applications. It begins executing with the **WinMain** function, which performs the following tasks:

1. **WinMain** places the application-instance handle in a global variable. Because this handle is used in various places throughout an application, it is common to place it in a global variable that is accessible to all functions. The smallest possible interval a timer can measure is the system-tick interval.
2. **WinMain** calls the application-defined **InitApplication** function, which then calls the **RegisterClass** function to register the application's main window class. More complicated applications might need to register more window classes and determine if other instances of the application are running.

3. **WinMain** calls the application-defined **InitInstance** function, which then calls the **CreateWindow** function to create a window. **CreateWindow** returns a window handle identifying the new window. This handle is used to refer to the window in subsequent function calls.
4. **WinMain** creates the message loop by calling the **GetMessage**, **TranslateMessage**, and **DispatchMessage** functions in the format displayed in the sample application. This loop receives messages and dispatches them to the window procedures.

Note that the application does not directly call the window procedure, **MainWndProc**. The system calls this function as the message loop receives and dispatches messages. In this application, **MainWndProc** processes only the WM_CLOSE message that tells the window to close. When the window receives a WM_CLOSE message, it calls the **PostQuitMessage** function, which then calls **GetMessage** to return FALSE. This, in turn, causes the message loop to terminate and the application to exit.

Windows CE sends many other messages to the window besides WM_CLOSE. **MainWndProc** passes all other messages to the **DefWindowProc** function, the default window procedure provided by the system. Pass all messages to **DefWindowProc** that you do not process yourself; otherwise, your window might not function correctly.

The following code example shows a framework for creating a Windows CE–based application.

```
#include <windows.h>

HINSTANCE g_hInst = NULL;      // Handle to the application instance
HWND g_hwndMain = NULL;        // Handle to the application main window
TCHAR g_szTitle[80] = TEXT ("Main Window"),
                               // Application main window name
      g_szClassName[80] = TEXT ("Main window class");
                               // Main window class name

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

FUNCTION:
  WndProc

PURPOSE:
  The callback function for the main window. It processes messages sent
  to the main window.

*************************************************************************/
LRESULT CALLBACK WndProc (HWND hwnd, UINT umsg, WPARAM wParam,
                          LPARAM lParam)
{
```

```
  switch (umsg)
  {
    // Add cases such as WM_CREATE, WM_COMMAND, WM_PAINT if you don't
    // want to pass these messages along for default processing.

    case WM_CLOSE:
      DestroyWindow (hwnd);
      return 0;

    case WM_DESTROY:
      PostQuitMessage (0);
      return 0;
  }
return DefWindowProc (hwnd, umsg, wParam, lParam);
}

/**********************************************************************
FUNCTION:
  InitInstance

PURPOSE:
  Create and display the main window.

**********************************************************************/
BOOL InitInstance (HINSTANCE hInstance, int iCmdShow)
{

  g_hInst = hInstance;

  g_hwndMain = CreateWindow (
                  g_szClassName,  // Registered class name
                  g_szTitle,      // Application window name
                  WS_OVERLAPPED,  // Window style
                  0,              // Horizontal position of the window
                  0,              // Vertical position of the window
                  CW_USEDEFAULT,  // Window width
                  CW_USEDEFAULT,  // Window height
                  NULL,           // Handle to the parent window
                  NULL,           // Handle to the menu identifier
                  hInstance,      // Handle to the application instance
                  NULL);          // Pointer to the window-creation data

  // If it failed to create the window, return FALSE.
  if (!g_hwndMain)
    return FALSE;

  ShowWindow (g_hwndMain, iCmdShow);
  UpdateWindow (g_hwndMain);
  return TRUE;
}
```

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

FUNCTION:
  InitApplication

PURPOSE:
  Declare the window class structure, assign values to the window class
  structure members, and register the window class.

*********************************************************************/
BOOL InitApplication (HINSTANCE hInstance)
{
  WNDCLASS wndclass;
wndclass.style = CS_HREDRAW | CS_VREDRAW;
  wndclass.lpfnWndProc = (WNDPROC)WndProc;
  wndclass.cbClsExtra = 0;
  wndclass.cbWndExtra = 0;
  wndclass.hIcon = NULL;
  wndclass.hInstance = hInstance;
  wndclass.hCursor = NULL;
  wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
  wndclass.lpszMenuName = NULL;
  wndclass.lpszClassName = g_szClassName;
return RegisterClass (&wndclass);
}

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

FUNCTION:
  WinMain

PURPOSE:
  The WinMain function of the application. It is called by the system as
  the initial entry point for this Windows CE-based application.

*********************************************************************/
int WINAPI WinMain (
              HINSTANCE hInstance,       // Handle to the current
                                         // instance
              HINSTANCE hPrevInstance,   // Handle to the previous
                                         // instance
              LPWSTR lpCmdLine,          // Pointer to the command line
              int iCmdShow)              // Shows the state of the
                                         // window
{
  MSG msg;                               // Message structure
  HACCEL hAccel;                         // Handle of the accelerator
                                         // table

  if (!hPrevInstance)
```

```
{
    if (!InitApplication (hInstance))
      return FALSE;
  }
if (!InitInstance (hInstance, iCmdShow))
    return FALSE;

// Insert code here to load the accelerator table.
// hAccel = LoadAccelerators (...);
while (GetMessage (&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator (
                    g_hwndMain,     // Handle to the destination window
                    hAccel,         // Handle to the accelerator table
                    &msg))          // Address of the message data
    {
      TranslateMessage (&msg);
      DispatchMessage (&msg);
    }
  }

  return msg.wParam;
}
```

CHAPTER 3

Using Resources

Resources are objects that are used within an application, but they are defined outside an application. They are added to the executable file when the application is linked. The Windows CE operating system (OS) resources include menus, keyboard accelerators, dialog boxes, carets, cursors, icons, bitmaps, string-table entries, message-table entries, timers, and user-defined data. The Windows CE–based platform that is targeted determines available resources.

Resource files have an .rc extension. Text-based resources, such as menus, can be created by using a text editor. Resources such as icons are generated by using a resource editor. Resources created by using a resource editor must be referenced in the .rc file associated with your application. Resource files contain a special resource language, or *script*, that must be compiled by a resource compiler. The resource compiler converts the .rc file into a resource (.res) file, and then it links the file to the application.

Regardless of how a resource is compiled, you must load resources into memory before you can use them. The **FindResource** function finds a resource in a module and returns a handle to the binary resource data. The **LoadResource** function uses the resource handle returned by **FindResource** to load the resource into memory. After you load a resource by using **LoadResource**, Windows CE automatically unloads and reloads the resource as memory conditions and application execution require. Thus, you need not explicitly unload a resource that you no longer need.

To find and load any type of resource in one call, use the **FindResource** and **LoadResource** functions; use these functions only if you must access the binary resource data for subsequent function calls.

The following table shows the resource-specific functions you can use.

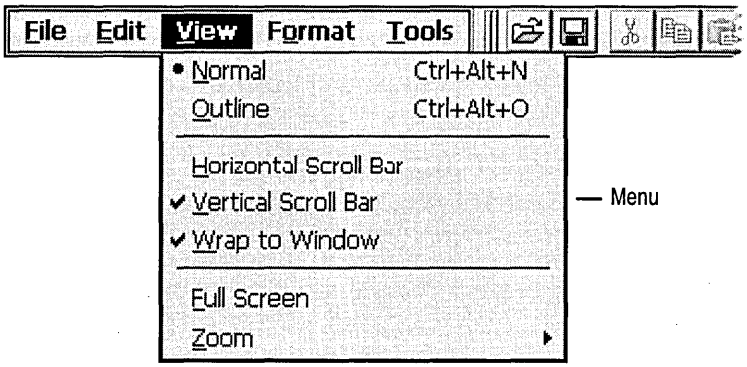
Function	Description
FormatMessage	Loads and formats a message-table entry
LoadAccelerators	Loads an accelerator table
LoadBitmap	Loads a bitmap resource
LoadCursor	Loads a cursor resource
LoadIcon	Loads an icon resource
LoadImage	Loads an icon, cursor, or bitmap
LoadMenu	Loads a menu resource
LoadString	Loads a string-table entry

Before terminating, an application should release the memory that is occupied by accelerator tables, bitmaps, cursors, icons, and menus. The following table shows the functions an application can use to do this.

Resource	Release function
Accelerator table	**DestroyAcceleratorTable**
Bitmap	**DeleteObject**
Cursor	**DestroyCursor**
Icon	**DestroyIcon**
Menu	**DestroyMenu**

Creating Menus

A *menu* is a list of options called *menu items* from which a user can choose to perform an action. Choosing a menu item opens a submenu or causes the application to execute a command. Virtually all main windows contain some type of menu. In many Windows CE–based applications, these menus are placed in a *command bar*, but you can also place a menu directly on the window itself. A command bar combines the features of a menu bar and a toolbar in one control. For more information about creating a command bar, see Chapter 4, "Creating Controls."



In addition to standard menus and command bar menus, some Windows CE–based applications include scrolling menus. If a menu does not fit on the screen, Windows CE adds scroll arrows so that users can scroll up and down through the menu. When a user cannot scroll any further in one direction or the other, the associated scroll arrow is dimmed. An application dims unavailable items to provide a visual indicator that a selection is unavailable. Pressing the up or down scroll arrow scrolls through the menu one item at a time. No menu item is highlighted while scrolling. Changing the selection by using a keyboard arrow key or keyboard mnemonic causes the newly selected item to scroll into view, if it is not already displayed. If a menu has too many columns to fit the width of the display area, Windows CE ignores all column breaks and converts the menu into a single-column scrolling menu. If an individual menu item is too large to be drawn without being clipped by the up or down scroll arrow, the item is not drawn. This might leave a large blank space next to a scroll arrow.

Regardless of whether a menu is standard or scrolling, all menus in Windows CE are implemented as a top-level, *pop-up window*. A pop-up window menu is a floating menu that displays commands specific to the object selected by the user or to the object's immediate context. Each menu must have an owner window. Windows CE sends a WM_COMMAND message to a menu's owner window when the user selects the menu or chooses an item from the menu. When a user selects a menu item that opens a submenu, Windows CE does not send a command message to the menu's owner window. Rather, Windows CE sends a WM_INITMENUPOPUP message before displaying the submenu. Obtain the handle to the submenu associated with an item by using the **GetSubMenu** or **GetMenuItemInfo** function.

There are two ways to create a menu: define a menu template in your resource file or use menu creation functions.

Defining a Menu Template

A *menu template* defines a menu, including all associated menu items and submenus, in a resource file. Implementing a menu as a resource makes an application easier to localize because only the resource-definition file needs to be localized for each language, and not the application source code. The following code example shows the syntax for menu resource definitions.

```
menuID MENU [[optional-statements]] { item-definitions . . . }
```

Here, *menuID* is either a unique string or unique 16-bit unsigned integer that identifies the menu, *optional-statements* specify options you can include when creating a menu, and *item-definitions* are used to create menu items.

There are two types of menu items you can create: **MENUITEM** and **POPUP**. A **MENUITEM** statement specifies a final selection; a **POPUP** statement specifies a popup submenu, which also may contain **MENUITEM** and **POPUP** statements. The following code example shows the syntax for these two menu items.

```
MENUITEM text, result, [[optionlist]] MENUITEM SEPARATOR
POPUP text, [[optionlist]] { item-definitions . . . }
```

Here, *text* is a string that contains the name of the menu, *optionlist* is a parameter that specifies the appearance of the menu, such as checked or dimmed, and *result* is the number that is generated when the user chooses the menu item. This parameter accepts an integer value and returns an integer; when the user selects the menu item name, the result is sent to the window that owns the menu.

Windows CE provides a special type of menu item, called a *separator*, that appears as a horizontal line. You can use a separator to divide a menu into groups of related items. The **MENUITEM SEPARATOR** form of the **MENUITEM** statement creates a separator. A separator cannot be used in a command bar, and the user cannot select a separator.

The following code example shows a complete **MENU** statement.

```
#define IDR_CEPADMENU                   101
#define IDM_NEW                         40001
#define IDM_OPEN                        40002
#define IDM_SAVE                        40003
#define IDM_SAVEAS                      40004
#define IDM_EXIT                        40005
#define IDM_ABOUT                       40006
#define IDM_UNDO                        40007
#define IDM_CUT                         40008
#define IDM_COPY                        40009
#define IDM_PASTE                       40010
#define IDM_CLEAR                       40011
#define IDM_SELECTALL                   40012
#define IDM_FIND                        40013
#define IDM_FINDNEXT                    40014
#define IDM_REPLACE                     40015

IDR_CEPADMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New          Ctrl+N",        IDM_NEW
        MENUITEM "&Open...      Ctrl+O",        IDM_OPEN
        MENUITEM "&Save         Ctrl+S",        IDM_SAVE
        MENUITEM "Save &As...",                 IDM_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "&Exit",                       IDM_EXIT
    END

    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo         Ctrl+Z",        IDM_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t          Ctrl+X",        IDM_CUT
        MENUITEM "&Copy         Ctrl+C",        IDM_COPY
        MENUITEM "&Paste        Ctrl+V",        IDM_PASTE
        MENUITEM "Clea&r        Del",           IDM_CLEAR
        MENUITEM SEPARATOR
        MENUITEM "Select A&ll   Ctrl+A",        IDM_SELECTALL
    END
```

```
    POPUP "&Search"
    BEGIN
        MENUITEM "&Find...       Ctrl+F",        IDM_FIND
        MENUITEM "Find &Next F3",                IDM_FINDNEXT
        MENUITEM "R&eplace...    Ctrl+H",        IDM_REPLACE
    END

    POPUP "&Help"
    BEGIN
        MENUITEM "&About CE Pad",                IDM_ABOUT
    END
END
```

Menu-template resources can be loaded explicitly or assigned as the default menu for a window class. To load a menu explicitly, call the **LoadMenu** function. To assign a menu to a window class, assign the name of the menu-template resource to the *lpszMenuName* member of the **WNDCLASS** structure that is used to register the class.

Using Menu Creation Functions

You use menu creation functions when you want to create or change a menu at run time. To create an empty menu bar, use the **CreateMenu** function; to create an empty menu, use the **CreatePopupMenu** function. To add items to a menu, use the **AppendMenu** and **InsertMenu** functions.

When a menu contains more items than will fit in one column, the menu is truncated unless you force a line break. You can cause a column break to occur at a specific item in a menu by assigning the MFT_MENUBREAK menu item type flag to the item. Windows CE places that item and all subsequent items in a new column. You can also assign the MFT_MENUBARBREAK menu item type flag to the item, which has the same effect as the MFT_MENUBREAK menu item type flag except that a vertical line appears between the new and existing columns.

To display a shortcut menu, use the **TrackPopupMenuEx** function. Shortcut menus, also called floating pop-up menus or context menus, are typically displayed when the WM_CONTEXTMENU message is processed. The older **TrackPopupMenu** function is supported, but new applications should use the **TrackPopupMenuEx** function.

If a menu is assigned to a window and that window is destroyed, Windows CE automatically destroys the menu, freeing the menu handle and the memory occupied by the menu. Windows CE does not automatically destroy a menu that is not assigned to a window. An application must destroy the unassigned menu by calling the **DestroyMenu** function.

Setting Menu Item Attributes

Menu items possess attributes that affect their appearance. For example, a menu item can be checked or unchecked, mutually exclusive, or dimmed. You can change other menu attributes as well by creating an owner-drawn menu item. Each menu item in a command bar or menu has a unique position value. The leftmost item in a command bar or the top item in a menu has position zero. The position value is incremented for subsequent menu items. Windows CE assigns a position value to all menu items, including separators. When calling a menu function that modifies or retrieves information about a specific menu item, specify the item by using either its handle or its position.

A check mark attribute controls whether a menu item is checked. Applications check or uncheck a menu item to indicate whether an option is in effect. For example, suppose that an application has a toolbar that a user can hide or display by using a **Toolbar** command on a menu. When the toolbar is hidden, the **Toolbar** menu item is unchecked. When the user chooses the command, the application checks the menu item and shows the toolbar. Windows CE displays a bitmap next to checked menu items to indicate their checked state; it does not display a bitmap next to unchecked items. Only menu items in a menu can be checked. Items appearing in a command bar cannot be checked. To set a menu item check mark attribute, call the **CheckMenuItem** function.

Sometimes, a group of menu items corresponds to a set of mutually exclusive options. In this case, indicate the selected option by using a checked radio menu item—analogous to a radio button control. Checked radio items are displayed with a bullet bitmap instead of a check mark bitmap. To check a menu item and make it a radio item, use the **CheckMenuRadioItem** function.

Another menu item attribute you can change is whether or not a menu item is enabled. When a menu item is not available, the item should be dimmed. Dimmed menu items cannot be chosen. You can use a dimmed item when an action is not appropriate. For example, you can dim the **Print** command on the **File** menu when the system does not have a printer installed. A menu item can be enabled or dimmed by using the **EnableMenuItem** function. To determine whether a menu item is enabled or dimmed, use the **GetMenuItemInfo** function.

You can control a menu item's appearance by using an owner-drawn item. Owner-drawn items require an application to draw selected, checked, and unchecked states. For example, if an application provides a font menu, it can draw each menu item by using the corresponding font; the item for Roman will be drawn in Roman, the item for Italic will be drawn in Italic, and so on.

Windows CE handles owner-drawn menu items differently from Windows-based desktop platforms. In some respects, it treats an owner-drawn item as any other menu item. On other Windows-based platforms, the device context is initialized to its default state. On Windows CE, the device context is initialized to the dimmed or highlighted status of the current item. Also, unlike other Windows-based platforms, Windows CE automatically highlights an owner-drawn menu item when it has the keyboard focus.

Creating Keyboard Accelerators

A *keyboard accelerator*, also known as a shortcut key, is a keystroke or combination of keystrokes that generates a WM_COMMAND message. Keyboard accelerators are often used as shortcuts for commonly used menu commands, but you can also use them to generate commands that have no equivalent menu items. Include keyboard accelerators for any common or frequent actions, and provide support for the common shortcut keys where they apply.

You can use an ASCII character code or a *virtual-key* code to define the accelerator. A virtual key is a device-independent value that identifies the purpose of a keystroke as interpreted by the Windows keyboard device driver. An ASCII character code makes the accelerator case-sensitive. The ASCII "C" character can define the accelerator as ALT+c rather than ALT+C. Because accelerators do not need to be case-sensitive, most applications use virtual-key codes for accelerators rather than ASCII character codes.

If an application defines an accelerator that is also defined in the system accelerator table, the application-defined accelerator overrides the system accelerator, but only within the application context. Avoid this, because it prevents the system accelerator from performing its standard role in the Windows CE user interface (UI).

▶ **To create an accelerator table**

1. Use a resource compiler to define an *accelerator table* resource and add it to your executable file.

   An accelerator table consists of an array of **ACCEL** data structures, each of which defines an individual accelerator.

2. Call the **LoadAccelerators** function at run time to load the accelerator table and to retrieve the handle of the accelerator table.
3. Pass a handle to the accelerator table to the **TranslateAccelerator** function to activate the accelerator table.

You can also create an accelerator table for an application at run time by passing an array of **ACCEL** structures to the **CreateAcceleratorTable** function. This method supports user-defined accelerators in the application. **CreateAcceleratorTable** creates an accelerator table that must be destroyed before an application closes. Call the **DestroyAcceleratorTable** function to destroy the accelerator table.

Defining an Accelerator Table

The following code example shows the syntax for accelerator table definitions.

```
acctablename ACCELERATORS [optional-statements] { event, idvalue, [type]
[options] . . . }
```

When using the **ACCELERATORS** statement in your resource-definition file, assign a unique name or resource identifier to the accelerator table. Windows CE uses the identifier to load the resource at run time.

Each accelerator you define requires a separate entry in the accelerator table. In each entry, you define the keystroke that generates the accelerator and the accelerator identifier. The keystroke is either an ASCII character code or a virtual-key code. Also specify if the keystroke must be used in some combination with the ALT, SHIFT, or CTRL key.

An ASCII keystroke is specified either by enclosing the ASCII character in double quotation marks or by using the integer value of the character in combination with the ASCII flag. The following code examples show how to define ASCII accelerators.

```
"A",    ID_ACCEL1          ; SHIFT+A
65,     ID_ACCEL2, ASCII   ; SHIFT+A
```

A keystroke that generates a virtual-key code is specified differently depending on whether the keystroke is an alphanumeric key or a non-alphanumeric key. For an alphanumeric key, the key's letter or number, enclosed in double quotation marks, is combined with the VIRTKEY flag. For a non-alphanumeric key, the Windows CE virtual-key code for the specific key is combined with the VIRTKEY flag. The following code examples show how to define virtual-key code accelerators.

```
"a",          ID_ACCEL3, VIRTKEY    ; A (caps-lock on) or an
VK_INSERT,    ID_ACCEL4, VIRTKEY    ; INSERT key
```

If you want the user to press the ALT, SHIFT, or CTRL key in some combination with the accelerator keystroke, specify the ALT, SHIFT, and CONTROL flags in the accelerator definition. The following code examples show possible combinations.

```
"B",    ID_ACCEL5, ALT                      ; ALT+SHIFT+B
"I",    ID_ACCEL6, CONTROL, VIRTKEY         ; CTRL+I
VK_F5, ID_ACCEL7, CONTROL, ALT, VIRTKEY    ; CTRL+ALT+F5
```

You can also set the NOINVERT flag in the *options* parameter. NOINVERT prevents the selected menu item from being highlighted when its accelerator key is pressed. This is useful when defining accelerators for actions such as scrolling that do not correspond to a menu item.

Loading and Activating an Accelerator Table

An application loads an accelerator-table resource by calling the **LoadAccelerators** function and specifying the instance handle to the application whose executable file contains the resource and the name or identifier of the resource. **LoadAccelerators** loads the specified accelerator table into memory and returns the handle to the accelerator table.

An application can load an accelerator-table resource at any time. Usually, a single-threaded application loads its accelerator table before entering its main message loop. An application that uses multiple threads typically loads the accelerator-table resource for a thread before entering the message loop for the thread. An application or thread might also use multiple accelerator tables, each associated with a particular application window. This type of application loads the accelerator table for the window each time the user activates the window.

Windows CE maintains accelerator tables for each application. An application can define any number of accelerator tables for use with its own windows. A unique 32-bit handle, **HACCEL**, identifies each table. However, only one accelerator table can be active at a time for a specified thread.

To activate an accelerator table, call the **TranslateAccelerator** function in the message loop associated with the thread message queue to process accelerator keystrokes for a specified thread. The handle of the accelerator table passed to the **TranslateAccelerator** function determines which accelerator table is active for a thread. This function also monitors keyboard input to the message queue, checking for key combinations that match an entry in the accelerator table. When **TranslateAccelerator** finds a match, it translates the keyboard input—that is, the WM_KEYUP and WM_KEYDOWN messages—into a WM_COMMAND or WM_SYSCOMMAND message. It then sends the message to the window procedure of the specified window. The WM_COMMAND message includes the identifier of the accelerator that caused **TranslateAccelerator** to generate the message. The window procedure examines the identifier to determine the source of the message, and then it processes the message accordingly.

The following code example shows how to call **TranslateAccelerator** from within a message loop.

```
while (GetMessage (&msg, NULL, 0, 0))
{
  if (!TranslateAccelerator (
                  g_hwndMain,   // Handle to the destination window.
                  hAccel,       // Handle to the accelerator table.
                  &msg))        // Address of the message data.
  {
    TranslateMessage (&msg);
    DispatchMessage (&msg);
  }
}
```

Note Unlike Windows-based desktop platforms, Windows CE does not maintain a system-wide accelerator table that applies to all applications.

To change the active accelerator table, pass a different accelerator-table handle to **TranslateAccelerator**.

Creating Dialog Boxes

A *dialog box* is a temporary window that contains controls. A dialog box is used to display status information and to prompt the user for input. Windows CE supports two types of dialog boxes: *modal* and *modeless*. A modal dialog box requires the user to supply information or dismiss the dialog box before enabling the application to continue. Applications use modal dialog boxes in conjunction with commands that require additional information before they can proceed. A modeless dialog box enables the user to supply information and return to a previous task without closing the dialog box. Modal dialog boxes are simpler to manage than their modeless counterparts because they are created and destroyed by calling a single function.

To create either a modal or modeless dialog box, you must define a *dialog box template* to describe the dialog box style and content. The dialog box template is a binary description of the dialog box and the controls that it contains. You must also supply a dialog box procedure to execute tasks. You can create this template as a resource that you can load from your executable file.

▶ To define a dialog box template

1. Define an identifier for the dialog box in a header file.
2. Define a dialog box in the application resource file with the **DIALOG** statement. The statement defines the position and dimensions of the dialog box on the screen as well as the dialog box style, and has the parameters described in the following table.

Parameter	Description	Use
nameID	Dialog box name	Specifies a unique identifier for the dialog box
x	*x* coordinate	Specifies the *x* coordinate of the upper-left corner of the dialog box
y	*y* coordinate	Specifies the *y* coordinate of the upper-left corner of the dialog box
width	Dialog box width	Specifies the width of the dialog box
height	Dialog box height	Specifies the height of the dialog box
optional-statements	Dialog box options	Specifies one or more features of the dialog box
control-statement	Controls associated with the dialog box	Specifies one or more controls using the appropriate control statement

For a listing of option and control statements, see the *Windows CE API Reference*.

The following code example shows how to use the **DIALOG** statement to define a dialog box template.

```
#include <windows.h>

#define IDD_REPLACE                     106
#define IDC_FINDWHAT                    1000
#define IDC_REPLACE                     1001
#define IDC_BTNREPLACE                  1008
#define IDC_BTNREPLACEALL               1009
#define IDC_BTNFINDNEXT                 1010
#define IDC_STATIC1                     1011
#define IDC_STATIC2                     1012

IDD_REPLACE DIALOG DISCARDABLE  0, 0, 191, 73
STYLE DS_MODALFRAME | DS_CENTER | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Replace"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_FINDWHAT,55,4,128,14,ES_AUTOHSCROLL
    EDITTEXT        IDC_REPLACE,55,21,128,14,ES_AUTOHSCROLL
    PUSHBUTTON      "&Find Next",IDC_BTNFINDNEXT,18,38,71,14
    PUSHBUTTON      "&Replace",IDC_BTNREPLACE,18,55,71,14
    PUSHBUTTON      "Replace &All",IDC_BTNREPLACEALL,101,38,71,14
    PUSHBUTTON      "Cancel",IDCANCEL,101,55,71,14
    LTEXT           "Fi&nd what:",IDC_STATIC1,7,4,43,8
    LTEXT           "Re&place with:",IDC_STATIC2,7,21,43,8
END
```

The *dialog box procedure* is an application-defined callback function that the system calls when it has input for the dialog box or tasks for the dialog box to execute. A dialog box procedure is similar to a window procedure in that the system sends messages to the procedure when it has data or tasks to execute. Although a dialog box procedure is similar to a window procedure, it does not have the same responsibilities. Unlike a window procedure, a dialog box procedure never calls the **DefWindowProc** function. Rather, it returns TRUE if it processes a message or FALSE if it does not.

The following code example shows the form of a dialog box procedure.

```
BOOL CALLBACK ReplaceDialogProc (
                    HWND hwndDlg,   // Handle to the dialog box.
                    UINT uMsg,      // Message.
                    WPARAM wParam,  // First message parameter.
                    LPARAM lParam)  // Second message parameter.
{
  switch(uMsg)
  {
```

```
        case WM_INITDIALOG:
          // Insert code here to put the string (to find and replace with)
          // into the edit controls.
          // ...
          return TRUE;

        case WM_COMMAND:
          switch (LOWORD(wParam))
          {
            case IDC_BTNFINDNEXT:
              // Insert code here to handle the case of pushing the
              // "Find Next" button.
              // ...
              break;

            case IDC_BTNREPLACE:
              // Insert code here to handle the case of pushing the
              // "Replace" button.
              // ...
              break;

            case IDC_BTNREPLACEALL:
              // Insert code here to handle the case of pushing the
              // "Replace All" button.
              // ...
              break;

            case IDCANCEL:
              EndDialog(hwndDlg, 0);
              break;
          }
          return TRUE;
      }
      return FALSE;
    }
```

Here, the *hwndDlg* parameter receives the window handle of the dialog box.

Most dialog box procedures process the WM_INITDIALOG message and the WM_COMMAND messages sent by the controls, but process few if any other messages. If a dialog box procedure does not process a message, it must return FALSE to direct the system to process the messages internally. The only exception to this rule is the WM_INITDIALOG message. Typically, you initialize the dialog box and its contents while processing the WM_INITDIALOG message. The most common task you perform with the WM_INITDIALOG message is to initialize the controls to reflect the current dialog box settings. Another common task is to center a dialog box on the screen or within its owner window. A useful task for some dialog boxes is to set the input focus to a given control rather than accept the default input focus. The dialog box procedure must return TRUE to direct the system to process the WM_INITDIALOG message.

Typically, you create a dialog box by calling either the **DialogBox** or **CreateDialog** function. **DialogBox** creates a modal dialog box; **CreateDialog** creates a modeless dialog box. When calling these functions, you must specify the identifier or name of a dialog box template resource and the address of the dialog box procedure. The **DialogBox** and **CreateDialog** functions load the specified dialog box template, display the dialog box, and process all user input until the user closes the dialog box.

The following code example shows how to create a modal dialog box.

```
DialogBox (
        g_hInst                         // Handle to the application
                                        // instance.
        MAKEINTRESOURCE (IDD_REPLACE),  // Identifies the dialog box
                                        // template.
        g_hwndMain,                     // Handle to the owner window.
        (DLGPROC) ReplaceDialogProc);   // Pointer to the dialog box
                                        // procedure.
```

The following code example shows how to create a modeless dialog box.

```
CreateDialog (
        g_hInst,                        // Handle to the application
                                        // instance.
        MAKEINTRESOURCE (IDD_REPLACE),  // Identifies the dialog box
                                        // template.
        g_hwndMain,                     // Handle to the owner window.
        (DLGPROC) ReplaceDialogProc);   // Pointer to the dialog box
                                        //procedure.
```

Dialog boxes usually belong to a predefined, exclusive window class. The system uses this window class and its corresponding window procedure for both modal and modeless dialog boxes. When the function is called, it creates the window for the dialog box, as well as the windows for the controls in the dialog box, and then it sends selected messages to the dialog box procedure.

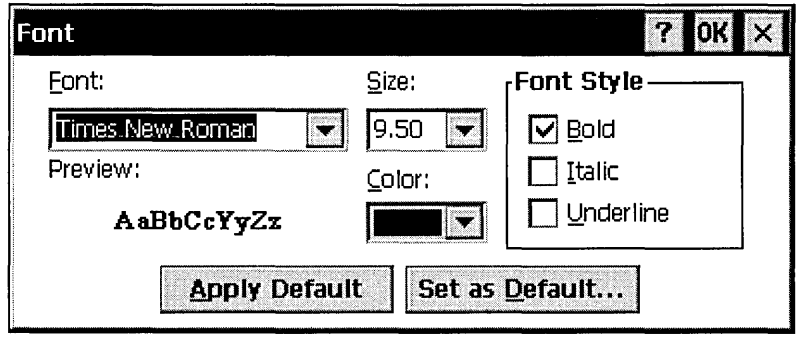
While the dialog box is visible, the predefined window procedure manages all messages, processing some messages and passing others to the dialog box procedure so that the procedure can execute tasks. You do not have direct access to the predefined window class or window procedure, but you can use the dialog box template and dialog box procedure to modify the style and behavior of a dialog box.

The following table shows dialog box types.

Dialog box type	Description
Application-defined dialog box	Helps a user perform application-specific tasks
Common dialog box	Provides a familiar way for users to perform common application tasks
Message box	Notifies a user of an event or situation and offers limited responses
Property sheet, which is a collection of tabbed dialog boxes	Provides a convenient way to view and modify object properties

Application-Defined Dialog Boxes

An application-defined dialog box is a child window that you design to suit the needs of your application. You can use any format for a dialog box; in addition, you can use any kind of control in a dialog box. The following screen shot shows an application-defined dialog box.



In Windows CE, all dialog boxes are control parents. They are also recursive. This means that if a dialog box has a child dialog box when a user tabs through the parent dialog box, the dialog box manager tabs into the child dialog box as well. If a dialog box is outside the visible area of the screen, Windows CE does not automatically reposition it.

If a user presses ALT+H while the dialog box has the input focus, the system posts a WM_HELP message to the dialog box procedure. Respond to this message by displaying context-sensitive Help for the dialog box.

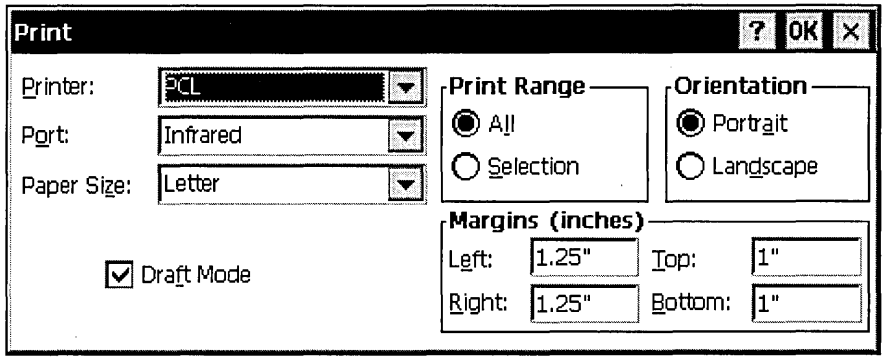
Sometimes it is necessary for a dialog box to appear on top of all other windows. For example, under low memory conditions, the **System Out of Memory** dialog box will send a WM_CLOSE message to an application. If the application is not in the foreground, any dialog box it displays will be hidden behind the current foreground window unless you create the dialog box with the DS_SETFOREGROUND style. Because putting the dialog box in the foreground will not bring the application's main window forward, put in the dialog box any information that a user might need to decide what action to take.

In Windows CE, dialog boxes have the WS_POPUP style by default. If you want to use the WS_CHILD style, specify it in the *style* member of the **DLGTEMPLATE** structure you pass in the *lpTemplate* parameter to any of these functions. You can also specify the DS_SETFOREGROUND or DS_CENTER style.

For a list of dialog box styles that are supported by Windows CE, see Appendix A, "Window and Control Styles."

Common Dialog Boxes

A common dialog box is a system-defined dialog box that standardizes how users perform complex operations that are common to most applications. Windows CE supports the **Color**, **Open**, **Save As**, and **Print** common dialog boxes. The following screen shot shows a **Print** dialog box.



The following table shows each of the common dialog boxes that are supported in Windows CE.

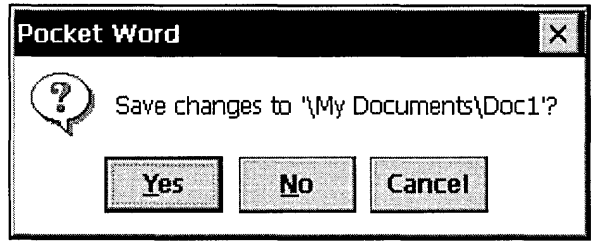
Common dialog box	Description
Color	Provides users with a way to select a color from a set of custom colors or from a set of basic colors as determined by the display driver.
Open	Provides users with a way to select a file to open.
Save As	Provides users with a way to save a file under another file name.
Print	Provides users with a way to select print options.

Users must print the entire document or the currently selected portion and can print only one copy at a time. The settings in the **Print** dialog box initialize to the default printer. If a user has never used the **Print** dialog box before, the first printer registered in the registry is the default. After that, the last printer that the user selected is the default. You can set the widths and minimum widths of the left, top, right, and bottom margins of the printed page by including values for the *rcMargin* and *rcMinMargin* members of the **PRINTDLG** structure.

Common dialog boxes are centered vertically and horizontally on the screen and are not movable. They always have the **Help** button displayed.

Message Boxes

A *message box* is a special kind of modal dialog box that an application uses to display messages and prompt for input. A message box typically contains a text message and one or more predefined buttons. The following screen shot shows a message box.



To create a message box, call the **MessageBox** function and specify the text and the number and types of buttons to display; because Windows CE controls the message box creation and management, you do not need to provide a dialog box template and dialog box procedure. Windows CE creates its own template based on the text and buttons specified for the message box and supplies its own dialog box procedure.

As with dialog boxes, sometimes it is necessary for a message box to appear on top of all other windows. In particular, under low memory conditions, the **System Out of Memory Dialog Box** sends a WM_CLOSE message to an application. If the application is not in the foreground, any message box it brings up is hidden behind the current foreground window unless you create the message box with the MB_SETFOREGROUND style. Because putting the message box in the foreground will not bring the application's main window forward, put any information in the message box that a user might need in order to decide what action to take.

The **MessageBeep** function, generally used with message boxes, plays a waveform sound. The waveform sound for each sound type is identified by an entry in the sounds section of the registry.

For a list of message box styles supported by Windows CE, see Appendix A, "Window and Control Styles."

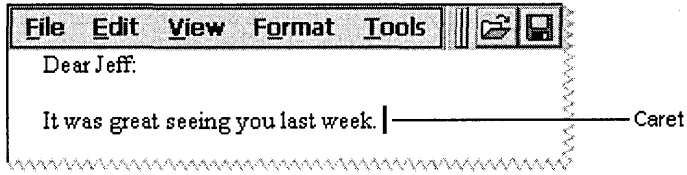
Creating a Caret

A *caret* is a flashing line or block in the window client area that indicates the place where text or graphics will be inserted. Windows CE provides one caret per message queue. You should create a caret only when its associated window has the keyboard focus or is active. You should destroy the caret before the window loses the keyboard focus or becomes inactive. Once you create a caret, you can change how frequently a caret blinks, modify the caret position within a window, or temporarily remove a caret from view by hiding it. An application should create and display a caret while processing the WM_SETFOCUS message. Windows CE sends the WM_SETFOCUS message to a window when it receives the keyboard focus.

▶ **To create and display a caret in a window**

1. Call the **CreateCaret** function when the window receives focus. Windows CE formats a caret by inverting the pixel color within the rectangle specified by the caret's position, width, and height.
2. Set the caret position by calling the **SetCaretPos** function.
3. Make the caret visible by calling the ShowCaret function. When the caret appears, it begins flashing.

The following screen shot shows a caret as it appears in text.



The elapsed time, in milliseconds, that is required to invert the caret is called the *blink time*. The *flash time* is the elapsed time, in milliseconds, that is required to display, invert, and restore the caret's display. The flash time of a caret is twice as much as the blink time. The caret will blink as long as the thread that owns the message queue processes messages. A user can set the blink time of the caret by using Control Panel, and applications should maintain the settings that the user has chosen. An application can determine the caret blink time by using the **GetCaretBlinkTime** function. If you are writing an application that enables the user to adjust the blink time, such as a Control Panel application, use the **SetCaretBlinkTime** function to set the blink time rate to a specified number of milliseconds.

To determine the caret position, use the **GetCaretPos** function. An application can move a caret in a window by using the **SetCaretPos** function. A window can move a caret only if it already owns the caret. **SetCaretPos** can move the caret whether or not it is visible.

You can temporarily remove a caret by hiding it, or you can permanently remove the caret by destroying it. To hide the caret, use the **HideCaret** function. This is useful when the application must redraw the screen while processing a message, but must keep the caret out of view. When the application finishes drawing, it can display the caret again by using the **ShowCaret** function. Hiding the caret does not destroy its shape or invalidate the insertion point. Hiding the caret is cumulative; that is, if the application calls **HideCaret** five times, it must also call **ShowCaret** five times before the caret will reappear.

To remove the caret from the screen and destroy its shape, use the **DestroyCaret** function. **DestroyCaret** destroys the caret only if the window involved in the current task owns the caret.

Creating a Cursor

A *cursor* is a small bit image that reflects the position of a pointing device.

Because standard cursors are predefined, it is unnecessary to create them. To use a standard cursor, an application retrieves a *cursor handle* by calling the **LoadCursor** function. A cursor handle is a unique value of the **HCURSOR** type that identifies a standard or custom cursor.

The following code example shows the syntax for the **LoadCursor** function.

```
HCURSOR LoadCursor(
   HINSTANCE hInstance, // Handle to the application instance
   LPCTSTR lpCursorName // Name string or cursor resource identifier
);
```

Here, *hInstance* is a handle to an instance of the module whose executable file contains the cursor to be loaded, and *lpCursorName* is a pointer to the name of the cursor to be loaded. It can also point to a resource identifier. To use a predefined cursor, the application must set *hInstance* to NULL and *lpCursorName* to one of the predefined cursor values.

Windows CE–based platforms implement cursors in different ways depending on the platform configuration. For example, on many Windows CE–based platforms, users interact with applications by tapping the stylus on the screen; because there is no mouse, there is no need for a cursor to indicate the current mouse position. Target platforms not requiring mouse support typically implement Iconcurs.dll. This component enables you to specify only the wait cursor when calling the **LoadCursor** function. Applications should display the wait cursor when executing a command that renders the current window or the system unresponsive to user input. To establish the shape of a wait cursor, you must call the **SetCursor** function in conjunction with the **LoadCursor** function. The following code example shows how to establish the shape of a wait cursor.

```
SetCursor(LoadCursor(NULL, IDC_WAIT));
```

Target platforms that support mouse cursors typically include Mcursor.dll. This component implements cursors similar to Windows-based desktop platforms; all standard cursors, except color cursors, are available when calling the **LoadCursor** function. Windows CE also supports custom cursors.

▶ **To create a custom cursor**

1. Draw the cursor by using a graphics application.
2. Include the cursor as a resource in the application resource-definition file.

   Using a cursor resource avoids device dependence, simplifies localization, and enables applications to share cursor designs.
3. Call **LoadCursor** at run time to retrieve the cursor handle.

   Cursor resources contain data for several different display devices. The **LoadCursor** function automatically selects the most appropriate data for the current display device. To load a cursor directly from a .cur or .ani file, use the **LoadCursorFromFile** function instead of the **LoadCursor** function.

Once you create and load a cursor, you can hide and redisplay the cursor, without changing the cursor design, by using the **ShowCursor** function. This function uses an internal counter to determine when to hide or display the cursor. An attempt to show the cursor increments the counter; an attempt to hide the cursor decrements the counter. The cursor is visible only if this counter is greater than or equal to zero.

Additionally, you can change the design of the cursor by using the **SetCursor** function and specifying a different cursor handle.

Creating Icons, Bitmaps, Images, and Strings

An *icon* is an image that is used to identify an application, file, or other object. It consists of a bit image combined with a mask. An application icon always appears on the taskbar while the application is running, and it can be used to recover the application main window when another window has the foreground. The icon can also be used to identify the application in the Windows CE Explorer. Every application should register both 16 by 16 pixel and 32 by 32 pixel icons for its main executable file and the types of files it stores in the file system.

A *bitmap* is a graphics image that you can include in an application. Unlike icons, which have a fixed size, you determine the bitmap size. Both icons and bitmaps must be defined in a resource file.

▶ **To create an icon**

1. Draw the icon by using a graphics application.
2. Include the icon as a resource in the application resource-definition file.
3. Call the WM_SETICON message to associate an icon with a window class.

   Icons are associated with window classes rather than with individual windows.
4. Call the WM_GETICON message to retrieve the handle of the icon that is associated with a window class.

Note Windows CE does not support any of the standard predefined icons (IDI_*) that Windows-based desktop platforms support.

▶ To create a bitmap

1. Draw the bitmap by using a graphics application.
2. Include the bitmap as a resource in the application resource-definition file.
3. Call the **LoadBitmap** function to initialize the bitmap.

   The bitmap you create with this function will be read-only because Windows CE does not copy the bitmap into RAM as Windows-based desktop platforms do.
4. Call the **SelectObject** function to select the bitmap into a device context. This enables you to display the bitmap.

   When you select the bitmap into a device context, you cannot modify the device context—for example, by drawing text into it—because that would require the ability to write to the bitmap.

Images and strings are created similarly to icons and bitmaps. Both are resources that must be defined in a resource file. The following code example shows how to define a resource for an icon, bitmap, cursor, and string.

```
#include "windows.h"

#define IDB_BITMAP                      101
#define IDC_CURSOR                      102
#define IDI_CEPADICON                   103

IDB_CEPADBITMAP     BITMAP  DISCARDABLE     "CePad.bmp"
IDC_CEPADCURSOR     CURSOR  DISCARDABLE     "CePad.cur"
IDI_CEPADICON       ICON    DISCARDABLE     "CePad.ico"

STRINGTABLE DISCARDABLE
BEGIN
  1                 "CePad"
  2                 "CePad Application"
END
```

When adding an image or string to an application, call the **LoadImage** function to load an image and call the **LoadString** function to load a string. Windows CE does not support stretching and shrinking of images or any loading options other than LR_DEFAULTCOLOR. Windows CE supports only Unicode strings.

Creating Timers

A *timer* is a system resource that can notify an application at regular intervals. An application associates a timer with a window and sets the timer for a specific time-out period. Each time the specified interval, or *time-out value*, for a specified timer elapses, the system uses a WM_TIMER message to notify the window associated with the timer; because the accuracy of a timer depends on the system clock rate and how often the application retrieves messages from the message queue, the time-out value is only approximate. The smallest possible interval that a timer can measure is the system tick interval.

To create a timer, call the **SetTimer** function. The timer can be associated with a particular window or with just the thread. If you associate the timer with a window, message loop processing will cause the WM_TIMER message to be dispatched to the window procedure for the window. If you do not associate the timer with a window, you must design the message loop to recognize and handle the WM_TIMER message.

If the call to **SetTimer** includes a **TimerProc** callback function, the procedure is called when the timer expires. This call is done inside the **GetMessage** or **PeekMessage** function. This means that a thread must be executing a message loop to service a timer, even if you are using a timer callback procedure.

A new timer starts timing its interval as soon as it is created. An application can change a time-out value for a timer by calling the **SetTimer** function, and it can destroy a timer by calling the **KillTimer** function. To use system resources efficiently, applications should destroy unnecessary timers.

You can use the timer and window identifiers to identify timers that are associated with a window. You can identify timers that are not associated with a particular window by using the identifier that is returned by the **SetTimer** call.

Timer messages have low priority in the message queue. Although you know that the window associated with a timer is notified sometime after the timer interval expires, you cannot know the exact time it will receive the notification.

Timers expire at regular intervals, but a timer that expires multiple times before being serviced does not generate multiple WM_TIMER messages.

CHAPTER 4

Creating Controls

This chapter explains how to add window, common, and Windows CE–specific controls to windows and dialog boxes. It also explains how to handle control notification messages. The window controls discussed in this chapter include buttons, combo boxes, edit controls, list boxes, scroll bars, and static controls.

A *control* is a child window that an application uses in conjunction with another window to perform simple I/O tasks. Windows CE defines two basic kinds of controls: *window controls* and *common controls*. Window controls all send WM_COMMAND messages. Common controls generally send WM_NOTIFY messages, though a few send WM_COMMAND messages.

Windows CE also supports two Windows CE–specific controls: an *HTML viewer control* and a *Rich Ink control.* These controls are neither window controls nor common controls. The HTML viewer control provides a simple interface for rendering HTML text, displaying embedded images, and notifying the application of user events. The Rich Ink control enables a user to write and draw on a touch-sensitive screen with a pointing device.

Controls are most often placed within dialog boxes, but they can also be placed directly on the surface of a normal window client area. Each control has attributes that affect its appearance and behavior. When you create a control, you can apply one or more styles to the control. For a list of dialog box styles supported by Windows CE, see Appendix A, "Window and Control Styles."

Working with Window Controls

A *window control* is a predefined child window that enables a user to make selections, carry out commands, and perform I/O tasks. You can place a window control within a dialog box or in the client area of a normal window. Controls placed within dialog boxes provide a user with the means to type text, select options, and direct a dialog box to complete its action. Controls placed in normal windows provide a variety of services, such as choosing commands, scrolling, and viewing and editing text.

Although you can create your own window controls, Windows CE has several predefined window classes that you can use to add a standard window control to your application. The following table shows predefined window classes supported by Windows CE.

Window class	Description
BUTTON	Creates a button control, which notifies the parent window when a user selects the button.
COMBOBOX	Creates a combo box—a combination of list box and edit control—that enables a user to select and edit items.
EDIT	Creates an edit control, which lets a user view and edit text.
LISTBOX	Creates a list box, which displays a list from which a user can select one or more items.
SCROLLBAR	Creates a scroll bar control, which enables a user to scroll horizontally and vertically within a window.
STATIC	Creates a static control, which often acts as a label for another control; static controls can display both text and images such as icons.

Because window controls are child windows, create a window control by calling the **CreateWindowsEx** function. This creates a single control in a normal window. To create a control in a dialog box, use the dialog box template contained in your application resource file. By using a resource file, you can create multiple controls simultaneously. For more information about resources and resource files, see Chapter 3, "Using Resources."

Most compilers come with automated tools, called resource editors, to create resources. Using a resource editor is probably the most accurate and efficient way to add a control to a dialog box. However, because resource editors vary, providing instruction for using a resource editor is beyond the scope of this book.

To use a window control, you must include either the Windows.h or the Winuser.h header file in your application. Windows.h includes Winuser.h.

▶ To create a window control in a normal window

1. Define an identifier for the control in the application header file.

   A *control identifier* is a value that uniquely identifies a control sending the message. In Windows CE, control identifiers are valid only for child windows.

2. Call the **CreateWindowEx** function and specify the following parameters.

Parameter	Description	Use
DWORD *dwExStyle*	Extended window style	Specify an extended window style.
LPCTSTR *lpClassName*	Class name	Specify a predefined window class. For example, to create a push button, specify "button."
LPCTSTR *lpWindowName*	Window text	Specify the text you want to appear on the control.
DWORD *dwStyle*	Window style	Specify a control style. Each predefined window class has a corresponding set of control styles that enables an application to vary the appearance and behavior of the controls it creates. For example, the BUTTON class supports styles to create a push button, radio button, check box, or group box.
int *x*	*x* coordinate	Specify the *x* coordinate of the upper-left corner of the control relative to the upper-left corner of the parent window client area.
int *y*	*y* coordinate	Specify the upper-left corner *y* coordinate of the control relative to the upper-left corner of the parent window client area.
int *nWidth*	Width	Specify the control width.
int *nHeight*	Height	Specify the control height.
HWND *hWndParent*	Parent window	Specify the handle to the parent window, *hWnd*.
HMENU *hMenu*	Child window identifier	Specify the control identifier.
HINSTANCE *hInstance*	Instance handle	Specify the application or module to be associated with the window.
LPVOID *lpParam*	Extra parameters	Specify NULL when creating a control.

Once you call **CreateWindowEx**, Windows CE handles all repainting tasks. It also destroys all controls upon the termination of the application.

The following code example shows how to add a control to a normal window using the **CreateWindowEx** function.

```
DWORD dwStyle = WS_VISIBLE | WS_CHILD | TVS_HASLINES | TVS_LINESATROOT |
                TVS_HASBUTTONS;

hwndTreeView = CreateWindow (
      WC_TREEVIEW,          // Class name
      TEXT ("Tree View"),   // Window name
      dwStyle,              // Window style
      0,                    // x coordinate of the upper-left corner
      0,                    // y coordinate of the upper-left corner
      CW_USEDEFAULT,        // The width of the treeview control window
      CW_USEDEFAULT,        // The height of the treeview control window
      hwnd,                 // Window handle of parent window
      (HMENU)IDC_TREEVIEW,  // The treeview control identifier
      hInst,                // The instance handle
      NULL);                // Specify NULL for this parameter when
                            // creating a control
```

▶ To create a control in a dialog box

1. Define an identifier for each control in a header file.
2. Define a dialog box in your application's resource file using the **DIALOG** statement. The statement defines the position and dimensions of the dialog box on the screen as well as the dialog box style, and has the following parameters.

Parameter	Description	Use
nameID	Dialog box name	Specify a unique identifier for the dialog box.
x	*x* coordinate	Specify the *x* coordinate of the upper-left corner of the dialog box.
y	*y* coordinate	Specify the *y* coordinate of the upper-left corner of the dialog box.
Width	Dialog width	Specify the width of the dialog box.
Height	Dialog height	Specify the height of the dialog box.

Parameter	Description	Use
Option-statements	Dialog box options	Specify one or more features of the dialog box. For example, use **CAPTION** to add a title to the dialog box or **DISCARDABLE** to remove the dialog box from memory when not in use. For a listing of option statements, see the **DIALOG** statement in the Windows CE API Reference.
Control-statements	Controls associated with the dialog box	Specify one or more controls using the appropriate **CONTROL** statement.

3. Call either the **DialogBox** function or the **CreateDialog** function and specify the identifier or name of the dialog box template and the address of the dialog box procedure.

   DialogBox creates a modal dialog box and **CreateDialog** creates a modeless dialog box. For more information about creating dialog boxes, see Chapter 2, "Working with Windows and Messages."

The following code example shows how to create a push button and static text control in a dialog box.

```
#include <windows.h>

#define IDD_ABOUT               103
#define IDC_STATIC              -1

IDD_ABOUT DIALOG DISCARDABLE  0, 0, 132, 55
STYLE DS_MODALFRAME | DS_CENTER | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About CE Pad"
FONT 8, "MS Sans Serif"
BEGIN
  DEFPUSHBUTTON   "OK",IDOK,39,34,50,14
  CTEXT           "Microsoft Window CE",IDC_STATIC,7,7,118,8
  CTEXT           "CePad Sample Application",IDC_STATIC,7,19,118,8
END
```

Handling Notification Messages

All window controls respond to user input or changes to the control by sending a *notification message* to its parent window. A notification message is a WM_COMMAND message that includes a control identifier and a notification code identifying the nature of the event. An application must trap these notification messages and react to them.

The following code example shows one method of trapping a WM_COMMAND message.

```
BOOL CALLBACK AboutDialogProc (
                        HWND hwndDlg,     // Handle to the dialog box
                        UINT uMsg,        // Message
                        WPARAM wParam,    // First message parameter
                        LPARAM lParam)    // Second message parameter
{
  switch (uMsg)
  {
    case WM_INITDIALOG:
      return TRUE;

    case WM_COMMAND:
      switch (LOWORD (wParam))
      {
        case IDOK:
          EndDialog (hwndDlg, IDOK);
          return TRUE;

        case IDCANCEL:
          EndDialog (hwndDlg, IDCANCEL);
          return TRUE;
      }
      break;
  }
  return FALSE;
}
```

Some window controls receive messages as well as generate them. Typically, a window procedure sends a message to a control directing it to execute a task. The control processes the message and carries out the requested action. Windows CE has several predefined messages, such as WM_GETTEXT and WM_GETDLGCODE, that it sends to controls. These messages typically correspond to window-management functions that carry out actions on windows. The window procedure for an application-defined control processes any predefined control message that affects the operation of the control.

The following table shows these messages.

Message	Recommendation
WM_GETDLGCODE	Process if the control uses the ENTER, ESC, TAB, or arrow keys. The **IsDialogMessage** function sends this message to controls in a dialog box to determine whether to process the keys or pass them to the control.
WM_GETFONT	Process if the WM_SETFONT message is also processed.
WM_GETTEXT	Process if the control text is not the same as the title specified by the **CreateWindowEx** function.
WM_GETTEXTLENGTH	Process if the control text is not the same as the title specified by the **CreateWindowEx** function.
WM_KILLFOCUS	Process if the control displays a caret, a focus rectangle, or another item to indicate that it has the input focus.
WM_SETFOCUS	Process if the control displays a caret, a focus rectangle, or another item to indicate that it has the input focus.
WM_SETTEXT	Process if the control text is not the same as the title specified by the **CreateWindowEx** function.
WM_SETFONT	Process if the control displays text. Windows CE sends this message when creating a dialog box that has the DS_SETFONT style.

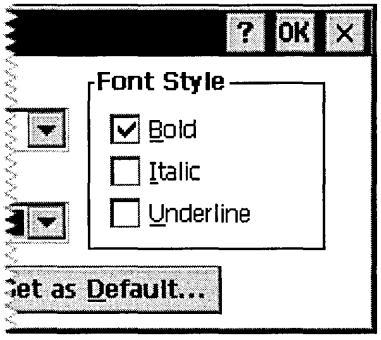
You can also send messages to a control by calling the **SendMessage** function. One control that calls the **SendMessage** function to receive messages is the button.

Creating a Button

A button is a window control that a user can turn on or off to provide input to an application. Buttons can be used alone or in groups and can appear with or without a label. Buttons belong to the BUTTON window class.

Windows CE provides four kinds of buttons: check boxes, push buttons, radio buttons, and group boxes. Each button type has one or more styles that affect its appearance, behavior, or both.

A check box contains one or more items that appear checked when selected. More than one item in a check box can be selected at one time. Applications display check boxes in a group box to enable a user to choose from a set of related, but independent, options. When a user selects a check box of any style, the check box receives the keyboard focus from Windows CE, which sends the check box parent window a WM_COMMAND message containing the BN_CLICKED notification code. The parent window does not acknowledge this message if the message is sent from an automatic check box or automatic three-state check box because Windows CE sets the check state for those styles. The parent window must acknowledge the message if the message is sent from an application-defined check box or three-state check box because the parent window, not Windows CE, is responsible for setting the check state for those styles. Regardless of the check box style, Windows CE repaints the check box once its state is changed. The following screen shot shows a check box.



▶ **To create a check box using the CreateWindow function**

1. Specify the BUTTON window class in the *lpClassName* parameter of the **CreateWindow** or **CreateWindowEx** function.
2. Specify one or more check box styles in the *dwStyle* parameter of the **CreateWindow** or **CreateWindowEx** function.

▶ **To create a check box in a dialog box**

- Add the following **CHECKBOX** resource-definition statement to your **DIALOG** resource.

```
CHECKBOX text, id, x, y, width, height [[, style [[, extended-
style]]]]
```

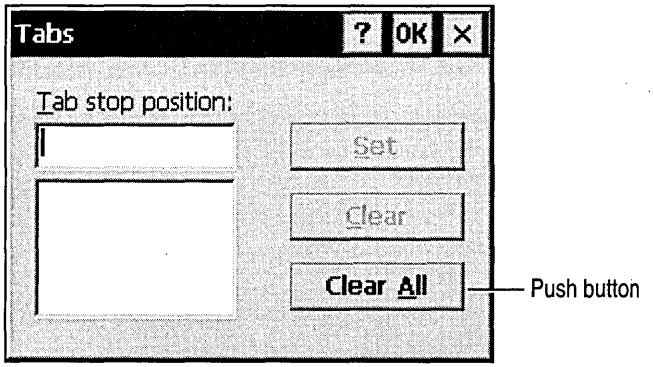
Here, *text* is the text displayed to the right of the control and *id* is the value that identifies the check box. The upper-left corner of the control is positioned at *x*, *y*, and its dimension is determined by *width* and *height*. *Style* and *extended-style* determine the appearance of the check box. The **CHECKBOX** resource statement creates a manual check box. This means that your application must manually check and uncheck the box each time a user selects the control. If you want Windows CE to toggle between checked and unchecked states when a user selects the control, use the **AUTOCHECKBOX** resource statement.

A *push button*, also known as a command button, is a small, rectangular control that a user can turn on or off by tapping it with the stylus. A push button has a raised appearance in its default, or off state, and a depressed appearance in its on state. Windows CE supports owner-drawn push buttons discussed later in this chapter.

When a user taps a push button, it receives the keyboard focus from Windows CE, which sends the button's parent window a WM_COMMAND message containing the BN_CLICKED notification code. In response, the dialog box closes and carries out the operation indicated by the button.

Note Windows CE does not support the BS_BITMAP, BS_FLAT, BS_ICON, BS_PUSHBOX, BS_TEXT, or BS_USERBUTTON styles. Use the BS_OWNERDRAW style to create the effects you would otherwise achieve by using the BS_BITMAP, BS_ICON, or BS_USERBUTTON button styles.

The following screen shot shows a push button for the **Tabs** dialog box.



▶ **To create a push button using CreateWindow**

1. Specify the BUTTON window class in the *lpClassName* parameter of the **CreateWindow** or **CreateWindowEx** function.
2. Specify one or more push box styles in the *dwStyle* parameter of the **CreateWindow** or **CreateWindowEx** function.

▶ **To create a push button in a dialog box**

- Add the following **PUSHBUTTON** resource-definition statement to your **DIALOG** resource.

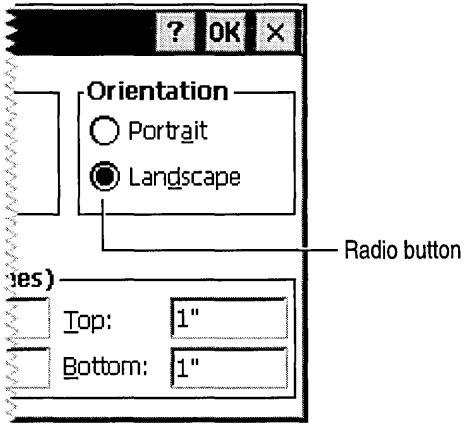
```
PUSHBUTTON "string", id, x, y, width, height [[, style [[, extended-
style]]]]
```

Here, *string* is the text you want displayed inside the push button, and *id* is the value that identifies the push button. The upper-left corner of the control is positioned at *x*, *y*, and its dimension is determined by *width* and *height*. *Style* and *extended-style* determine the push button appearance.

A *radio button*, also known as an option button, is similar to a check box in that you can select from one or more options. Unlike a check box, however, when there are multiple radio buttons only one item can be selected, making radio buttons mutually exclusive.

When a user selects an automatic radio button, Windows CE sets the check state of all other radio buttons within the same group to unchecked. For standard radio buttons, use the WS_GROUP style to achieve the same effect.

Windows CE supports most of the radio button styles that Windows-based desktop platforms support; it does not support the BS_LEFTTEXT style that places the radio button to the right of the associated text. You can achieve the same effect by using the BS_RIGHTBUTTON style. The following screen shot shows a radio button.



▶ **To create a radio button using CreateWindow**

1. Specify the BUTTON window class in the *lpClassName* parameter of the **CreateWindow** or **CreateWindowEx** function.
2. Specify one or more radio button styles in the *dwStyle* parameter of the **CreateWindow** or **CreateWindowEx** function.

▶ **To create a radio button in a dialog box**

- Add the following **RADIOBUTTON** resource-definition statement to your **DIALOG** resource.

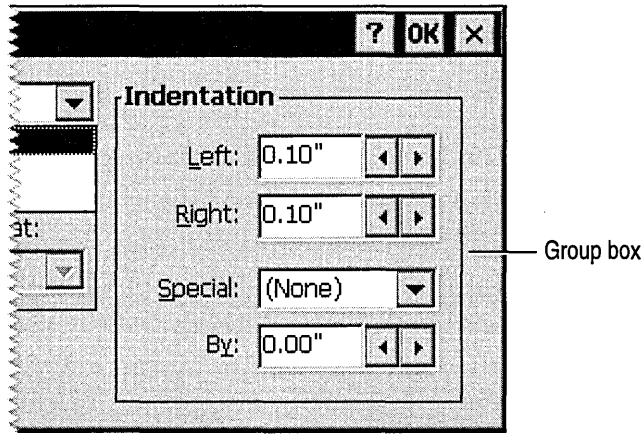
```
RADIOBUTTON "string", id, x, y, width, height [[, style [[, extended-
style]]]]
```

Here, *string* is the text you want displayed inside the radio button, and *id* is the value that identifies the radio button. The upper-left corner of the control is positioned at *x*, *y*, and its dimension is determined by *width* and *height*. *Style* and *extended-style* determine the appearance of the radio button. The **RADIOBUTTON** resource statement creates a manual radio button. This means that your application must manually clear other radio buttons in the group each time a user selects a button. If you want Windows CE to automatically clear other radio buttons when a user selects an option, use the **AUTORADIOBUTTON** resource statement.

A *group box* is a rectangular area within a dialog box in which you can group together controls that are semantically related. Controls are grouped by drawing a rectangular border around them. Any text associated with the group box is displayed in its upper-left corner. The sole purpose of a group box is to organize controls related by a common purpose, usually indicated by the label. The group box has only one style, defined by the constant BS_GROUPBOX. Because a group box cannot be selected, it has no check state, focus state, or push state. An application cannot send messages to a group box.

Because group boxes are opaque in Windows CE, add them to your dialog box template after you add other elements. Anything you add to the template after you add the group box will be hidden underneath it. By adding group boxes last, you ensure that the group boxes are at the bottom of the z-order and will not hide your other controls. The z-order is a stack of overlapping windows.

The following screen shot shows a group box.



▶ **To create a group box using CreateWindow**

1. Specify the BUTTON window class in the *lpClassName* parameter of the **CreateWindow** or **CreateWindowEx** function.
2. Specify one or more group box styles in the *dwStyle* parameter of the **CreateWindow** or **CreateWindowEx** function.

▶ **To create a group box in a dialog box**

- Add the following **GROUPBOX** resource-definition statement to your **DIALOG** resource.

```
GROUPBOX "title", id, x, y, width, height [[, style [[, extended-
style]]]]
```

Here, *title* is the title of the box, and *id* is the value that identifies the group box. The upper-left corner of the control is positioned at *x*, *y*, and its dimension is determined by *width* and *height*. *Style* and *extended-style* determine the appearance of the group box.

When a user selects a button, either the operating system or the application must change one or more of the button's state elements. A button's state can be characterized by its focus state, push state, and check state. Windows CE automatically changes the focus state for all button types, the push state for push buttons, and the check state for all automatic buttons. The application must make all other state changes, taking into account the button's type, style, and current state. An application can determine a button's state by sending it a BM_GETCHECK or BM_GETSTATE message; the application can set a button's state by sending it a BM_SETCHECK or BM_SETSTATE message.

Handling Button Messages

When a user selects a button, its state changes, and the button sends notification messages to its parent window about the changed state. For example, a push button control sends the BN_CLICKED notification message when a user selects the button. In all cases, the low-order word of *wParam* contains the control identifier, the high-order word of *wParam* contains the notification code, and *lParam* contains the control window handle. Both the message and the parent window's response to it depend on the type, style, and current button state.

For automatic buttons, the OS handles all state changes and the application processes only the BN_CLICKED notification message. For buttons that are not automatic, the application usually responds to the notification message by sending a message to change the state of the button. When a user selects an owner-drawn button, the button sends its parent window a WM_DRAWITEM message containing the identifier of the control to be drawn and information about its dimensions and state.

A button can also receive messages. A parent window can send messages to a button in an overlapped or child window by using the **SendMessage** function. It can send messages to a button in a dialog box by using the **SendDlgItemMessage** and **CheckRadioButton** functions.

Windows also provides default color values for buttons. The system sends a WM_CTLCOLORBTN message to a button's parent window before the button is drawn. This message contains a handle of the button's device context and a handle of the child window. The parent window can use these handles to change the button's text and background colors. An application can retrieve the default values for these colors by calling the **GetSysColor** function, or it can set the values by calling the **SetSysColors** function.

The window procedure for the predefined button control window class processes defaults for all messages that the button control procedure does not process. When the button control procedure returns FALSE for any message, the predefined window procedure checks the messages.

Creating an Edit Control

An *edit control*, also called a text box, is a rectangular window in which a user can enter and edit text. Generally, you provide a label for an edit control by placing a static control with the appropriate text above or next to the edit control. However, if you do not have enough space to do this, you can enclose the label within angle brackets—for example, <edit control label>—and include the enclosed label as the default text inside the edit control.

▶ To create an edit control using CreateWindow

1. Specify the EDIT window class in the *lpClassName* parameter of the **CreateWindow** or **CreateWindowEx** function.
2. Specify one or more edit control styles in the *dwStyle* parameter of the **CreateWindow** or **CreateWindowEx** function.

   The edit control style values you select can establish the appearance of a single-line or multiline edit control, align the text in the control, and determine if and how text appears in the edit control. The number and type of styles the application uses depend on the type and purpose of the edit control.

The following code example shows how to use **CreateWindow** to create an edit control.

```
#define EDITID 1

// Specify the edit control window style.
DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
                WS_BORDER | ES_LEFT | ES_MULTILINE | ES_NOHIDESEL |
                ES_AUTOHSCROLL | ES_AUTOVSCROLL;

// Create the edit control window.
g_hwndEdit = CreateWindow (
                TEXT("edit"),   // Class name
                NULL,           // Window text
                dwStyle,        // Window style
                0,              // x coordinate of the upper-left corner
                0,              // y coordinate of the upper-left corner
                CW_USEDEFAULT,  // Width of the edit control window
                CW_USEDEFAULT,  // Height of the edit control window
                hwnd,           // Window handle of parent window
                (HMENU) EDITID, // Control identifier
                g_hInst,        // Instance handle
                NULL);          // Specify NULL for this parameter when
                                // creating a control
```

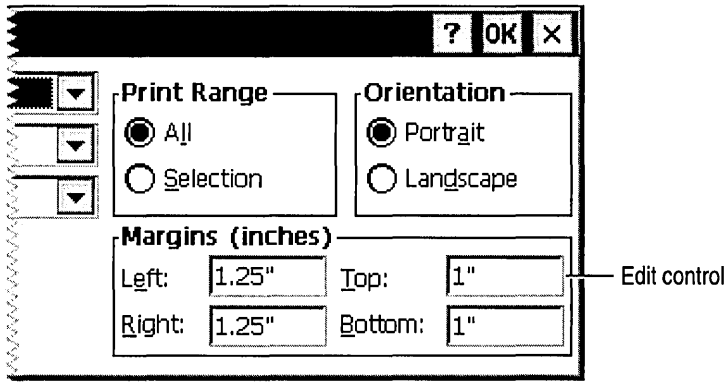
▶ To create an edit control in a dialog box

- Add the following **EDITTEXT** resource-definition statement to your **DIALOG** resource.

  ```
  EDITTEXT id, x, y, width, height [[, style [[, extended-style]]]]
  ```

  Here, *id* is the value that identifies the edit box. The upper-left corner of the control is positioned at *x*, *y*, and its dimension is determined by *width* and *height*. *Style* and *extended-style* determine the appearance of the edit box.

The following screen shot shows an edit control.



Modifying the Text Buffer

When Windows CE creates an edit control, it automatically creates a text buffer, sets its initial size, and increases the size as necessary. Windows CE stores edit control text in a text buffer and copies it to the control. The size can be up to a predefined limit of 30,000 characters for single-line edit controls. Because this limit can change, it is a soft limit. You can set a hard limit to the buffer size by sending an EM_SETLIMITTEXT message to the edit control. If the buffer exceeds either limit, Windows CE sends the application an EN_ERRSPACE message. You can retrieve the current text limit by sending an EM_GETLIMITTEXT message.

You free the buffer by calling the **LocalFree** function, or you can obtain a new buffer, and buffer handle, by calling the **LocalAlloc** function.

You can initialize or reinitialize an edit control's text buffer by calling the **SetDlgItemText** function. It can retrieve the content of a text buffer by calling the **GetDlgItemText** function.

For each edit control, Windows CE maintains a read-only flag that indicates whether the control's text is read/write, which is the default, or read-only. An application can set the read/write or read-only flag for the text by sending the control an EM_SETREADONLY message. To determine whether an edit control is read-only, an application can call the **GetWindowLong** function using the GWL_STYLE constant. The EM_SETREADONLY message applies to both single-line and multiline edit controls.

You can change the font that an edit control uses by sending the WM_SETFONT message. Changing the font does not change the size of the edit control; applications that send the WM_SETFONT message might have to retrieve the font metrics for the text and recalculate the size of the edit control.

Changing the Formatting Rectangle

The visibility of an edit control's text is governed by the dimensions of its window rectangle and its formatting rectangle. The window rectangle is the client area of the window containing the edit control. The formatting rectangle is a construct maintained by Windows CE for formatting the text displayed in the window rectangle. When an edit control is first displayed, the two rectangles are identical on the screen. An application can make the formatting rectangle larger or smaller than the window rectangle. Making the formatting rectangle larger limits the visibility of the edit control's text; making it smaller creates extra white space around the text.

You can set the coordinates of an edit control's formatting rectangle by sending it an EM_SETRECT message. The EM_SETRECT message automatically redraws the edit control's text. To establish the coordinates of the formatting rectangle without redrawing the control's text, send the control an EM_SETRECTNP message. To retrieve the coordinates of the formatting rectangle, send the control an EM_GETRECT message. These messages apply to multiline edit controls only.

Working with Text

After selecting an edit control, a user can select text in the control by using a pointing device or keyboard keys. You can retrieve the starting and ending character positions of the current selection in an edit control by sending the control an EM_GETSEL message.

You can also select text in an edit control by sending the control an EM_SETSEL message with the starting and ending character indexes for the selection. For example, you can use EM_SETSEL with EM_REPLACESEL to delete text from an edit control. These three messages apply to both single-line and multiline edit controls.

You can replace selected text in an edit control by sending the control an EM_REPLACESEL message with a pointer to the replacement text. If there is no current selection, EM_REPLACESEL inserts the replacement text at the insertion point. You might receive an EN_ERRSPACE notification message if the replacement text exceeds the available memory. This message applies to both single-line and multiline edit controls. You can use EM_REPLACESEL to replace part of an edit control's text or the **SetDlgItemText** function to replace all of it.

Manipulating Text

Windows CE provides four messages for moving text between an edit control and the Clipboard. These four messages apply to both single-line and multiline edit controls.

The following table describes these messages.

Message	Description
WM_COPY	Copies the current selection, if any, from an edit control to the Clipboard without deleting it from the edit control
WM_CUT	Deletes the current selection, if any, in the edit control and copies the deleted text to the Clipboard
WM_CLEAR	Deletes the current selection, if any, from an edit control, but does not copy it to the Clipboard unless a user presses the SHIFT key
WM_PASTE	Copies text from the Clipboard into an edit control at the insertion point

When a user selects, deletes, or moves text in an edit control, Windows CE maintains an internal flag for each edit control indicating whether the content of the control has been modified. Windows CE clears this flag when it creates the control and sets the flag when the text in the control is modified. You can retrieve the modification flag by sending the control an EM_GETMODIFY message and set or clear the modification flag by sending the control an EM_SETMODIFY message. These messages apply to both single-line and multiline edit controls.

The default limit of text that a user can type in an edit control is 30,000 characters. You can change the amount of text a user can type by sending the control an EM_SETLIMITTEXT message. This message sets a hard limit to the number of bytes a user can type into an edit control, but affects neither text already in the control when the message is sent nor text copied to the control by the **SetDlgItemText** function or the WM_SETTEXT message. For example, suppose that you use the **SetDlgItemText** function to place 500 characters in an edit control, and a user also types 500 characters in the edit control, creating a total of 1,000 characters. If you send an EM_SETLIMITTEXT message limiting user-entered text to 300 characters, the 1,000 characters already in the edit control remain there, and a user cannot add any more text. On the other hand, if you send an EM_SETLIMITTEXT message limiting user-entered text to 1,300 characters, the 1,000 characters remain, but a user can add 300 more characters.

When a user reaches the character limit of an edit control, Windows CE sends the application a WM_COMMAND message containing an EN_MAXTEXT notification message. This notification message does not mean that memory has been exhausted, but that the limit for user-entered text has been reached; a user cannot type any more text. To change this limit, you must send the control a new EM_SETLIMITTEXT message with a higher limit.

Working with Wordwrap Functions

You can direct a multiline edit control to add or remove a soft line break character—two carriage returns and a linefeed—automatically at the end of wrapped text lines. An application can turn this feature on or off by sending the edit control an EM_FMTLINES message. This message applies only to multiline edit controls and does not affect a line that ends with a hard line break—one carriage return and a linefeed typed by a user.

Retrieving Points and Characters

To determine which character is closest to the specified point in an edit control, send the EM_CHARFROMPOS message. The message returns the character index and line index of the character nearest the point. Similarly, you can determine the client coordinates of the specified character in an edit control by sending the EM_POSFROMCHAR message. You specify the index of a character and the message returns the x-coordinate and y-coordinate of the upper-left corner of the character.

Undoing Text Operations

Every edit control maintains an undo flag that indicates whether an application can reverse the most recent operation, such as a text deletion, on the control. The edit control sets the undo flag to indicate that the operation can be undone and resets it to indicate that the operation cannot be undone. You can determine the setting of the undo flag by sending the control an EM_CANUNDO message.

To undo the most recent operation, send the control an EM_UNDO message. An operation can be undone provided that no other edit control operation occurs first. For example, a user can delete text, replace the text or undo the deletion, and then delete the text again or undo the replacement. The EM_UNDO message applies to both single-line and multiline edit controls and always works for single-line edit controls.

Scrolling Text in an Edit Control

To implement scrolling in an edit control, you can use the automatic scrolling styles, or you can explicitly add scroll bars to the edit control. To add a horizontal scroll bar, use the style WS_HSCROLL; to add a vertical scroll bar, use the style WS_VSCROLL. An edit control with scroll bars processes its own scroll bar messages.

Windows CE provides three messages that you can send to an edit control with scroll bars. The EM_LINESCROLL message can scroll a multiline edit control both vertically and horizontally. The *lParam* parameter specifies the number of lines to scroll vertically starting from the current line and the *wParam* parameter specifies the number of characters to scroll horizontally, starting from the current character. The edit control does not acknowledge messages to scroll horizontally if it has the ES_CENTER or ES_RIGHT style. This message applies to multiline edit controls only.

The EM_SCROLL message scrolls a multiline edit control vertically, which is the same effect as sending a WM_VSCROLL message. The *wParam* parameter specifies the scrolling action. The EM_SCROLL message applies to multiline edit controls only.

Adding Tab Stops and Margins

To set tab stops in a multiline edit control, use the EM_SETTABSTOPS message. The default for a tab stop is eight characters. When you add text to the edit control, tab characters in the text automatically generate space up to the next tab stop. The EM_SETTABSTOPS message does not automatically cause Windows CE to redraw the text. To do that, you can call the **InvalidateRect** function. The EM_SETTABSTOPS message applies to multiline edit controls only.

To set the width of the left and right margins for an edit control, use the EM_SETMARGINS message. After sending this message, Windows CE redraws the edit control to reflect the new margin settings. You can retrieve the width of the left or right margin by sending the EM_GETMARGINS message. By default, the edit control margins are set to be just wide enough to accommodate the largest character horizontal overhang, known as a negative ABC width, for the font currently in use in the edit control.

Using Password Characters

You can use a password character in an edit control to conceal user input. When a password character is set, it is displayed in place of each character typed. When a password character is removed, the control displays the characters that a user types. If you create an edit control using the style ES_PASSWORD, the default password character is an asterisk. An application can use the EM_SETPASSWORDCHAR message to remove or define a different password character and the EM_GETPASSWORDCHAR message to retrieve the current password character. These messages apply to single-line edit controls only.

Creating a List Box

A *list box* is a window that displays a list of character strings. A user selects a string from the list by tapping it with the stylus. When a string is selected, it is highlighted. List boxes are typically placed in dialog boxes, but they can also be placed in a normal window by specifying the LISTBOX window class in the *lpClassName* parameter of the **CreateWindow** or **CreateWindowEx** function.

▶ **To create a list box control in a dialog box**

- Add the following **LISTBOX** resource-definition statement to the **DIALOG** resource.

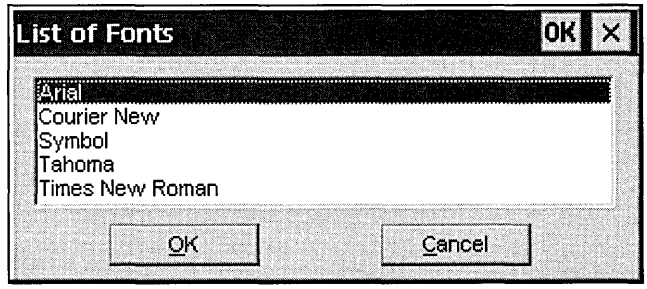
```
LISTBOX id, x, y, width, height [[, style [[, extended-style]]]]
```

Here, *id* is the value that identifies the list box. The upper-left corner of the control is positioned at *x*, *y*, and its dimension is determined by *width* and *height*. *Style* and *extended-style* determine the appearance of the list box. The Win32 API provides two types of list boxes: single-selection, which is the default, and multiple-selection. In a single-selection list box, a user can select only one item at a time. In a multiple-selection list box, a user can select more than one item at a time. To create a multiple-selection list box, specify the LBS_MULTIPLESEL or the LBS_EXTENDEDSEL style.

Note Windows CE supports the LBS_EX_CONSTSTRINGDATA style, which saves RAM when a large table of strings in ROM is inserted into a list box.

All list boxes in Windows CE have the LBS_HASSTRINGS style by default. Windows CE does not support owner-drawn list boxes.

The following screen shot shows a list box.



Because list boxes are empty by default, you must initialize or populate the list box with data when the dialog box containing the list box is displayed. Each time a dialog box is activated, the system sends your dialog box procedure a WM_INITDIALOG message. A dialog box procedure is responsible for initializing and monitoring its child windows, including any list boxes. The dialog box procedure communicates with the list box by sending messages to it and by processing the notification messages sent by the list box. To initialize the list box, modify your dialog box procedure by adding a case to the **SWITCH** statement in you dialog box procedure. The following code example shows how to do this.

```
#define IDC_FONTLIST  1000          // List control identifier

int iNumOfFonts = 0;                // The total number of fonts
int iCurrItem = -1;                 // The index of newly selected or
                                    // added font name in the list box
BOOL CALLBACK FontListDlgProc (HWND hwndDlg, UINT uMsg, WPARAM wParam,
                               LPARAM lParam)
{
  int index = 1,
      iFontIndex;           // Font list index
  TCHAR szFontName[80],     // For the currently enumerated font name
        szFontNamePrev[80]; // For the previously enumerated font name
  HWND hwndFontListCtrl;    // Window handle of the font list box

  // Get the window handle of the font list box control in the dialog
  // box.
  hwndFontListCtrl = GetDlgItem (hwndDlg, IDC_FONTLIST);

  switch (uMsg)
  {
    case WM_INITDIALOG:
      // Need to find the total number of fonts, iNumOfFonts, prior to
      // the following code.

      for (iFontIndex = 0; iFontIndex < iNumOfFonts; ++iFontIndex)
      {
        // Insert code here to retrieve the name of each font and then
        // put it in szFontName.
        // ...

        // Add the font name to the list control.
        iCurrItem = SendMessage (hwndFontListCtrl, LB_ADDSTRING, 0,
                                 (LPARAM)(LPCTSTR) szFontName);

        // Set a 32-bit value, (LPARAM) iFontIndex, associated with the
        // newly added item in the list control.
        SendMessage (hwndFontListCtrl, LB_SETITEMDATA,
                     (WPARAM) iCurrItem, (LPARAM) iFontIndex);
      }
```

```
        // Select the first font name in the list control.
        SendMessage (hwndFontListCtrl, LB_SETCURSEL, 0, 0);

        return TRUE;

      case WM_COMMAND:
        switch (LOWORD (wParam))
        {
          case IDOK:
            // Retrieve the index of the currently selected font.
            if ((iCurrItem = SendMessage (hwndFontListCtrl, LB_GETCURSEL,
                                          0, 0)) != LB_ERR)
            {
              iFontIndex = SendMessage (hwndFontListCtrl, LB_GETITEMDATA,
                                        iCurrItem, 0);

              // Get the currently selected font from the index.
              CurrLogFont = g_lpEnumLogFont[iFontIndex].elfLogFont;

              EndDialog (hwndDlg, 0);
              return TRUE;
            }

          case IDCANCEL:
            iCurrItem = -1;
            EndDialog (hwndDlg, 0);
            return TRUE;
        }
        break;
    }
    return FALSE;
}
```

Creating a Combo Box

A *combo box* is a control that combines a list box with an edit control. Selecting an item in the list box displays the selected text in the edit control. If the combo box style accepts keyboard input, typing characters into the edit control highlights the first list box item that matches the characters typed. A combo box can appear either in a dialog box or on the command bar.

▶ **To create a combo control in a dialog box**

- Add the following **COMBOBOX** resource-definition statement to your **DIALOG** resource.

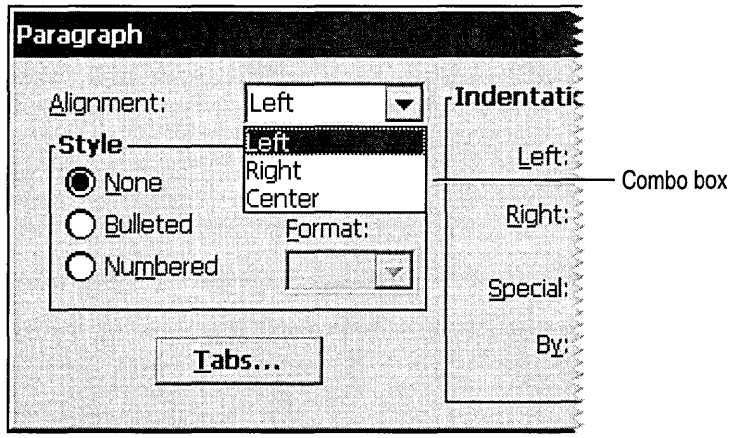
```
COMBOBOX id, x, y, width, height [[, style [[, extended-style]]]]
```

Here, *id* is the value that identifies the combo box. The upper-left corner of the control is positioned at *x*, *y*, and its dimension is determined by *width* and *height*. *Style* and *extended-style* determine the appearance of the combo box. Because of limited screen space, Windows CE–based devices use either the CBS_DROPDOWN or CBS_DROPDOWNLIST style rather than the CBS_SIMPLE style popular on Windows-based desktop platforms. In the CBS_SIMPLE style, the list box is always visible and the current selection is displayed in the edit control. In the CBS_DROPDOWN or CBS_DROPDOWNLIST styles, the list box is not displayed until a user selects an icon next to the edit control, which conserves screen space. The difference between the two styles is that the CBS_DROPDOWNLIST style has a static text field that always displays the current selection instead of having an edit control.

Note If you specify the CBS_EX_CONSTSTRINGDATA style when the application inserts a string into the list part of a combo box, the combo box stores the pointer passed to it by the application rather than copying the string. This saves RAM when you have a large table of strings in ROM that you want to insert into a combo box.

All list boxes in Windows CE have the LBS_HASSTRINGS style by default. Windows CE does not support owner-drawn combo boxes.

To create a command bar combo box and insert it into a command bar, use the **CommandBar_InsertComboBox** function. The following screen shot shows a combo box.



Working with Edit Control Selection Fields

The edit control selection field is the portion of a combo box that displays the currently selected list item. In drop-down combo boxes, which are combo boxes that have the CBS_DROPDOWN style, the selection field is an edit control and can be used to type text not in the list.

You can retrieve or set the contents of the edit control selection field and can determine or set the edit selection. You can also limit the amount of text a user can type in the selection field. When the contents of the selection field change, Windows CE sends notification messages to the parent window or dialog box procedure.

To retrieve the content of the edit control selection field, you send a WM_GETTEXT message to the combo box. To set the contents of the selection field of a drop-down combo box, you send the WM_SETTEXT message to the combo box.

Creating a Scroll Bar

A *scroll bar* is used to scroll text in a window. Scroll bars should be included in any window for which the content of the client area extends beyond the window borders. The orientation of a scroll bar determines the direction in which scrolling occurs when a user operates the scroll bar. A horizontal scroll bar enables a user to scroll the content of a window to the left or right. A vertical scroll bar enables a user to scroll the content up or down.

You can use as many scroll bar controls as needed in a single window. When you create a scroll bar control, you must specify the size and position of the scroll bar. However, if a scroll bar control's window can be resized, your application must adjust the size of the scroll bar when the size of the window changes.

▶ **To create a scroll bar using CreateWindow**

1. Specify the SCROLLBAR window class in the *lpClassName* parameter of the **CreateWindow** or **CreateWindowEx** function.
2. Specify one or more scroll bar control styles in the *dwStyle* parameter of the **CreateWindow** or **CreateWindowEx** function.

   A scroll bar control can have a number of styles to control the orientation and position of the scroll bar. Some of the styles create a scroll bar control that uses a default width or height. However, you must always specify the *x* and *y* coordinates and the other scroll bar dimensions.

The following code example shows how to use **CreateWindow** to create a scroll bar.

```
#define SCROLLBARID 100

DWORD dwStyle = SBS_BOTTOMALIGN | SBS_HORZ | WS_VISIBLE | WS_CHILD;

hwndSB = CreateWindow (
            TEXT("scrollbar"),  // Class name
            NULL,               // Window text
            dwStyle,            // Window style
            0,                  // x coordinate of the upper-left corner
            0,                  // y coordinate of the upper-left corner
            CW_USEDEFAULT,      // The width of the edit control window
            CW_USEDEFAULT,      // The height of the edit control window
            hwnd,               // Window handle of parent window
            (HMENU) SCROLLBARID,// The control identifier
            hInst,              // The instance handle
            NULL);              // Specify NULL for this parameter when
                                // creating a control
```

▶ To create a scroll bar control in a dialog box

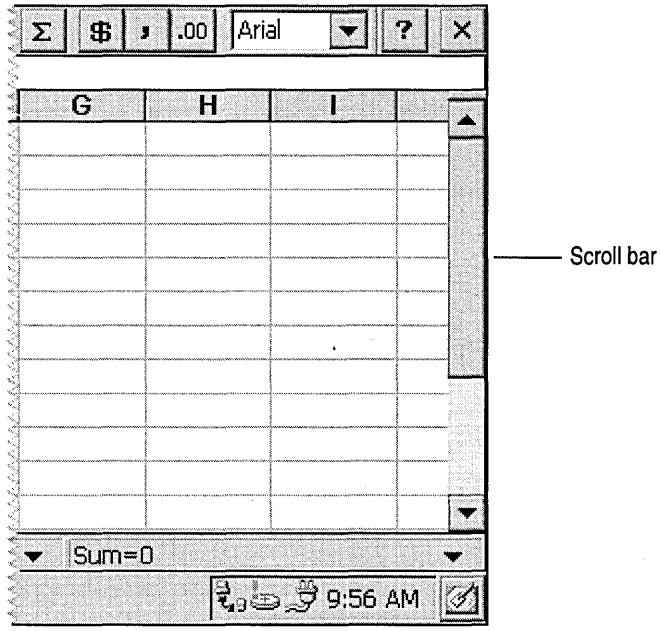
- Add the following **SCROLLBAR** resource-definition statement to your **DIALOG** resource.

```
SCROLLBAR id, x, y, width, height [[, style [[, extended-style]]]]
```

Here, *id* is the value that identifies the scroll bar. The *x* and *y* parameters determine the scroll bar position and are represented as integers. They are relative to the left or upper end of the scroll bar, depending on whether the scroll bar is horizontal or vertical. The position must be within the minimum and maximum values of the scrolling range. For example, in a scroll bar with a range from 0 through 100, position 50 is the middle, with the remaining positions distributed equally along the scroll bar. The initial range depends on the scroll bar. Standard scroll bars have an initial range from 0 through 100. Scroll bar controls have an empty range—both minimum and maximum values are zero—unless you supply an explicit range when you create the control. You can alter the range at any time after its initial creation. You can use the **SetScrollInfo** function to set the range values and the **GetScrollInfo** function to retrieve the current range values.

The *width* and *height* parameters determine the scrollbar size. You can set a scroll bar equal to a page size. The page size represents the number of data units that can fit in the client area of the owner window given its current size. For example, if the client area can hold eight lines of text, an application would set the page size to eight. Windows CE uses the page size, along with the scrolling range and length of the scroll bar's gray area, to set the size of the scroll bar. When a window containing a scroll bar is resized, an application should call the **SetScrollInfo** function to set the page size. An application can retrieve the current page size by calling the **GetScrollInfo** function.

Style and *extended-style* determine the appearance of the edit box. The default style of a scroll bar is SBS_HORZ, which creates a horizontal scroll bar. The following screen shot shows a horizontal scroll bar. For a vertical scroll bar, specify the SBS_VERT style.



To establish a useful relationship between the scroll bar range and the data object, your application must adjust the range when the size of the data object changes.

As a user moves the scroll box in a scroll bar, the scroll bar reports the scroll box position as an integer in the scrolling range. If the position is the minimum value, the scroll box is at the top of a vertical scroll bar or at the left of a horizontal scroll bar. If the position is the maximum value, the scroll box is at the bottom of a vertical scroll bar or at the right end of a horizontal scroll bar.

Your application must move the scroll box in a scroll bar. Although a user makes a request for scrolling in a scroll bar, the scroll bar does not automatically update the scroll box position. Rather, it passes the request to the parent window, which must scroll the data and update the scroll box position. Use the **SetScrollInfo** function in your application to update the scroll box position. Because your application controls the scroll box movement relative to the window data object, you determine the incremental position settings for the scroll box that work best for the data being scrolled.

Handling Scroll Bar Requests

Unlike other controls, a scroll bar does not send WM_COMMAND messages. Rather, the scroll bar sends WM_HSCROLL and WM_VSCROLL messages to the window procedure when a user taps the scroll bar or drags the scroll box. The low-order words of WM_VSCROLL and WM_HSCROLL each contain a notification message that indicates the direction and magnitude of the scrolling action.

When you process the WM_HSCROLL and WM_VSCROLL messages, you should examine the scroll bar notification message and calculate the scrolling increment. After you apply the increment to the current scrolling position, you can scroll the window to the new position by using the **ScrollWindowEx** function. You can use the **SetScrollInfo** function to adjust the position of the scroll box.

The corresponding SBM_SETSCROLLINFO message passes parameters into the **SCROLLINFO** structure to indicate the new scroll box position. The SBM_GETSCROLLINFO message retrieves the current position contained within the **SCROLLINFO** structure.

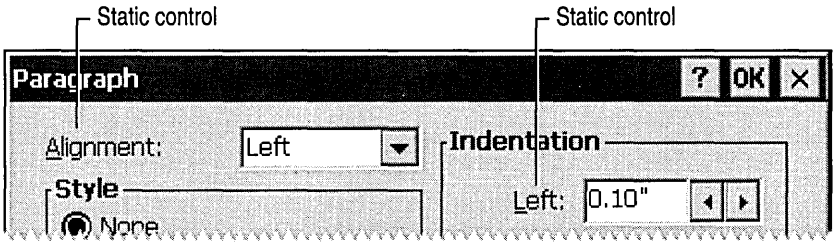
After you scroll a window, it makes part of the window's client area invalid. To ensure that the invalid region is updated, use the **UpdateWindow** function to generate a WM_PAINT message.

Usually an application scrolls the content of a window in the direction opposite to that indicated by the scroll bar. For example, when a user taps the gray area below the scroll box, an application scrolls the object in the window upward to reveal a portion of the object below the visible portion. An application can also scroll a rectangular region using the **ScrollDC** function.

When you process the WM_CREATE message you can set scrolling units. It is convenient to base the scrolling units on the dimensions of the font associated with the window display context. To retrieve the font dimensions for a specific display context, use the **GetTextMetrics** function. When you process the WM_SIZE message, you can adjust the scrolling range and scrolling position to reflect the dimensions of the client area as well as the number of lines of text displayed.

Creating a Static Control

A *static control* is 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, but it can notify its parent window of a stylus tap if the static control is created with **SS_NOTIFY** style. The following screen shot shows a static control.



Although you can use static controls in overlapped, pop-up, and child windows, they are designed for use in dialog boxes, where Windows CE standardizes their behavior. If you use static controls outside of dialog boxes, you increase the risk that the application might behave in a nonstandard fashion. Windows CE does not support owner-drawn static controls.

Choosing a Control Style

In Windows CE, you can use only the SS_CENTERIMAGE style in conjunction with the SS_BITMAP style.

The following code shows how to use the SS_Bitmap style.

```
SendMessage( hStatic, STM_SETIMAGE, IMAGE_BITMAP, (LPARAM) hBitmap );
```

If you specify SS_CENTERIMAGE and do not specify either SS_ICON or SS_BITMAP, the static control will behave as though you had specified the SS_BITMAP style.

Windows CE does not support the SS_SIMPLE static control styles, but you can emulate this style by using the SS_LEFT or SS_LEFTNOWORDWRAP style. Windows CE also does not support the SS_BLACKFRAME, SS_BLACKRECT, SS_GRAYFRAME, SS_GRAYRECT, SS_OWNERDRAW, SS_WHITEFRAME, or SS_WHITERECT styles, but you can use the WM_PAINT message to achieve the same results.

Creating an Application-Defined Window Control

You can create custom, application-defined window controls to perform tasks not supported by predefined controls. Windows CE provides the following ways to create a custom control:

- Use owner-drawn buttons.
- Use the subclass procedure to produce a custom control.
- Register and implement an application-defined window class.

Buttons have owner-drawn styles available that direct the control to send a message to the parent window when the control must be drawn. This feature enables you to alter the appearance of a control. For buttons, the owner-drawn style affects how the system draws the entire control.

You can designate buttons as owner-drawn controls by creating them with the appropriate style. When a control has the owner-drawn style, Windows CE handles a user's interaction with the control as usual, performing such tasks as detecting when a user has chosen a button and then notifying the button's owner of the event. However, because the control is owner-drawn, the parent window of the control is responsible for the visual appearance of the control.

When you use the BS_OWNERDRAW style for a button, you assume all responsibility for drawing the button. You cannot use any other button styles with the BS_OWNERDRAW style. When you use an owner-drawn button, you have to trap the WM_DRAWITEM message in the window procedure for the button's parent window and insert the code that erases the background, if necessary, and draws the button.

The WM_DRAWITEM message is not generated by the window manager; it is part of the interface between a button and its owner. When you use a built-in button class, the button's window procedure automatically sends the WM_DRAWITEM message to the button's parent window when the button receives a WM_PAINT message. If you create a new class of button—a button that is not a built-in button—and you want it to support the WM_DRAWITEM message, you must send the WM_DRAWITEM message to the button's parent window when the button needs to be redrawn.

You can use the subclass procedure to create a custom control. The subclass procedure alters selected behaviors of the control by processing those messages that affect the selected behaviors. All other messages pass to the original window procedure for the control.

You can create custom controls by registering an application-defined window class and specifying the name of the window class in the **CreateWindowEx** function or in the dialog box template. The process for registering an application-defined window class for a custom control is the same as for registering a class for an ordinary window. Each class must have a unique name, a corresponding window procedure, and other information.

At a minimum, the window procedure draws the control. If an application uses the control to let a user type information, the window procedure also processes input messages from the keyboard and stylus and sends notification messages to the parent window. In addition, if the control supports control messages, the window procedure processes messages sent to it by the parent window or other windows. For example, controls often process the WM_GETDLGCODE message sent by dialog boxes to direct a dialog box to process keyboard input in a specified way.

Because an application-defined control message is specific to the designated control, you must explicitly send it to the control by using the **SendMessage** or **SendDlgItemMessage** function. The numeric value for each message must be unique and must not conflict with the values of other window messages.

Working with Common Controls

Common controls are a set of windows supported by the common control library, which is a dynamic-link library (DLL) included with the Windows CE OS. Like other control windows, a common control is a child window that an application uses in conjunction with another window to perform I/O tasks.

Common controls offer users a familiar interface for performing common tasks, which makes applications easier to use and learn. Most common controls send the WM_NOTIFY message instead of the WM_COMMAND message sent by window controls.

The following list shows common controls supported by Windows CE.

Command bars	Tree views
Command bands	Up-down controls
Rebars	Date and time picker
Toolbars	Month calendar controls
ToolTips	Status bars
Header controls	Progress bars
Image lists	Property sheets
List views	Tab controls
Trackbars	

Windows CE does not support the following controls commonly used on Windows-based desktop platforms: animation controls, **ComboBoxEx** controls, drag lists, flat scroll bars, hot keys, Internet Protocol (IP) address controls, or Rich Ink edit controls.

Before you can create or use any common controls, you must register them. You can do this in either of two ways: call the **InitCommonControls** function, which registers all the common controls at once except for the rebar, month calendar, and date and time picker controls, or call the **InitCommonControlsEx** function, which registers a specific common control class. Calling either of these functions ensures that the common DLL is loaded.

To use most of the common controls, you must include the Commctrl.h header file in your application. To use property sheets, you must include the Prsht.h header file.

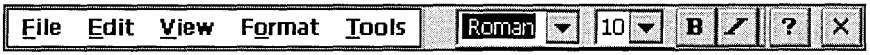
When creating a common control, it is important to understand that all common controls are child windows that you create by calling **CreateWindowEx**. You can also create a common control by calling a control-specific API function. Because common controls are windows, you can manage them the same way that you manage other application windows.

Though Windows CE supports some styles that apply to a broad spectrum of common controls, each of the common controls also has a set of styles unique to that control. Unless otherwise noted, these unique styles apply to header controls, toolbar controls, rebars, and status windows.

Creating a Command Bar

A command bar is a toolbar that can include a menu bar as well as the **Close** (**X**) button, the **Help** (**?**) button, and the **OK** button, and is usually found on the title bar of Windows-based desktop applications. A command bar can contain menus, combo boxes, buttons, and *separators*. A separator is a blank space you can use to divide other elements into groups or to reserve space in a command bar. You create a command bar to organize your application menus and buttons.

You create a command bar by using the **CommandBarCreate** function. Windows CE registers this class when it loads the common control DLL. You can use the **InitCommonControls** function to ensure that this DLL is loaded. The following screen shot shows a Windows CE command bar.



▶ **To create a command bar**

1. Create the command bands control by calling the **CommandBar_Create** function.
2. Add controls to the command bar by calling the **CommandBar_InsertMenubar**, **CommandBar_AddBitmap**, **CommandBar_AddButtons**, and **CommandBar_InsertComboBox** functions.
3. Add the **Close** and **Help** buttons by calling the **CommandBar_AddAdornments** function and passing CMDBAR_HELP in the *dwFlags* parameter. Windows CE automatically adds the **Close** button.

In addition to creating and registering command bars, Windows CE supports many functions you can use to manipulate a command bar.

The following table shows how to manipulate a command bar control.

To	Call
Add the **Close** (**X**), **Help** (**?**), and **OK** buttons to the command bar. Minimally, every command bar must have a **Close** button.	**CommandBar_AddAdornments**
Destroy the command bar without destroying the parent window.	**CommandBar_Destroy**
Add a single button or separator to a command bar.	**CommandBar_InsertButton**

To	Call
Add several buttons or separators at once to a command bar. When creating a separator, specify TBSTYLE_SEP as the *fsStyle* member of the **TBBUTTON** structure you pass in the *lpButton* parameter.	**CommandBar_AddButtons**
Determine the usable portion of the application window.	**GetClientRect**
Subtract the height of the command bar from the size of the client rectangle.	**CommandBar_Height**
Determine whether a command bands control is visible.	**CommandBands_GetRestoreInformation**
Add ToolTips describing the command bar buttons.	**CommandBar_AddTooltips**
Show or hide the command bands control.	**CommandBands_Show**
Determine if a command bar is visible.	**CommandBar_IsVisible**
Create a combo box and insert it into a command bar. This function always creates a combo box with the WS_CHILD and WS_VISIBLE styles.	**CommandBar_InsertComboBox**
Insert a menubar, identified by a resource identifier, into a command bar.	**CommandBar_InsertMenubar**
Insert a menubar, identified by a resource name or menu handle, into a command bar.	**CommandBar_InsertMenubarEx**
Obtain the handle of a menu bar in a command bar.	**CommandBar_GetMenu**
Obtain the handle of a submenu on the menu bar.	**GetSubMenu**
Redraw the command bar after modifying a menu bar on the command bar.	**CommandBar_DrawMenuBar**

Each element in a command bar has a zero-based index by which command bar functions can identify it. The leftmost element has an index of zero, the element immediately to its right has an index of one, and so on. When you use any of the **CommandBar_Insert** functions, the menu bar, button, or combo box is inserted to the left of the button whose index you specify in the *iButton* parameter.

A command bar stores the information needed to draw the button images in an internal list, which is empty when the command bar is created. Each image has a zero-based index that you use to associate the image with a button. Use the **CommandBar_AddBitmap** function to add an array of images to the end of the list. This function returns the index of the first new image added. The system includes a set of predefined command bar buttons with header files that define constant values for their indexes.

Note Do not use 0xFFFFFFFF as the command identifier of a command bar control. This identifier is reserved for use by the command bar.

The following code example shows how to create a command bar.

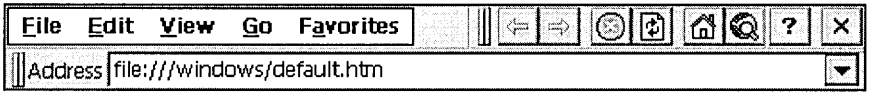
```
// Create a command bar.
hwndCB = CommandBar_Create (hInst, hwnd, 1);

// Adds ToolTip strings to the command bar.
CommandBar_AddToolTips (hwndCB, uNumSmallTips, szSmallTips);

// Adds 15 images to the list of button images available for use
// in the command bar.
CommandBar_AddBitmap (hwndCB, HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
                      15, 16, 16);

// Insert the menu bar into the command bar.
CommandBar_InsertMenubar (hwndCB, hInst, IDM_MAIN_MENU, 0);

// Add buttons in tbSTDButton to the command bar.
CommandBar_AddButtons (hwndCB,
                       sizeof (tbSTDButton) / sizeof (TBBUTTON),
                       tbSTDButton);

// Add Help, OK, and Exit buttons to the command bar.
CommandBar_AddAdornments (hwndCB, WM_HELP | CMDBAR_OK, 0)
```

Command bars do not automatically resize when you resize a main window. To resize a Command bar along with a main window, wait until the main window receives a WM_SIZE message. Then send a TB_AUTOSIZE message to the command bar and call **CommandBar_AlignAdornments**. If you do not perform this procedure, the **OK**, **CANCEL**, and **HELP** command bar buttons will not stay flush with the right border of the window when the window size changes. The following code example shows how to resize a command bar along with a main window.

```
case WM_SIZE:
   // Tell the command bar to resize itself to fill the top of the
      window.
   SendMessage(hwndCB, TB_AUTOSIZE, 0L, 0L);
   CommandBar_AlignAdornments(hwndCB);
   break;
```

Creating a Command Bands Control

The command bands control is a special kind of rebar control. It has a fixed band at the top containing a toolbar with a **Close** (**X**) button and, optionally, a **Help** (**?**) button and an **OK** button in the right corner. By default, each band in the command bands control contains a command bar. You can override this, however, if you want a band to contain some other type of child window. The following screen shot shows a Windows CE command band.



▶ **To create a command bands control**

1. Initialize an **INITCOMMONCONTROLSEX** structure with (ICC_BAR_CLASSES | ICC_COOL_CLASSES) as the *dwICC* member.
2. Register the command bands control class and the command bar class by calling the **InitCommonControlsEx** function and passing in the **INITCOMMONCONTROLSEX** structure.
3. Create the image list to use for the band images.
4. Create the command bands control by calling the **CommandBands_Create** function and then passing the image list handle in the *himl* parameter.
5. Initialize an array of **REBARBANDINFO** structures, one for each band in the command bands control.
6. Add the bands by calling the **CommandBands_AddBands** function, passing the array of **REBARBANDINFO** structures in the *prbbi* parameter.
7. Add controls to the command bars in the bands by calling the appropriate command bar functions for the controls you want to add.
8. Call the **CommandBands_AddAdornments** function to add the **Close** and **Help** buttons. When you do this, the **Close** button is added by default.

The following code example shows how to register and create a command bands control.

```
HWND WINAPI CreateCmdband (HWND hwnd)
{
  HWND hwndCBar = NULL,     // The handle of the command bar control
       hwndCBand = NULL;    // The handle of the command bands control
  REBARBANDINFO rbi[3];     // REBARBANDINFO structures for command bands
  HIMAGELIST hImageList = NULL;
                            // Handle of the image list for command bands
  INITCOMMONCONTROLSEX iccex;
                            // INITCOMMONCONTROLSEX structure
```

```
iccex.dwSize = sizeof (INITCOMMONCONTROLSEX);
iccex.dwICC = ICC_BAR_CLASSES | ICC_COOL_CLASSES;

// Register toolbar and rebar control classes from the common control
// dynamic-link library (DLL).
InitCommonControlsEx (&iccex);

// Create the image list for the command bands.
if (!(hImageList = ImageList_LoadImage (
                hInst,
                MAKEINTRESOURCE (IDB_BANDIMAGE),
                16,
                3,
                CLR_DEFAULT,
                IMAGE_BITMAP,
                LR_DEFAULTCOLOR)))
  return NULL;

// Create the command bands control.
if (!(hwndCBand = CommandBands_Create (
                hInst,
                hwnd,
                ID_BAND,
                RBS_VARHEIGHT | RBS_BANDBORDERS | RBS_AUTOSIZE,
                hImageList)))
  return NULL;

// REBARBANDINFO for the menu band.
rbi[0].cbSize = sizeof (REBARBANDINFO);
rbi[0].fMask = RBBIM_STYLE | RBBIM_ID | RBBIM_SIZE;
rbi[0].fStyle = RBBS_CHILDEDGE | RBBS_NOGRIPPER;
rbi[0].wID = ID_BAND_MENUBAR;
rbi[0].cx = 150;

// REBARBANDINFO for the main toolbar band.
rbi[1].cbSize = sizeof (REBARBANDINFO);
rbi[1].fMask = RBBIM_TEXT | RBBIM_ID | RBBIM_IMAGE | RBBIM_STYLE;
rbi[1].fStyle = RBBS_BREAK | RBBS_GRIPPERALWAYS;
rbi[1].lpText = TEXT("Toolbar");
rbi[1].wID = ID_BAND_TOOLBAR;
rbi[1].iImage = 0;

// REBARBANDINFO for the font toolbar band.
rbi[2].cbSize = sizeof (REBARBANDINFO);
rbi[2].fMask = RBBIM_TEXT | RBBIM_ID | RBBIM_IMAGE | RBBIM_STYLE;
rbi[2].fStyle = RBBS_GRIPPERALWAYS;
rbi[2].lpText = TEXT("Font");
rbi[2].wID = ID_BAND_FONT_TOOLBAR;
rbi[2].iImage = 1;
```

```
// Adds bands to the command bands control.
  if (!CommandBands_AddBands (hwndCBand, hInst, 3, rbi))
    return NULL;

  // Insert a menu bar into the menu command band.
  if (hwndCBar = CommandBands_GetCommandBar (hwndCBand, 0))
    CommandBar_InsertMenubar (hwndCBar, hInst, IDM_MAIN_MENU, 0);

  // Add the buttons to the main toolbar band.
  if (hwndCBar = CommandBands_GetCommandBar (hwndCBand, 1))
  {
    CommandBar_AddBitmap (hwndCBar, hInst, IDB_TOOLBAR, 11, 0, 0);
    CommandBar_AddButtons (
                  hwndCBar,
                  sizeof (tbButtons) / sizeof (TBBUTTON),
                  tbButtons);
  }

  // Add the buttons to the font toolbar band.
  if (hwndCBar = CommandBands_GetCommandBar (hwndCBand, 2))
  {
    CommandBar_AddBitmap (hwndCBar, hInst, IDB_TOOLBAR, 11, 0, 0);
    CommandBar_AddButtons (
                  hwndCBar,
                  sizeof (tbFontButtons) / sizeof (TBBUTTON),
                  tbFontButtons);
  }

  // Add the Help and Close buttons to the command bands.
  CommandBands_AddAdornments (hwndCBand, hInst, CMDBAR_HELP, NULL);

  return hwndCBand;
}
```

Once you create a command bands control, you might want to add additional controls to the band or resize the band. Windows CE supports several functions for manipulating command bands.

The following table shows how to manipulate a command bands control.

To	Call
Add a band with the **Close** (**X**) button, the **Help** (**?**) button, and the **OK** button.	**CommandBands_AddAdornments**
Add one or more bands to the control. By default, each band has a command bar as its child window.	**CommandBands_AddBands**
Create a command bands control.	**CommandBands_Create**
Retrieve a command bar from a band in a command bands control. Pass the zero-based index of the band that contains the command bar you want to retrieve.	**CommandBands_GetCommandBar**
Return the height of the command bands control.	**CommandBands_Height**
Get the parent rectangle of the control.	**GetClientRect**
Determine whether a command bands control is visible.	**CommandBands_IsVisible**
Retrieve information about the bands in a command bands control so you can save the information in the registry to restore the command bands control to a previous state.	**CommandBands_GetRestoreInformation**
Show or hide the command bands control.	**CommandBands_Show**

Because a command band is a rebar control and a toolbar control, you can also manipulate it using rebar and toolbar messages.

Command bands controls support the custom draw service, which makes it easy to customize the appearance of a command bands control. For information about the custom draw service, see "Using the Custom Draw Service" later in this chapter.

Creating a Rebar Control

A rebar control, which has one or more bands, is a container for child windows. Each band can contain one child window, which can be a toolbar or any other control. Each band can have its own bitmap, which is displayed as a background for the toolbar on that band. A user can resize or reposition a band by dragging its *gripper bar*. A gripper bar appears on a rebar or a command bands control and is especially useful for bringing off-screen rebar or command bar controls into view. If a band has a text label next to its gripper bar, a user can maximize the band and restore it to its previous size by tapping the label with the stylus. The following screen shot shows a Windows CE rebar.

Like other common controls, a rebar control sends WM_NOTIFY messages to its parent window. A rebar control also forwards to its parent window all messages it receives from the child windows assigned to its bands.

Rebar controls also support the Windows CE custom draw service, which makes it easy to customize the appearance of a rebar control. For more information about the custom draw service, see "Using the Custom Draw Service" later in this chapter.

▶ To create a rebar control

1. Specify REBARCLASSNAME in the *lpClassName* parameter of the **CreateWindowEx** function.

   This class is registered when the common control DLL is loaded. You can also use the **InitCommonControlsEx** function to ensure that this DLL is loaded. To register the rebar control class using the **InitCommonControlsEx** function, specify the ICC_COOL_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

2. Specify a rebar style in the *dwStyle* parameter of the **CreateWindowEx** function.

   To place a toolbar inside a rebar, you must specify the CCS_NOPARENTALIGN style to ensure proper alignment.

The following code example shows how to create a rebar control.

```
HWND CreateRebar (HWND hwnd)
{
  HWND hwndRB = NULL,          // The handle to the rebar control
       hwndTB = NULL,          // The handle to the toolbar
       hwndCombo = NULL;       // The handle to the combo box control
  DWORD dwStyle;               // The window style used in CreateWindowEx
  int index;                   // An integer
  RECT rect;                   // A RECT structure
  TCHAR szString[64];          // A temporary string
  HICON hIcon;                 // A handle to a icon
  REBARINFO rbi;               // Contains information that describes
                               // rebar control characteristics
  HIMAGELIST himlRB;           // A handle to an image list
  REBARBANDINFO rbbi[2];       // Contains information that defines bands
                               // in the rebar control
  INITCOMMONCONTROLSEX iccex; // Carries information used to load rebar
                               // control classes
```

```
// Initialize the INITCOMMONCONTROLSEX structure.
  iccex.dwSize = sizeof (INITCOMMONCONTROLSEX);

  // Load rebar and toolbar control classes.
  iccex.dwICC = ICC_COOL_CLASSES | ICC_BAR_CLASSES;

  // Register rebar and toolbar control classes from the common control
  // dynamic-link library (DLL).
  InitCommonControlsEx (&iccex);

  // Create rebar control.
  dwStyle = WS_VISIBLE | WS_BORDER | WS_CHILD | WS_CLIPCHILDREN |
            WS_CLIPSIBLINGS | RBS_VARHEIGHT | RBS_BANDBORDERS |
            CCS_NODIVIDER | CCS_NOPARENTALIGN;

  if (!(hwndRB = CreateWindowEx (0,
                                 REBARCLASSNAME,
                                 NULL,
                                 dwStyle,
                                 0,
                                 0,
                                 CW_USEDEFAULT,
                                 100,
                                 hwnd,
                                 (HMENU)ID_REBAR,
                                 g_hInst,
                                 NULL)))
  {
    return NULL;
  }

  // Set the characteristics of the rebar control.
  himlRB = ImageList_Create (32, 32, ILC_COLORDDB | ILC_MASK, 1, 0);
  hIcon = LoadIcon (g_hInst, MAKEINTRESOURCE (IDI_REBAR));
  ImageList_AddIcon (himlRB, hIcon);

  rbi.cbSize = sizeof (rbi);
  rbi.fMask = RBIM_IMAGELIST;
  rbi.himl = himlRB;

  if (!SendMessage (hwndRB, RB_SETBARINFO, 0, (LPARAM)&rbi))
    return NULL;

  // Create a toolbar.
  dwStyle = WS_VISIBLE | WS_CHILD | TBSTYLE_TOOLTIPS |
            CCS_NOPARENTALIGN | CCS_NORESIZE;
```

```
if (!(hwndTB = CreateToolbarEx (hwnd,
                                dwStyle,
                                (UINT) ID_TOOLBAR,
                                NUMIMAGES,
                                g_hInst,
                                IDB_TOOLBAR,
                                tbButton,
                                sizeof (tbButton) / sizeof (TBBUTTON),
                                BUTTONWIDTH,
                                BUTTONHEIGHT,
                                IMAGEWIDTH,
                                IMAGEHEIGHT,
                                sizeof (TBBUTTON))))
  {
    return NULL;
  }

  // Add tooltips to the toolbar.
  SendMessage (hwndTB, TB_SETTOOLTIPS, (WPARAM) NUMIMAGES,
               (LPARAM) szToolTips);

  // Retrieve the dimensions of the bounding rectangle of the toolbar.
  GetWindowRect (hwndTB, &rect);

  memset (&rbbi[0], 0, sizeof (rbbi[0]));
  rbbi[0].cbSize = sizeof (REBARBANDINFO);
  rbbi[0].fMask = RBBIM_SIZE | RBBIM_CHILD | RBBIM_CHILDSIZE | RBBIM_ID
                  | RBBIM_STYLE | RBBIM_TEXT | RBBIM_BACKGROUND | 0;

  rbbi[0].cxMinChild = rect.right - rect.left + 2;
  rbbi[0].cyMinChild = rect.bottom - rect.top + 2;
  rbbi[0].cx = 250;
  rbbi[0].fStyle = RBBS_BREAK | RBBS_GRIPPERALWAYS;
  rbbi[0].wID = ID_TOOLBAR;
  rbbi[0].hwndChild = hwndTB;
  rbbi[0].lpText = TEXT("Toolbar");
  rbbi[0].hbmBack = LoadBitmap (g_hInst, MAKEINTRESOURCE (IDB_BKGRD));

  // Insert the toolbar band in the rebar control.
  SendMessage (hwndRB, RB_INSERTBAND, (WPARAM)-1,
               (LPARAM) (LPREBARBANDINFO)&rbbi[0]);

  // Create a combo box.
  dwStyle = WS_VISIBLE | WS_CHILD | WS_TABSTOP | WS_VSCROLL |
            WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
            CBS_AUTOHSCROLL | CBS_DROPDOWN;
```

```
if (!(hwndCombo = CreateWindowEx (0,
                                  TEXT("combobox"),
                                  NULL,
                                  dwStyle,
                                  0, 0, 100, 200,
                                  hwndRB,
                                  (HMENU)ID_COMBOBOX,
                                  g_hInst,
                                  NULL)))
  {
    return NULL;
  }

  // Add 10 items to the combo box.
  for (index = 0; index < 10; index++)
  {
    wsprintf (szString, TEXT("Item %d"), index + 1);
    SendMessage (hwndCombo, CB_ADDSTRING, 0, (LPARAM) szString);
  }

  // Select the first item as default.
  SendMessage (hwndCombo, CB_SETCURSEL, (WPARAM)0, 0);

  // Retrieve the dimensions of the bounding rectangle of the combo box.
  GetWindowRect (hwndCombo, &rect);

  memset (&rbbi[1], 0, sizeof (rbbi[1]));
  rbbi[1].cbSize = sizeof (REBARBANDINFO);
  rbbi[1].fMask = RBBIM_SIZE | RBBIM_CHILD | RBBIM_CHILDSIZE | RBBIM_ID
                | RBBIM_STYLE | RBBIM_TEXT | RBBIM_BACKGROUND
                | RBBIM_IMAGE | 0;

  rbbi[1].cxMinChild = rect.right - rect.left;
  rbbi[1].cyMinChild = rect.bottom - rect.top;
  rbbi[1].cx = 100;
  rbbi[1].fStyle = RBBS_CHILDEDGE | RBBS_FIXEDBMP | 0;
  rbbi[1].wID = ID_COMBOBOX;
  rbbi[1].hwndChild = hwndCombo;
  rbbi[1].lpText = TEXT("ComboBox");
  rbbi[1].hbmBack = LoadBitmap (g_hInst, MAKEINTRESOURCE (IDB_BKGRD));
  rbbi[1].iImage = 0;

  // Insert the combo box band in the rebar control.
  SendMessage (hwndRB, RB_INSERTBAND, (WPARAM)-1,
               (LPARAM) (LPREBARBANDINFO)&rbbi[1]);
```

```
    // Reposition the rebar control.
    MoveRebar (hwnd, hwndRB);

    return hwndRB;
}
```

Creating a Toolbar

A toolbar is a control that contains buttons. The buttons in a toolbar usually correspond to items on the application menu, providing a quick way for a user to access these commands. Toolbar buttons are bit images, and not child windows as are other buttons. When a user taps a toolbar button, the toolbar sends its parent window a WM_COMMAND message with the button's command identifier.

Each button in a toolbar can include a bitmap image. A toolbar maintains an internal list that contains all the bitmaps assigned to each of its toolbar buttons. When you call the **CreateToolbarEx** function, you specify a monochrome or color bitmap that contains the initial images. The toolbar then adds the information to the internal list of images. You can add additional images later by using the TB_ADDBITMAP message.

Each image has a zero-based index. The first image added to the internal list has an index of zero, the second image has an index of one, and so on. TB_ADDBITMAP adds images to the end of the list and returns the index of the first new image that it added. You use an image index to associate the image with a button.

Windows CE assumes that all toolbar bitmaps are the same size. You specify the size when you create the toolbar by calling **CreateToolbarEx**. If you call the **CreateWindowEx** function to create a toolbar, the size of its bitmaps is set to the default dimensions of 16 x 15 pixels. You can use the TB_SETBITMAPSIZE message to change the dimensions of the bitmaps, but you must do so before adding any images to the internal list of images.

Each button can display a string in addition to, or instead of, an image. A toolbar maintains an internal list that contains all of the strings available to toolbar buttons. You add strings to the internal list by using the TB_ADDSTRING message, specifying the address of the buffer containing the strings to add. Each string must be null-terminated, and the last string must be terminated with two null characters.

Each string has a zero-based index. The first string added to the internal list of strings has an index of zero, the second string has an index of one, and so on. TB_ADDSTRING adds strings to the end of the list and returns the index of the first new string. You use a string's index to associate the string with a button.

Each button in a toolbar has a current state that indicates whether the button is hidden or visible, enabled or disabled, and pressed or not pressed. You set a button's initial state when adding the button to the toolbar, and the toolbar updates the button state in response to a user's actions, for example, when a user taps it with a stylus. You can use the TB_GETSTATE and TB_SETSTATE messages to retrieve and set the state of a button.

The following table shows toolbar button states supported by Windows CE.

State	Description
TBSTATE_CHECKED	The button has the TBSTYLE_CHECKED style and is pressed.
TBSTATE_ELLIPSES	The button displays ellipses if the text does not fit the size of the button. This style is unique to Windows CE.
TBSTATE_ENABLED	The button accepts user input. A button without this state does not accept user input and is dimmed.
TBSTATE_HIDDEN	The button is not visible and cannot receive user input.
TBSTATE_HIGHLIGHTED	The button is highlighted.
TBSTATE_INDETERMINATE	The button is dimmed.
TBSTATE_PRESSED	The button is being pressed.
TBSTATE_WRAP	The button has a line break following it. The button must also have the TBSTATE_ENABLED state.

▶ To create a toolbar

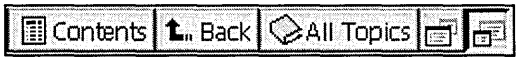
- Use the **CreateToolbarEx** function, which has the following syntax:

```
HWND CreateToolbarEx(
    HWND hwnd,
    DWORD ws,
    UINT wID,
    int nBitmaps,
    HINSTANCE hBMInst,
    UINT wBMID,
    LPCTBBUTTON lpButtons,
    int iNumButtons,
    int dxButton,
    int dyButton,
    int dxBitmap,
    int dyBitmap,
    UINT uStructSize
);
```

Here, *hwnd* is the handle to the parent window that owns the toolbar and *ws* is the style of the toolbar. Minimally, a toolbar must include the WS_CHILD style. You can also specify other styles. For example, in Windows CE the TBSTYLE_LIST style creates a toolbar with variable-width buttons. If you want to use the TBSTYLE_LIST style with fixed-width buttons, you can override the default behavior by sending a TB_SETBUTTONSIZE or TB_SETBUTTONWIDTH message. To keep a toolbar from automatically aligning to the top or bottom of a parent window, specify the CCS_NOPARENTALIGN style.

The identifier associated with the toolbar is specified in *wID* and the number of button images contained in the bitmap specified by *hBMInst* and *wBMID* is specified in *nBitmaps*.

Information about each button is contained in an array of structures called **TBBUTTON**; *lpButtons* is the address of this array. Specific information about buttons, such as the number of buttons to add to the toolbar, button height and width, and the height and width of button images, are specified in *iNumButtons*, *dxButton*, *dyButton*, *dxBitmap*, and *dyBitmap*, respectively. The size of the **TBBUTTON** structure is specified in *uStructSize*. The following screen shot shows a Windows CE toolbar.



The following code example shows how to create and register a toolbar.

```
HWND WINAPI CreateToolbar (HWND hwnd)
{
  int iCBHeight;              // Command bar height
  DWORD dwStyle;              // Style of the toolbar
  HWND hwndTB = NULL;         // Handle of the command bar control
  RECT rect,                  // Contains the coordinates of the main
                              // window's client area
       rectTB;                // Contains the dimensions of the bounding
                              // rectangle of the toolbar control
  INITCOMMONCONTROLSEX iccex; // INITCOMMONCONTROLSEX structure

  iccex.dwSize = sizeof (INITCOMMONCONTROLSEX);
  iccex.dwICC = ICC_BAR_CLASSES;

  // Register toolbar control classes from the common control
  // dynamic-link library (DLL).
  InitCommonControlsEx (&iccex);

  // Create the toolbar control.
  dwStyle = WS_VISIBLE | WS_CHILD | TBSTYLE_TOOLTIPS |
            CCS_NOPARENTALIGN;
```

```
    if (!(hwndTB = CreateToolbarEx (
                    hwnd,               // Parent window handle
                    dwStyle,            // Toolbar window styles
                    (UINT) ID_TOOLBAR,  // Toolbar control identifier
                    NUMIMAGES,          // Number of button images
                    hInst,              // Module instance
                    IDB_TOOLBAR,        // Bitmap resource identifier
                    tbButton,           // Array of TBBUTTON structure
                                        // contains button information
                    sizeof (tbButton) / sizeof (TBBUTTON),
                                        // Number of buttons in toolbar
                    BUTTONWIDTH,        // Width of the button in pixels
                    BUTTONHEIGHT,       // Height of the button in pixels
                    IMAGEWIDTH,         // Button image width in pixels
                    IMAGEHEIGHT,        // Button image height in pixels
                    sizeof (TBBUTTON))))// Size of a TBBUTTON structure
    {
      return NULL;
    }

    // Add tooltips to the toolbar.
    SendMessage (hwndTB, TB_SETTOOLTIPS, (WPARAM) NUMIMAGES,
                 (LPARAM) szToolTips);

    // Reposition the toolbar.
    GetClientRect (hwnd, &rect);
    GetWindowRect (hwndTB, &rectTB);
    iCBHeight = CommandBar_Height (hwndCB);
    MoveWindow (hwndTB,
                0,
                iCBHeight - 2,
                rect.right - rect.left,
                rectTB.bottom - rectTB.top,
                TRUE);

    return hwndTB;
}
```

You can also call the **CreateWindowEx** function to create a toolbar. Using this method, however, creates a toolbar that initially contains no buttons. You can then add buttons to the toolbar by using the TB_ADDBUTTONS or TB_INSERTBUTTON message. You register the toolbar class by specifying the **TOOLBARCLASSNAME** window class. Windows CE registers the **TOOLBARCLASSNAME** class when it loads the common control DLL. You can call the **InitCommonControls** function to ensure that this DLL is loaded. To register the toolbar class using the **InitCommonControlsEx** function, specify the ICC_BAR_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

If you use **CreateWindowEx** to create a toolbar, you must specify the WS_CHILD window style. **CreateToolbarEx** includes the WS_CHILD style by default. You must specify the initial parent window when creating the toolbar, but you can change the parent window after creation by using the TB_SETPARENT message.

Windows CE does not support user customization of toolbars or drag-and-drop operations for toolbars.

Specifying Toolbar Size, Position, and Appearance

The window procedure for a toolbar automatically sets the size and position of the toolbar window. The height is based on the height of the buttons in the toolbar. The width is the same as the width of the parent window's client area. The CCS_TOP and CCS_BOTTOM common control styles determine whether the toolbar is positioned along the top or bottom of the client area. By default, a toolbar has the CCS_TOP style.

The toolbar window procedure automatically adjusts the size of the toolbar when it receives a WM_SIZE or TB_AUTOSIZE message. An application should send either of these messages when the size of the parent window changes or after sending a message that requires the size of the toolbar to be adjusted, for example, after sending the TB_SETBUTTONSIZE message.

Windows CE also supports messages that enable you to customize the look and behavior of toolbars and toolbar buttons.

Create transparent toolbars by specifying the TBSTYLE_FLAT or TBSTYLE_TRANSPARENT styles. If you give a toolbar the TBSTYLE_FLAT style, the toolbar displays its buttons but the toolbar itself is transparent. If you give a toolbar the TBSTYLE_TRANSPARENT style, the client area shows through the buttons as well as through the underlying toolbar.

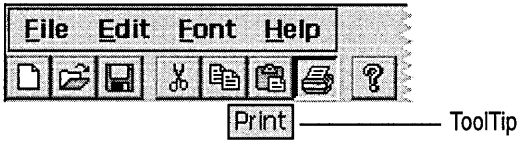
You can use image lists to customize the way a toolbar displays buttons in various states. You can set and retrieve image lists for toolbar buttons by using the TB_GETIMAGELIST and TB_SETIMAGELIST messages for buttons in their default unpressed state, and the TB_GETDISABLEDIMAGELIST and TB_SETDISABLEDIMAGELIST messages for buttons in their disabled state. Use the TB_LOADIMAGES message to load images into a toolbar image list.

Windows CE supports a toolbar button style called a drop-down button. When a user taps a button that has the TBSTYLE_DROPDOWN style, the toolbar sends a TBN_DROPDOWN notification to its parent window. The parent window usually responds by displaying a pop-up menu or list box under the drop-down button.

Because toolbars in Windows CE support the custom draw service, you have flexibility to customize the appearance of a toolbar. If a toolbar provides this service, it sends the new NM_CUSTOMDRAW notification at specific times during drawing operations. The *lParam* of the NM_CUSTOMDRAW notification is a pointer to an **NMCUSTOMDRAW** structure, which contains the information necessary to draw the customized toolbar. For information about the custom draw service, see "Using the Custom Draw Service" later in this chapter.

Creating ToolTips

A *ToolTip* is a small, rectangular pop-up window that displays a brief description of the purpose of a command bar button when a user holds the stylus on the button for more than 0.5 seconds. If a user lifts the stylus from the screen while it is still positioned over the button, the button is activated. If a user moves the stylus away from the button before raising the stylus from the screen, the button is not activated. The following screen shot shows a ToolTip.



Windows CE supports ToolTips only for command bar and toolbar buttons and for command bar menus. It does not support ToolTips for the combo boxes in a command bar. To add ToolTips to a command bar, call the **CommandBar_AddTooltips** function.

The **CommandBar_AddtoolTips** function does not make a copy of the array of ToolTip strings you pass to it. It directly uses the memory address you pass to it in the *lpToolTips* parameter. Do not release the memory allocated for this array until the application exits.

To add a ToolTip to a toolbar, use the TB_SETTOOLTIPS message.

ToolTips usually display only the name of a button command, but they can also display the shortcut key for the command.

Creating a Header Control

A *header control* is a horizontal window 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. A user can drag the dividers between the partitions to set the width of each column. A header can also perform an action, such as sorting the rows of data according to the values in a column a user selects.

A header control sends notification messages to its parent window when a user taps or double-taps an item, when a user drags an item divider, and when the item attributes change. The parent window receives the notifications in the form of WM_NOTIFY messages.

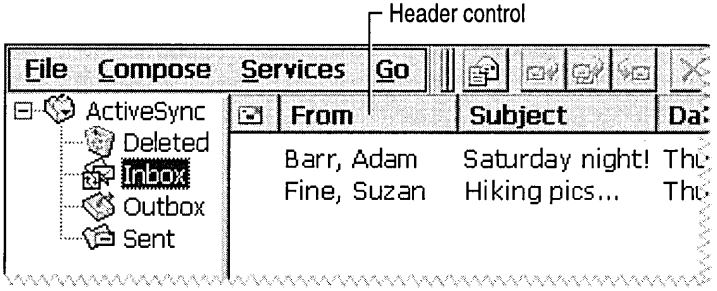
Windows CE supplies macros to send header control messages as well as to support the use of image lists, drag-and-drop features, and custom ordering of header control items.

▶ **To create a header control**

1. Specify WC_HEADER in the *lpClassName* parameter of the **CreateWindowEx** function. This class is registered when the common control DLL is loaded. Use the **InitCommonControls** function to ensure that this DLL is loaded.
2. Specify a control style in the *dwStyle* parameter of the **CreateWindowEx** function.

▶ **To register the header control class using the InitCommonControlsEx function**

- Specify the ICC_ LISTVIEW_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter. The following screen shot shows a Windows CE header control.



Setting Header Control Size and Position

Typically, you must set the size and position of a header control to fit within the boundaries of a particular rectangle, such as the client area of a window. By using the HDM_LAYOUT message, you can retrieve the appropriate size and position values from the header control.

When sending the HDM_LAYOUT message, you specify the address of an **HDLAYOUT** structure that contains the coordinates of the rectangle that the header control is to occupy and that provides a pointer to a **WINDOWPOS** structure. The control fills **WINDOWPOS** with size and position values appropriate for positioning the control along the top of the specified rectangle. The height value is the sum of the heights of the control's horizontal borders and the average height of characters in the font currently selected into the control's device context.

Adding Header Control Items

A header control typically has several header items that define the columns of the control. To add an item to a header control, send the HDM_INSERTITEM message to the control. The message includes the address of an **HDITEM** structure. This structure defines the properties of the header item. The following code example shows the **HDITEM** syntax.

```
typedef struct _HD_ITEM { hdiUINTmask;intcxy; LPSTRpszText; HBITMAPhbm;
intcchTextMax; intfmt; LPARAMlParam; } HD_ITEM;
```

The *fmt* member of an item's **HDITEM** structure can include either the HDF_STRING or HDF_BITMAP flag to indicate whether the control displays the item's string or bitmap. If you want to display both a string and a bitmap, create an owner-drawn item by setting the *fmt* member to include the HDF_OWNERDRAW flag. You can combine a string and an image from an image list by combining the HDF_IMAGE and HDF_STRING flags.

The **HDITEM** structure also specifies formatting flags that tell the control whether to center, left-align, or right-align the string or bitmap in the item's rectangle.

HDM_INSERTITEM returns the index of the newly added item. You can use the index in other messages to set properties or retrieve information about the item. To delete an item, use the HDM_DELETEITEM message, which specifies the index of the item to delete.

The HDM_SETITEM message sets the properties of an existing header item and the HDM_GETITEM message retrieves the current properties of an item. To retrieve a count of the items in a header control, use the HDM_GETITEMCOUNT message.

You can define individual items of a header control to be owner-drawn items. Using this technique gives you more control than you would otherwise have over the appearance of a header item.

Use the HDM_INSERTITEM message to insert a new owner-drawn item into a header control or the HDM_SETITEM message to change an existing item to an owner-drawn item. Both messages include the address of an **HDITEM** structure, which should have the *fmt* member set to the HDF_OWNERDRAW value.

Working with Advanced Header Control Features

Windows CE enables you to use image lists in header controls, as well as text and bitmaps. An *image list* is a collection of same-size images, such as bitmaps or icons.

You can use the HDM_SETIMAGELIST message to associate an image list with a header control. Use the HDM_GETIMAGELIST message to retrieve the handle of the image list associated with a header control. To display an image with a header control item, specify HDI_IMAGE as the *mask* member, HDF_IMAGE as the *fmt* member, and the zero-based index of an image in the list as the *iImage* member of the **HDITEM** structure that you use to add the item to the header control.

Header controls support callback requests for text and images in header control items. To create a callback item, set the *pszText* member to LPSTR_TEXTCALLBACK or the *iImage* member to I_IMAGECALLBACK in the **HDITEM** structure that you fill in when you add the item to the header control. This causes the header control to send the HDN_GETDISPINFO notification message when the item is about to be drawn. The *lParam* of the WM_NOTIFY message is a pointer to an **NMHDDISPINFO** structure. When the header control sends the notification, it sets the **NMHDDISPINFO** structure's members to specify the type of information it needs in order to draw the item. Return the requested information to the header control by filling in the appropriate members of the structure. If you set the *mask* member to HDI_DI_SETITEM, the header control stores the information and does not request it again. Otherwise, the header control sends the NMHDDISPINFO notification each time the item is redrawn.

Header controls also support drag-and-drop features. To create a header control that supports drag-and-drop operations, specify the HDS_DRAGDROP style when you create the header control. You can also customize a header control's drag-and-drop behavior by handling the HDN_BEGINDRAG and HDN_ENDDRAG notification messages and by sending HDM_CREATEDRAGIMAGE and HDM_SETHOTDIVIDER messages.

You can support custom ordering of items in a header control by setting the *iOrder* member in the **HDITEM** structure when you add an item to a header control and by using the HDM_GETORDERARRAY, HDM_SETORDERARRAY, and HDM_ORDERTOINDEX messages.

Header controls support the custom draw service, which gives you flexibility to customize the appearance of a header control. If a header control provides this service, it sends the NM_CUSTOMDRAW notification at specific times during drawing operations. The *lParam* of the NM_CUSTOMDRAW notification is a pointer to an **NMCUSTOMDRAW** structure, which contains the information necessary to draw the customized header control.

Creating an Image List

An image list is a collection of same-size images. You can create the images in a single wide bitmap or as individual bitmaps that you add to the list one at a time. Image lists manage images, but they do not display the images directly. They can be used independently or in conjunction with list view and tree view controls.

There are two types of image lists, nonmasked and masked. A nonmasked image list consists of a color bitmap that contains one or more images. A masked image list consists of two bitmaps of equal size. The first is a color bitmap that contains the images, and the second is a monochrome bitmap that contains a series of masks—one for each image in the first bitmap.

Windows CE draws a nonmasked image by simply copying it into the target device context and drawing it over the existing background color of the device context. Windows CE draws a masked image by combining its bits with the bits of the mask, typically producing transparent areas in the bitmap where the background color of the target device context shows through.

Note Most Windows CE–based platforms do not support cursors except for the wait cursor. Therefore, image lists cannot contain cursors.

▶ **To create an image list**

1. Call the **ImageList_Create** function.

   For a nonmasked image list, this function creates a single bitmap large enough to hold a specified number of images of the specified dimensions. Then it creates a screen-compatible device context and selects the bitmap into it. For a masked image list, the function creates two bitmaps and two screen-compatible device contexts. **ImageList_Create** selects the image bitmap into one device context and the mask bitmap into the other.

2. Specify the initial number of images in the image list, as well as the number of images by which the list can grow.

   If you attempt to add more images than you initially specified, the image list automatically grows to accommodate the images.

If **ImageList_Create** succeeds, it returns a handle to the HIMAGELIST type. Use this handle in other image list functions to access the image list and manage the images. You can add and remove images, copy images from one image list to another, and merge images from two different image lists. When you no longer need an image list, destroy it by specifying its handle in a call to the **ImageList_Destroy** function.

Use the **ImageList_Duplicate**, **ImageList_SetImageCount**, and **ImageList_RemoveAll** functions to respectively copy, resize, or remove all images from an image list.

The **IMAGELISTDRAWPARAMS** structure, used with the **ImageList_DrawIndirect** function, contains information about how to draw an image from an image list, such as what part of the image to draw, the foreground and background colors, the style, and a raster operation code specifying how to combine the image's colors with the background colors.

Using Images in Image Lists

You can add icons or other bit images to an image list. To add bit images, specify the handles to two bitmaps in a call to the **ImageList_Add** function. The first bitmap contains one or more images to add to the image bitmap, and the second bitmap contains the masks to add to the mask bitmap. Windows CE ignores the second bitmap handle for nonmasked images; you can set it to NULL.

The **ImageList_AddMasked** function adds bit images to a masked image list. This function is similar to **ImageList_Add**, in which you do not specify a mask bitmap. Instead, you specify a color that the system combines with the image bitmap to automatically generate the masks. Windows CE changes each pixel of the specified color in the image bitmap to black and sets the corresponding bit in the mask to one. As a result, any pixel in the image that matches the specified color is transparent when the image is drawn.

The **ImageList_AddIcon** function adds an icon to an image list. If the image list is masked, **ImageList_AddIcon** adds the mask provided with the icon to the mask bitmap. If the image list is not masked, the mask for the icon is not used when drawing the image.

To create an icon based on an image and mask in an image list, use the **ImageList_GetIcon** function. The function returns the handle to the icon. **ImageList_Add**, **ImageList_AddMasked**, and **ImageList_AddIcon** assign an index to each image as it is added to an image list. When more than one image is added at a time, the functions return the index of the first image. The **ImageList_Remove** function removes an image from an image list.

The **ImageList_Replace** and **ImageList_ReplaceIcon** functions replace an image in an image list with a new image. **ImageList_Replace** replaces an image with a bit image and mask, and **ImageList_ReplaceIcon** replaces an image with an icon. Use the **ImageList_Copy** function to move or copy images within an image list.

The **ImageList_Merge** function merges two images, storing the new image in a new image list. The new image is created by drawing the second image transparently over the first. The mask for the new image is the result of performing a logical **OR** operation on the bits of the masks for the two original images.

The **ImageList_GetImageInfo** function fills an **IMAGEINFO** structure with information about a single image, including the handles of the image and mask bitmaps, the number of color planes and bits per pixel, and the bounding rectangle of the image within the image bitmap. Use this information to directly manipulate the bitmaps for the image. The **ImageList_GetImageCount** function retrieves the number of images in an image list.

Use the **ImageList_DrawIndirect** function to specify custom drawing properties for an image in an image list. This function takes a pointer to an **IMAGELISTDRAWPARAMS** structure as a parameter. The **IMAGELISTDRAWPARAMS** structure contains information about how to draw the image.

Using Overlays in Image Lists

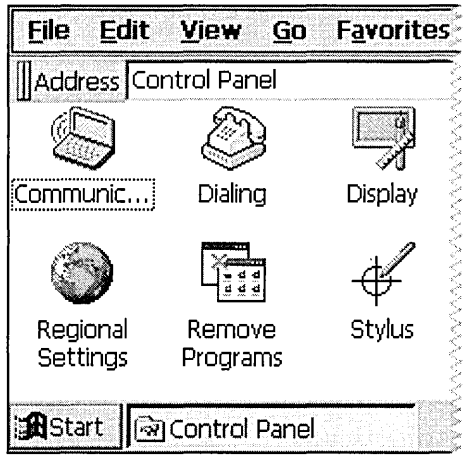
Every image list includes a list of indexes to use as overlays. An overlay is an image drawn transparently over another image. Any image currently in the image list can be used as an overlay. You can specify up to four overlays for each image list.

Add the index of an image to the list of overlays by using the **ImageList_SetOverlayImage** function, specifying the handle to the image list, the index of the existing image, and the overlay index that you want. The overlay indexes are one-based rather than zero-based because an overlay index of zero means that no overlay will be used.

Specify an overlay when drawing an image with the **ImageList_Draw** or **ImageList_DrawEx** function. The overlay is specified by performing a logical **OR** operation between the desired drawing flags and the result of the **INDEXTOOVERLAYMASK** macro. The **INDEXTOOVERLAYMASK** macro formats the overlay index for inclusion with the flags for these functions.

Creating a List View Control

A *list view* is a common control that displays a collection of items, such as files or folders. Each item has an icon and a label. A user can choose whether to have the items displayed as large icons, small icons, a list, or a detailed list. You can design list views so that a user can drag an item to a new location within the list view or sort the collection by tapping a column header. The following screen shot shows an image list in list view.



▶ **To create a list view control**

1. Specify the WC_LISTVIEW class in the *lpClassName* parameter of the **CreateWindowEx** function.

   This class is registered when the common control DLL is loaded. Use the **InitCommonControls** function to ensure that this DLL is loaded. To register the list view class using the **InitCommonControlsEx** function, specify the ICC_LISTVIEW_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

2. Specify a list view style in the *dwStyle* parameter of the **CreateWindowEx** function.

You can speed up the creation of large list views by disabling the painting of the list view before adding the items. You do this by sending a WM_SETREDRAW message with the redraw flag in *wParam* set to FALSE. When you are finished adding items, re-enable painting by sending a WM_SETREDRAW message with the redraw flag *wParam* set to TRUE. Before inserting items, send the LVM_SETITEMCOUNT message with the *cItems* parameter set to the number of items in question. When you do this, the list view will allocate the memory it needs all at once, instead of having to reallocate more memory incrementally as the internal data structures grow.

You can change the view type after a list view control is created. To retrieve and change the window style, use the **GetWindowLong** and **SetWindowLong** functions. To determine the window styles that correspond to the current view, use the LVS_TYPEMASK value.

You can control the way items are arranged in icon view or small icon view by specifying either the LVS_ALIGNTOP windows style, which is the default, or the LVS_ALIGNLEFT window style. You can change the alignment after a list view control is created. To isolate the window styles that specify the alignment of items, use the LVS_ALIGNMASK value.

Windows CE does not support hot tracking, hover selection, background images, or list view ToolTips.

Creating Image Lists

By default, a list view control does not display item images. To display item images, you must create image lists and associate them with the control. A list view control can have three image lists:

- One that contains full-sized icons displayed when the control is in icon view
- One that contains small icons displayed when the control is in small icon view, list view, or report view
- One that contains state images displayed to the left of the full-size icon or small icon

You can use state images, such as checked or cleared check boxes, to indicate application-defined item states. State images are displayed in icon view, small icon view, list view, or report view.

To assign an image list to a list view control, use the LVM_SETIMAGELIST message to specify whether the image list contains full-size icons, small icons, or state images. To retrieve the handle to an image list currently assigned to a list view control, use the LVM_GETIMAGELIST message. You can use the **GetSystemMetrics** function to determine appropriate dimensions for the full-size icons and small icons. Use the **ImageList_Create** function to create an image list, and use other image list functions to add bitmaps to the image list.

Create only the image list that the control will use. For example, if the list view control will never be in icon view, do not create and assign a large image list because the large images will never be used. If you create large and small icon image lists, each image list must contain the same images in the same order. This is because a single value is used to identify a list view item's icon in both image lists. You can associate an icon index with an item when you call the **ListView_InsertItem** or **ListView_SetItem** macro.

The full-size icon and small icon image lists can also contain overlay images, designed to be drawn transparently over the item icons.

▶ **To use overlay images in a list view control**

1. Call the **ImageList_SetOverlayImage** function to assign an overlay image index to an image in the full-size icon and small icon image lists.

   An overlay image is identified by a one-based index.

2. Call the **ListView_InsertItem** or **ListView_SetItem** macro to associate an overlay image index with an item.

3. Use the **INDEXTOOVERLAYMASK** macro to specify an overlay image index in the *state* member of the item's **LVITEM** structure.

   You must also set the **LVIS_OVERLAYMASK** bits in the *stateMask* member.

To associate a state image with an item, use the **INDEXTOSTATEIMAGEMASK** macro to specify a state image index in the *state* member of the **LVITEM** structure.

By default, when a list view control is destroyed, it destroys the image lists assigned to it. However, if a list view control has the LVS_SHAREIMAGELISTS window style, you are responsible for destroying the image lists when they are no longer in use. You should specify this style if you assign the same image lists to multiple list view controls; otherwise, more than one control might try to destroy the same image list.

The following code example shows how to create a list view control and accompanying image list. It also shows how to assign the image list to the control.

```
HWND CreateListView (HINSTANCE hInstance, HWND hwndParent)
{
  DWORD dwStyle;              // The window style of the list view
                              // control
  HWND hwndListView;          // The handle of the list view control
  HIMAGELIST himlSmall;       // The handle to the small image list
  HIMAGELIST himlLarge;       // The handle to the large image list
  INITCOMMONCONTROLSEX iccex; // INITCOMMONCONTROLSEX structure

  // Initialize the INITCOMMONCONTROLSEX structure.
  iccex.dwSize = sizeof (INITCOMMONCONTROLSEX);

  // Load list view and header control classes.
  iccex.dwICC = ICC_LISTVIEW_CLASSES;

  // Register tree view control classes from the common control
  // dynamic-link library (DLL).
  InitCommonControlsEx (&iccex);

  // Assign the list view window style.
  dwStyle = WS_TABSTOP | WS_CHILD | WS_BORDER | WS_VISIBLE |
            LVS_AUTOARRANGE | LVS_REPORT | LVS_OWNERDATA;

  // Create list view control.
  hwndListView = CreateWindowEx (
                 WS_EX_CLIENTEDGE,   // Extended window style
                 WC_LISTVIEW,        // Class name
                 TEXT(""),           // Window name
                 dwStyle,            // Window style
                 0,                  // Horizontal position of window
                 0,                  // Vertical position of window
                 0,                  // Window width
                 0,                  // Window height
                 hwndParent,         // Handle to parent window
                 (HMENU)ID_LISTVIEW, // Handle to menu identifier
                 g_hInst,            // Handle to application instance
                 NULL);              // Window-creation data

  // If it fails in creating the window, return.
  if (!hwndListView)
    return NULL;

  // Insert code to resize the list view window here since the list view
  // control was created zero in size.
  // ...
```

```
  // Create the large and small image lists.
  himlSmall = ImageList_Create (16, 16, ILC_COLORDDB | ILC_MASK, 1, 0);
  himlLarge = ImageList_Create (32, 32, ILC_COLORDDB | ILC_MASK, 1, 0);

  if (himlSmall && himlLarge)
  {
    HICON hIcon;

    // Load the small icon from the instance.
    hIcon = LoadImage (g_hInst, MAKEINTRESOURCE (IDI_DISK), IMAGE_ICON,
                       16, 16, LR_DEFAULTCOLOR);

    // Add the icon to the image list.
    ImageList_AddIcon (himlSmall, hIcon);

    // Load the small icon from the instance.
    hIcon = LoadIcon (g_hInst, MAKEINTRESOURCE (IDI_DISK));

    // Adds the icon to the image list.
    ImageList_AddIcon (himlLarge, hIcon);

    // Assign the large and small image lists to the list view control.
    ListView_SetImageList (hwndListView, himlSmall, LVSIL_SMALL);
    ListView_SetImageList (hwndListView, himlLarge, LVSIL_NORMAL);
  }

  return hwndListView;
}
```

Adding Items and Subitems

Each item in a list view control has an icon, a label, a current state, and an application-defined value. By using list view messages, you can add, modify, and delete items as well as retrieve information about items. You can also find items with specific attributes.

Each item can also have one or more *subitem*. A subitem is a string that, in report view, is displayed in a column to the right of an item's icon and label. To specify the text of a subitem, use the LVM_SETITEMTEXT or LVM_SETITEM message. All items in a list view control have the same number of subitems. The number of subitems is determined by the number of columns in the list view control.

The **LVITEM** structure defines a list view item or subitem. To add an item to a list view control, use the LVM_INSERTITEM message. Before adding multiple items, you can send the control an LVM_SETITEMCOUNT message to specify the number of items the control will ultimately contain. This message enables the list view control to reallocate its internal data structures only once rather than every time you add an item. Determine the number of items in a list view control by using the LVM_GETITEMCOUNT message.

Use the LVM_SETITEM message to change the attributes of a list view item. The LVM_SETITEMTEXT message changes only the text of an item or subitem.

To retrieve information about a list view item, use the LVM_GETITEM message specifying the address of the **LVITEM** structure to fill in. To retrieve only an item or subitem's text, use the LVM_GETITEMTEXT message. To delete a list view item, use the LVM_DELETEITEM message. Delete all items in a list view control by using the LVM_DELETEALLITEMS message.

Adding Callback Items and Callback Masks

For each of its items, a list view control typically stores the label text, the image list index of the item's icons, and a set of bit flags for the item state. A callback item in a list view control is an item for which the application stores the text, icon index, or both. You can define callback items or change the control's callback mask to indicate that the application—rather than the control—stores some or all of this data. You may want to use callbacks if your application already stores some of this data. You can define callback items when you send the LVM_INSERTITEM message to add an item to the list view control.

The callback mask of a list view control is a set of bit flags that specify the item states for which the application, rather than the control, stores the current data. The callback mask applies to all the control items, unlike the callback item designation, which applies to a specific item. The callback mask is zero by default, meaning that the list view control stores all item-state data. After creating a list view control and initializing its items, you can send the LVM_SETCALLBACKMASK message to change the callback mask. To get the current callback mask, send the LVM_GETCALLBACKMASK message.

When a list view control must display or sort a list view item for which the application stores callback data, the control sends the LVN_GETDISPINFO notification message to the control's parent window. This message specifies an **NMLVDISPINFO** structure that indicates the type of data required. The parent window must process LVN_GETDISPINFO to provide the requested data.

If the list view control detects a change in an item's callback data, the control sends an LVN_SETDISPINFO notification message to notify you of the change. Changes that the list view control detects are alterations to the text, the icon, or the state data being tracked by the application.

The following code example shows how the list view control requests data.

```
LRESULT ListViewNotify (HWND hwnd, LPARAM lParam)
{
  LPNMHDR lpnmh = (LPNMHDR) lParam; // Contains information about the
                                    // notification message.
  HWND hwndListView  = GetDlgItem (hwnd, ID_LISTVIEW);
                                    // Handle of the list view control
  switch (lpnmh->code)
  {
    case LVN_GETDISPINFO:
    {
      TCHAR szString[MAX_PATH];
      LV_DISPINFO *lpdi = (LV_DISPINFO *) lParam;

      // The message LVN_GETDISPINFO is sent by the list view control to
      // its parent window. It is a request for the parent window to
      // provide information needed to display or sort a list view item.

      return 0;
    }

    case LVN_ODCACHEHINT:
    {
      LPNMLVCACHEHINT lpCacheHint = (LPNMLVCACHEHINT)lParam;

      // The message LVN_ODCACHEHINT is sent by the list view control
      // when the contents of its display area have changed. For
      // example, a list view control sends this notification when the
      // user scrolls the control's display.

      return 0;
    }

    case LVN_ODFINDITEM:
    {
      LPNMLVFINDITEM lpFindItem = (LPNMLVFINDITEM)lParam;

      // The message LVN_ODFINDITEM is sent by the list view control
      // when it needs the owner to find a particular callback item.
      // Return -1 if the item is not found.

      return 0;
    }
  }
```

If you change a callback item's attributes or state bits, you can use the LVM_UPDATE message to force the control to repaint the item. This message also causes the control to arrange its items if it has the LVS_AUTOARRANGE style. You can use the LVM_REDRAWITEMS message to redraw a range of items by invalidating the corresponding portions of the list view control's client area.

Adding Columns

Columns control the way items and their subitems are displayed in report view. Each column has a title and width and is associated with a specific subitem. The attributes of a column are defined by an **LVCOLUMN** structure.

To add a column to a list view control, use the LVM_INSERTCOLUMN message. To delete a column, use the LVM_DELETECOLUMN message. You can retrieve and change the properties of an existing column by using the LVM_GETCOLUMN and LVM_SETCOLUMN messages. To retrieve or change a column width, use the LVM_GETCOLUMNWIDTH and LVM_SETCOLUMNWIDTH messages.

Unless the LVS_NOCOLUMNHEADER window style is specified, column headers appear in report view. A user can tap a column header, which causes the list view control to send an LVN_COLUMNCLICK notification message to the parent window. Typically, the parent window sorts the list view control by the specified column when a user taps the column header.

List view controls can set the order in which columns are displayed. To implement this feature, specify the LVCF_ORDER value and assign the proper value to the *iOrder* member in the **LVCOLUMN** structure.

The following code example shows how to add columns and set the number of items in the list view window.

```
BOOL InitListView (HWND hwndListView)
{
  int index;
  LV_COLUMN lvColumn;
  TCHAR szString[5][20] = {TEXT("Main Column"),
                           TEXT("Column 1"),
                           TEXT("Column 2"),
                           TEXT("Column 3"),
                           TEXT("Column 4")};

  // Empty the list in list view.
  ListView_DeleteAllItems (hwndListView);
```

```
  // Initialize the columns in the list view.
  lvColumn.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
  lvColumn.fmt = LVCFMT_LEFT;
  lvColumn.cx = 120;

  // Insert the five columns in the list view.
  for (index = 0; index < 5; index++)
  {
    lvColumn.pszText = szString[index];
    ListView_InsertColumn (hwndListView, index, &lvColumn);
  }

  // Set the number of items in the list to ITEM_COUNT.
  ListView_SetItemCount (hwndListView, ITEM_COUNT);

  return TRUE;
}
```

Arranging, Sorting, and Finding List Views

You can use list view messages to arrange and sort items and to find items based on their attributes or position. Although arranging items repositions them to align on a grid, the indexes of the items do not change. Sorting changes the sequence of items and their corresponding indexes, and then repositions them in the order specified. You can arrange items only in icon and small icon views, but you can sort items in any view.

To arrange items, use the LVM_ARRANGE message. You can ensure that items are arranged at all times by specifying the LVS_AUTOARRANGE window style.

To sort items, use the LVM_SORTITEMS message. When you sort using this message, you specify an application-defined callback function that the list view control calls to compare the relative order of any two items. By specifying the appropriate item data and supplying an appropriate comparison function, you can sort items by their labels, by any subitems, or by any other properties. Note that sorting items does not reorder the corresponding subitems. Thus, if any subitems are not callback items, you must regenerate the subitems after sorting.

Ensure that a list view control is always sorted by specifying the LVS_SORTASCENDING or LVS_SORTDESCENDING window style. Controls with these styles use the label text of the items to sort them in ascending or descending order. You cannot supply a comparison function when using these window styles.

You can find a list view item with specific properties by using the LVM_FINDITEM message. Use the LVM_GETNEXTITEM message to find a list view item in a specified state that bears a specified geometrical relationship to a specified item.

Setting the List View Item and Scroll Position

Every list view item has a position and size, which you can retrieve and set using messages. You can also determine which item, if any, is at a specified position. The position of list view items is specified in view coordinates, which are client coordinates offset by the scroll position.

To retrieve and set an item's position, use the LVM_GETITEMPOSITION and LVM_SETITEMPOSITION messages, respectively. LVM_GETITEMPOSITION works for all views, but LVM_SETITEMPOSITION works only for icon and small icon views.

You can determine which item, if any, is at a particular location by using the LVM_HITTEST message. To get the bounding rectangle for a list item, or for only its icon or label, use the LVM_GETITEMRECT message.

Unless the LVS_NOSCROLL window style is specified, you can use messages to perform a variety of scrolling operations. You can scroll a list view control to show items that do not fit in the client area of the control, determine a list view control's scroll position, scroll a list view control by a specified amount, or scroll a list view control so that a specified list item is visible.

In icon view or small icon view, the current scroll position is defined by the view origin. The view origin is the set of coordinates, relative to the visible area of the list view control, that corresponds to the view coordinates (0, 0). To get the current view origin, use the LVM_GETORIGIN message. This message should be used only in icon or small icon view; it returns an error in list or report view.

In list or report view, the current scroll position is defined by the top index. The top index is the index of the first visible item in the list view control. To get the current top index, use the LVM_GETTOPINDEX message. This message returns a valid result only in list view or report view; it returns zero in icon or small icon view.

Use the LVM_GETVIEWRECT message to get the bounding rectangle of all items in a list view control relative to the visible area of the control.

The LVM_GETCOUNTPERPAGE message returns the number of items that fit in one page of the list view control. This message returns a valid result only in list and report views; in icon and small icon views, it returns the total number of items.

To scroll a list view control by a specific amount, use the LVM_SCROLL message. Using the LVM_ENSUREVISIBLE message, you can scroll the list view control, if necessary, to ensure that a specified item is visible.

Editing Labels

A list view control that has the LVS_EDITLABELS window style enables a user to edit item labels in place. A user begins editing by clicking the label of an item that has the focus. An application can begin editing automatically by using the LVM_EDITLABEL message. The list view control notifies the parent window when editing begins, is canceled, or is completed. When editing is completed, the parent window is responsible for updating the item label, if appropriate.

When label editing begins, a list view control sends its parent window an LVN_BEGINLABELEDIT notification message. You can process this message to enable selective editing of specific labels; returning a nonzero value prevents label editing.

When label editing is canceled or completed, a list view control sends its parent window an LVN_ENDLABELEDIT notification message. The parent window is responsible for updating the item label if it keeps the new label.

During label editing, you can get the handle to the edit control used for label editing by using the LVM_GETEDITCONTROL message. To limit the amount of text a user can type, you can send the edit control an EM_LIMITTEXT message. You can even subclass the edit control to intercept and discard invalid characters.

Using Advanced List View Features

In Windows CE, you can set the order of the columns that display in report view by setting the *iOrder* member in the **LVCOLUMN** structure when you add a column to a list view control. You can also set the column order by using the LVM_GETCOLUMNORDERARRAY and LVM_SETCOLUMNORDERARRAY messages.

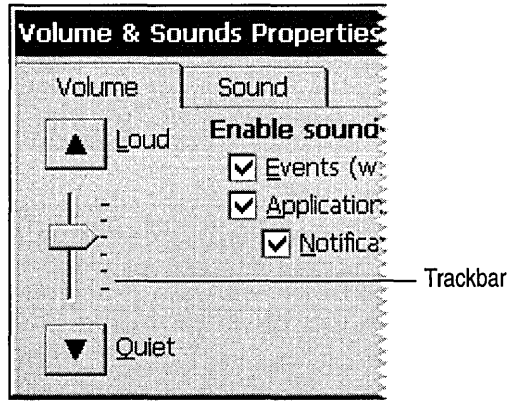
To display an image from an image list next to the title of a column in report view, specify LVCF_IMAGE in the *mask* member and LVCFMT_IMAGE in the *fmt* member. When you add a column to a list view control, specify the zero-based index of an image in the list in the *iImage* member of **LVCOLUMN**.

List view controls in Windows CE support a custom draw service, which gives you flexibility to customize the appearance of a list view. If a list view provides this service, it sends the NM_CUSTOMDRAW notification at specific times during drawing operations.

Windows CE supports a list view style, LVS_OWNERDATA, for creating a virtual list view. The only data that a virtual list view manages is input focus and item selection data. All other data is managed by the owner of the list view. This enables a list view to handle very large data sets, especially in cases where the data is stored in a database that has its own data access methods.

Creating a Trackbar

A *trackbar*, also known as a slider control, is a common 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, in one-tick increments. The following screen shot shows a Windows CE trackbar.



▶ **To create a trackbar**

1. Specify TRACKBAR_CLASS in the *lpClassName* parameter to the **CreateWindowEx** function.

   This class is registered when the common control DLL is loaded. You can use the **InitCommonControls** function to ensure that this DLL is loaded. To register the trackbar class using the **InitCommonControlsEx** function, specify the ICC_BAR_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

2. Specify a trackbar style using the *dwStyle* parameter of the **CreateWindowEx** function.

You can send messages to the trackbar to retrieve data about the window and to change its characteristics.

To retrieve the position of the slider, which is the value that a user has chosen, use the TBM_GETPOS message. To set the position of the slider, use the TBM_SETPOS message.

The range of a trackbar is the set of contiguous values that the trackbar can represent. Use the TBM_SETRANGE message to set the range of a trackbar when it is first created. You can dynamically alter the range by using the TBM_SETRANGEMAX and TBM_SETRANGEMIN messages. An application that accepts dynamic range changes retrieves the final range settings when a user has finished working with the trackbar. To retrieve these settings, use the TBM_GETRANGEMAX and TBM_GETRANGEMIN messages.

A trackbar automatically displays tick marks at each end, unless you specify the TBS_NOTICKS style. Use the TBS_AUTOTICKS style to automatically display additional tick marks at regular intervals along the trackbar. By default, a TBS_AUTOTICKS trackbar displays a tick mark at each increment of the trackbar's range. To specify a different interval for the automatic tick marks, send the TBM_SETTICFREQ message to the trackbar.

To set the position of a single tick mark, send the TBM_SETTIC message. A trackbar maintains an array of **DWORD** values that stores the position of each tick mark. The array does not include the first and last tick marks that the trackbar creates automatically. You can specify an index in this array when you send the TBM_GETTIC message to get the position of the corresponding tick mark. Alternatively, you can send the TBM_GETPTICS message to get a pointer to the array. To retrieve the physical position of a tick mark, send the TBM_GETTICPOS message. The TBM_CLEARTICS message removes all but the first and last of a trackbar's tick marks.

The trackbar line size determines how far the slider moves in response to keyboard input from the arrow keys, such as the RIGHT ARROW or DOWN ARROW key. To retrieve or set the line size, send the TBM_GETLINESIZE and TBM_SETLINESIZE messages, respectively.

The trackbar page size determines how far the slider moves in response to keyboard input, such as the PAGE UP or PAGE DOWN key, or mouse input, such as clicks in the trackbar channel. To retrieve or set the page size, send the TBM_GETPAGESIZE and TBM_SETPAGESIZE messages.

An application can send messages to retrieve the dimensions of a trackbar. The TBM_GETTHUMBRECT message retrieves the bounding rectangle for the slider. The TBM_GETTHUMBLENGTH message retrieves the length of the slider. The TBM_GETCHANNELRECT message retrieves the bounding rectangle for the trackbar's channel, which is the area over which the slider moves. If a trackbar has the TBS_FIXEDLENGTH style, you can send the TBM_SETTHUMBLENGTH message to change the length of the slider.

A trackbar with the TBS_ENABLESELRANGE style can indicate a selection range by highlighting a range of the trackbar channel and displaying triangular tick marks at the start and end of the selection. When a trackbar has this style, you can send messages to set and retrieve the selection range. Typically, an application handles the trackbar notification messages and sets the trackbar's selection range according to user input. The TBM_SETSEL message sets the starting and ending positions of a selection. To set just the starting position or just the ending position of a selection, use the TBM_SETSELSTART or TBM_SETSELEND message. To retrieve the starting and ending positions of a selection range, send the TBM_GETSELSTART and TBM_GETSELEND messages. To clear a selection range, send the TBM_CLEARSEL message.

Creating a Tree View

A *tree view control* is a hierarchical display of labeled items. Any item in a tree view control can have a list of subitems—called child items—associated with it. An item that has one or more child items is called a parent item. A child item is displayed below its parent item and is indented to indicate that it is subordinate to the parent. An item that has no parent appears at the top of the hierarchy and is called a root item.

▶ **To create a tree view control**

1. Specify the WC_ TREEVIEW class in the *lpClassName* parameter of the **CreateWindowEx** function.

   This class is registered when the common control DLL is loaded. You can use the **InitCommonControls** function to ensure that this DLL is loaded. To register the tree view class using the **InitCommonControlsEx** function, specify the ICC_ TREEVIEW_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

2. Specify a treeview style in the *dwStyle* parameter of the **CreateWindowEx** function.

   Tree view styles govern the appearance of a tree view control. You set the initial styles when you create the tree view control. You can retrieve and change the styles after creating the tree view control by using the **GetWindowLong** and **SetWindowLong** functions.

3. Add one or more items to the tree view control by sending the TVM_INSERTITEM message.

   The message returns a handle to the HTREEITEM type, which uniquely identifies the item. The message also includes a **TVINSERTSTRUCT** structure that specifies the handle to the parent item and the handle to the item after which the new item is to be inserted. When adding an item, specify the handle to the new item's parent item. If you specify NULL or the TVI_ROOT value instead of a parent item handle in the **TVINSERTSTRUCT** structure, the item is added as a root item. The second handle must identify either a child item of the specified parent or one of these values: TVI_FIRST, TVI_LAST, or TVI_SORT. When you specify TVI_FIRST or TVI_LAST, the tree view control places the new item at the beginning or end of the specified parent item's list of child items. When you specify TVI_SORT, the tree view control inserts the new item into the list of child items in alphabetical order based on the text of the item labels. You can put a parent item's list of child items in alphabetical order by using the TVM_SORTCHILDREN message. The TVM_SORTCHILDRENCB message enables you to sort child items based on criteria that you define. When you use this message, you specify an application-defined callback function that the tree view control can call when the relative order of two child items needs to be determined.

   Note A tree view control uses memory allocated from the heap of the process that creates the tree view control. The maximum number of items in a tree view is based on the amount of memory available in the heap.

At any time, the state of a parent item's list of child items can be either expanded, partially expanded, or collapsed. When the state is expanded, the child items of the expanded section are displayed below the parent item. When it is collapsed, the child items are not displayed. The list automatically toggles between the expanded and collapsed states when a user double-taps the parent item or, if the parent has the TVS_HASBUTTONS style, when a user clicks the button associated with the parent item. You can expand or collapse the child items by using the TVM_EXPAND message. A tree view control sends the parent window a TVN_ITEMEXPANDING notification message when a parent item's list of child items is about to be expanded or collapsed. The notification gives an application the opportunity to prevent the change or to set any attributes of the parent item that depend on the state of the list of child items. After changing the state of the list, the tree view control sends the parent window a TVN_ITEMEXPANDED notification message. When a list of child items is expanded, it is indented relative to the parent item. Set the amount of indentation by using the TVM_SETINDENT message or retrieve the current amount by using the TVM_GETINDENT message.

The following code example shows how to create a tree view control.

```
DWORD dwStyle;                  // Style flags of the tree view
INITCOMMONCONTROLSEX  iccex;  // INITCOMMONCONTROLSEX structure

// Initialize the INITCOMMONCONTROLSEX structure.
iccex.dwSize = sizeof (INITCOMMONCONTROLSEX);
iccex.dwICC = ICC_TREEVIEW_CLASSES;

// Register tree view control classes from the common control
// dynamic-link library (DLL).
InitCommonControlsEx (&iccex);

// Get the client area rectangle.
GetClientRect (hwnd, &rcClient);

// Create the command bar and insert menu.
g_hwndCB = CommandBar_Create (g_hInst, hwnd, 1);
CommandBar_InsertMenubar (g_hwndCB, g_hInst, IDR_MENU, 0);
CommandBar_AddAdornments (g_hwndCB, 0, 0);

// Get the command bar height.
iCBHeight = CommandBar_Height (g_hwndCB);

// Assign the window styles for the tree view.
dwStyle = WS_VISIBLE | WS_CHILD | TVS_HASLINES | TVS_LINESATROOT |
          TVS_HASBUTTONS;

// Create the tree view control.
g_hwndTreeView = CreateWindowEx (
    0,
    WC_TREEVIEW,           // Class name
    TEXT("TreeView"),      // Window name
    dwStyle,               // Window style
    0,                     // x coordinate of the upper left corner
    iCBHeight + 1,         // y coordinate of the upper left corner
    rcClient.right,        // The width of the edit control window
    rcClient.bottom - (iCBHeight + 1),
                           // The height of the edit control window
    hwnd,                  // Window handle of parent window
    (HMENU) IDC_TREEVIEW, // The treeview control identifier
    g_hInst,               // The instance handle
    NULL);                 // Specify NULL for this parameter when
                           // creating a control

// Be sure the tree view was actually created.
 if (!g_hwndTreeView)
  return 0;
```

Adding and Editing Item Labels

You typically specify the text of an item's label when you add the item to the tree view control. The TVM_INSERTITEM message includes a **TVITEM** structure that defines the item's properties, including a string containing the text of the label.

A tree view control allocates memory for storing each item; the text of the item labels takes up a significant portion of this memory. If you maintain a copy of the strings in the tree view control, you can decrease the memory requirements of the control by specifying the LPSTR_TEXTCALLBACK value in the *pszText* member of **TVITEM** instead of passing actual strings to the tree view. Using LPSTR_TEXTCALLBACK causes the tree view control to retrieve the text of an item's label from the parent window when the item needs to be redrawn.

A user can directly edit the labels of items in a tree view control that has the TVS_EDITLABELS style. A user begins editing by clicking the label of the item that has the focus. An application begins editing by using the TVM_EDITLABEL message. The tree view control notifies the parent window when editing begins and when it is canceled or completed. When a user or application completes editing, the parent window is responsible for updating the item's label, if appropriate.

When a user begins editing the label, a tree view control sends its parent window a TVN_BEGINLABELEDIT notification message. By processing this notification, an application can accept some label editing and reject editing others. Returning zero accepts editing, and returning nonzero rejects it.

When a user cancels or completes editing the label, a tree view control sends its parent window a TVN_ENDLABELEDIT notification message. The *pszText* member of TVITEM is zero if editing is canceled.

Modifying Tree View Item Appearance

Every tree view item has a current state that determines its appearance and features. You can retrieve and set this state by sending the TVM_GETITEM and TVM_SETITEM messages or by using the **TreeView_GetItem** and **TreeView_SetItem** macros. You set or retrieve the item state by using the *state* member of the **TV_ITEM** structure that you pass in the *pItem* parameter *(lParam)* to these messages and macros.

Windows CE supports the TVIS_EXPANDPARTIAL item state. This state indicates that a tree view item is partially expanded. This could happen if an error occurs during data retrieval and some of the child items cannot be retrieved from the data source. The tree view displays the items that were successfully retrieved, but continues to display the plus symbol next to the parent item as well. This indicates to a user that more information is available. When a user clicks the plus symbol again, the application repeats the query.

The following table shows item states supported by Windows CE.

State	Definition
TVIS_BOLD	Windows CE uses a bold font to draw the item.
TVIS_CUT	Windows CE selects the item for cutting and pasting.
TVIS_DROPHILITED	Windows CE selects the item for a drag-and-drop operation.
TVIS_EXPANDED	Windows CE expands the items list of child items so that the child items are visible. This state applies only to parent items.
TVIS_EXPANDEDONCE	Windows CE expands the item's list of child items at least once. Windows CE does not send the TVN_ITEMEXPANDING and TVN_ITEMEXPANDED notifications for parent items that have specified this value. This value applies only to parent items.
TVIS_EXPANDPARTIAL	Windows CE partially expands the items. This could happen if an error occurs during data retrieval and some of the child items cannot be retrieved from the data source.
TVIS_FOCUSED	Windows CE gives the item the focus and surrounds it with a standard focus rectangle. Although more than one item can be selected, only one item can have the focus.
TVIS_OVERLAYMASK	Windows CE includes the item's overlay image when it draws the image. The index of the overlay image must be specified in the *state* member of the **TV_ITEM** structure by using the **INDEXTOOVERLAYMASK** macro. The overlay image must be added to the tree view's image list by using the **ImageList_SetOverlayImage** function. This value should not be combined with any other value.
TVIS_SELECTED	Windows CE selects the item. The appearance of a selected item depends on whether it has the focus and on whether the system colors are used for selection.
TVIS_STATEIMAGEMASK	Windows CE includes the item's state image when it draws the item. The index of the state image must be specified in the *state* member of the **TV_ITEM** structure by using the **INDEXTOSTATEIMAGEMASK** macro. This value should not be combined with any other value.

Creating a Tree View Image List

Each item in a tree view control can have four bit images associated with it:

- An image such as an open folder, which appears when the item is selected.
- An image such as a closed folder, which is displayed when the item is not selected.
- An overlay image which is drawn transparently over the selected or unselected image.
- A state image, which is an additional image displayed to the left of the selected or unselected image. You can use state images, such as checked and cleared check boxes, to indicate application-defined item states.

By default, a tree view control does not display item images. To display item images, you must create image lists and associate them with the control.

A tree view control can have two image lists: a normal image list and a state image list. A normal image list stores the selected, unselected, and overlay images. A state image list stores state images.

To create an image list, call the **ImageList_Create** function, and use other image list functions to add bitmaps to the image list. Then, to associate the image list with the tree view control, use the TVM_SETIMAGELIST message. The TVM_GETIMAGELIST message retrieves a handle to one of a tree view control's image lists.

In addition to the selected and unselected images, a tree view control's normal image list can contain up to four overlay images. Overlay images are designed to be drawn transparently over the selected and unselected images. To assign an overlay mask index to an image in the normal image list, call the **ImageList_SetOverlayImage** function.

By default, all items display the first image in the normal image list for both the selected and unselected states. Also, by default, items do not display overlay images or state images. You can change these default behaviors for an item by sending the TVM_INSERTITEM or TVM_SETITEM messages. These messages use the **TVITEM** structure to specify image list indexes for an item.

To associate an overlay image with an item, use the **INDEXTOOVERLAYMASK** macro to specify an overlay mask index in the *state* member of the item's **TVITEM** structure. You must also set the TVIS_OVERLAYMASK bits in the *stateMask* member. Overlay mask indexes are one-based; an index of zero indicates that the application does not specify an overlay image.

To associate a state image with an item, use the **INDEXTOSTATEIMAGEMASK** macro to specify a state image index in the *state* member of the item's **TVITEM** structure. The index identifies an image in the control's state image list.

Note You can speed up the creation of large tree views by disabling the painting of the tree view before adding the items. You do this by sending a WM_SETREDRAW message with the redraw flag set to FALSE. When finished adding items, re-enable painting by sending a WM_SETREDRAW message with the redraw flag set to TRUE.

The following code example shows how to create and set an image list for a tree view control, and then redraw the control using the new images.

```
BOOL InitTreeViewImageLists (HWND hwndTreeView)
{
  HBITMAP hBmp;          // Handle of the bitmaps to be added.

  // Create the image list for the item pictures.
  if ((g_hImgList = ImageList_Create (CX_BITMAP, CY_BITMAP, ILC_MASK,
                                      NUM_BITMAPS, 0)) == NULL)
    return FALSE;

  // Load the bitmap resource.
  hBmp = LoadBitmap (g_hInst, MAKEINTRESOURCE (IDB_IMAGES));

  // Add the images to the image list, generating a mask from the
  // bitmap.
  if (ImageList_AddMasked (g_hImgList, hBmp, RGB (0, 255, 0)) == -1)
  {
    return FALSE;
  }

  // Delete the bitmap object and free system resources.
  DeleteObject (hBmp);

  // If not all the images were added, then return.
  if (ImageList_GetImageCount (g_hImgList) < NUM_BITMAPS)
    return FALSE;

  // Set the image list for the tree view control and redraw the
  // control by using the new images.
  TreeView_SetImageList (hwndTreeView, g_hImgList, TVSIL_NORMAL);

  return TRUE;
}
```

Handling Tree View Messages and Notifications

A tree view control notifies the parent window when the selection changes from one item to another by sending the TVN_SELCHANGING and TVN_SELCHANGED notification messages. The notifications also include data about the item that gains the selection and the item that loses the selection. You can use this data to set item attributes that depend on the selection state of the item. Returning TRUE in response to TVN_SELCHANGING rejects the selection change and returning FALSE accepts the selection change. Change the selection by sending the TVM_SELECTITEM message.

Tree view controls support a number of messages that retrieve data about items in the control.

The TVM_GETITEM message can retrieve an item's handle and attributes. An item's attributes include its current state, the indexes in the control's image list of the item's selected and unselected bit images, a flag that indicates whether the item has child items, the address of the item's label string, and the item's application-defined 32-bit value.

The TVM_GETNEXTITEM message retrieves the tree view item that bears the specified relationship to the current item. The message can retrieve an item's parent, the next or previous visible item, the first child item, and so on.

The TVM_GETITEMRECT message retrieves the bounding rectangle for a tree view item. The TVM_GETCOUNT and TVM_GETVISIBLECOUNT messages retrieve a count of the items in a tree view control and a count of the items that can be fully visible in the tree view control's window, respectively. You can ensure that a particular item is visible by using the TVM_ENSUREVISIBLE message.

Handling Drag-and-Drop Operations

A tree view control notifies the parent window when a user starts to drag an item with a mouse. The parent window receives a TVN_BEGINDRAG notification message when a user begins dragging an item with the left mouse button and a TVN_BEGINRDRAG notification message when a user begins dragging with the right button. You can prevent a tree view control from sending these notifications by giving the tree view control the TVS_DISABLEDRAGDROP style.

You obtain an image to display during a drag operation by using the TVM_CREATEDRAGIMAGE message. The tree view control creates a dragging bitmap based on the label of the item being dragged. Then, the tree view control creates an image list, adds the bitmap to it, and returns the handle to the image list.

You must provide the code that actually drags the item. This typically involves using the dragging capabilities of the image list functions and including code for processing the WM_MOUSEMOVE and WM_LBUTTONUP messages sent to the parent window after the drag operation begins.

To use an item in a tree view control as the target of a drag-and-drop operation, use the **SendMessage** function to send a TVM_HITTEST message to determine when the stylus is on a target item. To do this, specify the address of a **TVHITTESTINFO** structure that contains the current coordinates of the stylus. When the **SendMessage** function returns, the structure contains a flag indicating the location of the stylus relative to the tree view control. If the stylus is over an item in the tree view control, the structure contains the handle to the item as well.

You indicate that an item is the target of a drag-and-drop operation by using the TVM_SETITEM message to set the state to TVIS_DROPHILITED. An item that has this state is drawn in the style used to indicate a target for a drag-and-drop operation.

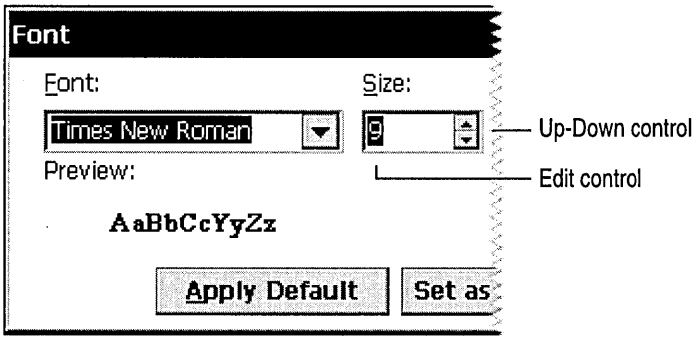
The following code example shows how to handle drag-and-drop messages.

```
case WM_NOTIFY:
{
  LPNMHDR  pnmh = (LPNMHDR) lParam;

  switch (pnmh->code)
  {
    case TVN_BEGINDRAG:
        {
            // Notifies the tree view control's parent window that a
            // drag-and-drop operation is being initiated.

            return 0;
        }

        case TVN_BEGINLABELEDIT:
        {
            // Notifies the tree view control's parent window about the
            // start of label editing for an item.

            return 0;
        }
```

```
    case TVN_ITEMEXPANDED:
    {
      // Notifies a tree view control's parent window that a parent
      // item's list of child items has expanded or collapsed. This
      // notification message is sent in the form of a WM_NOTIFY
      // message.

      return 0;
    }

    case TVN_ITEMEXPANDING:
    {
      // Notifies a tree view control's parent window that a parent
      // item's list of child items is about to expand or collapse.

      return 0;
    }

    default:
      return 0;
  }
  break;
}
```

Creating an Up-Down Control

An *up-down control* is a pair of arrow buttons that a user can tap with the stylus to increment or decrement a value. An up-down control may be used in one of two ways: as a stand-alone control or in conjunction with another control, called a buddy window. In Windows CE–based applications, up-down controls can be "buddies" only with edit controls. When an up-down control is paired with an edit control, it is called a *spin control*. The following screen shot shows an up-down control and buddy window.



You create an up-down control by using the **CreateUpDownControl** function. This class is registered when the common control DLL is loaded. You can use the **InitCommonControls** function to ensure that this DLL is loaded.

To register the up-down control class using the **InitCommonControlsEx** function, specify the ICC_UPDOWN_CLASS flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

The following code example is the syntax for the **CreateUpDownControl** function.

```
HWND CreateUpDownControl (DWORD dwStyle,
                          int x,
                          int y,
                          int cx,
                          int cy,
                          HWND hParent,
                          int nID,
                          HINSTANCE hInst,
                          HWND hBuddy,
                          int nUpper,
                          int nLower,
                          int nPos);
```

Here, *dwStyle* specifies the style of the up-down control. You must include the WS_CHILD, WS_BORDER, and WS_VISIBLE styles, and it may include any of the window styles specific to the up-down control.

The upper-left corner of a control is positioned at *x*, *y*, and its dimension is determined by *cx* and *cy*. The handle to the parent window is passed in *hParent* and the control identifier is specified in *nID*. The handle to the application is passed in *hInst*. The handle to the buddy window is passed in *hBuddy*. If there is no buddy window, this parameter must be NULL.

The upper-limit range of the control is passed in *nUpper* and the lower-limit range is passed in *nLower*. The initial position of the control is specified in *nPos*. This value must be within the specified range.

Modifying Control Position and Acceleration

After you have created an up-down control, you can change it in several ways. You can change its current position, minimum position, and maximum position by sending messages. You can change the radix base, that is, either base 10 or base 16, used to display the current position in the buddy window. You can change the rate at which the current position changes when the up or down arrow is tapped.

To retrieve the current position of an up-down control, use the UDM_GETPOS message. For an up-down control with a buddy window, the current position is the number in the buddy window's caption. The up-down control retrieves the current caption and updates its current position if the caption has changed because a user edited the text of an edit control.

The buddy window's caption can be either a decimal or hexadecimal string, depending on the radix base of the up-down control. Set the radix base by using the UDM_SETBASE message and retrieve the radix base by using the UDM_GETBASE message.

The UDM_SETPOS message sets the current position of a buddy window. Note that unlike a scroll bar, an up-down control automatically changes its current position when the up and down arrows are tapped. Therefore, an application does not need to set the current position when processing the WM_VSCROLL or WM_HSCROLL message.

You can change the minimum and maximum positions of an up-down control by using the UDM_SETRANGE message. The maximum position may be less than the minimum, in which case tapping the up arrow button decreases the current position. Put another way, up moves toward the maximum position. To retrieve the minimum and maximum positions for an up-down control, use the UDM_GETRANGE message.

You can control the rate at which the position changes when a user holds down an arrow button by setting the up-down control's acceleration. The acceleration is defined by an array of **UDACCEL** structures. Each structure specifies a time interval and the number of units by which to increment or decrement at the end of that interval. To set the acceleration, use the UDM_SETACCEL message. To retrieve acceleration data, use the UDM_GETACCEL message.

Creating a Date and Time Picker Control

The *date and time picker* is a control that displays information about dates and times and provides users with an easy way to modify this information. Each field in the control displays a time element, such as month, day, hour, or minute. A user selects a field by tapping it with the stylus and then types a new value from the keyboard.

The way date and time information is displayed is determined by a format string. A date and time picker control can display time information in any of three preset formats, or you can create custom format strings to specify a different order in which to display the fields. You can also add customized date and time information to a date and time picker control by using callback fields.

▶ **To create a date and time picker control**

1. Specify DATETIMEPICK_CLASS in the *lpClassName* parameter of the **CreateWindowEx** function.

   This class is registered when the common control DLL is loaded. You can use the **InitCommonControls** function to ensure that this DLL is loaded. To register the date and time picker class using the **InitCommonControlsEx** function, specify the ICC_DATE_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

2. Specify a date and time picker style in the *dwStyle* parameter of the **CreateWindowEx** function. The following screen shot shows the Windows CE date and time picker.

Displaying Information

As stated earlier, a date and time picker control relies on a format string to determine how it will display fields of information. By default, a date and time picker control can display time information fields in three preset formats or according to a custom format string. Custom format strings provide flexibility for your application. In a custom format string, you can specify the order in which the control will display fields of information or indicate specific callback fields. The format characters of the format string define the display and field layout for the date and time picker control.

The following list shows Window styles used by the preset formats, which are format strings.

DTS_LONGDATEFORMAT
: The control displays the date in long format. The default format string for this style is defined by LOCALE_SLONGDATEFORMAT, which produces the output "Friday, April 19, 1998."

DTS_SHORTDATEFORMAT
: The control displays the date in short format, which is the default style setting. The default format string for this style is defined by LOCALE_SSHORTDATE, which produces the output "4/19/98."

DTS_TIMEFORMAT
: The control displays the time. The default format string for this style is defined by LOCALE_STIMEFORMAT, which produces the output "5:31:42 PM." An up-down control is placed to the right of the date and time picker control to modify time values.

You can customize the display of a date and time picker control using custom format strings. Date and time picker controls support specified format characters that you can combine to create a format string. To assign the format string to the date and time picker control, use the DTM_SETFORMAT message.

The following table shows format characters supported by date and time picker controls.

String fragment	Description
d	The one-digit or two-digit day.
dd	The two-digit day. Single-digit day values are preceded by a zero.
ddd	The three-character weekday abbreviation.
dddd	The full weekday name.
gg	The period and era string contained in the CAL_SERASTRING value associated with the specified locale. Windows CE ignores this element if the date to be formatted does not have an associated era or period string.
h	The one-digit or two-digit hour in 12-hour format.
hh	The two-digit hour in 12-hour format. Single-digit values are preceded by a zero.
H	The one-digit or two-digit hour in 24-hour format.
HH	The two-digit hour in 24-hour format. Single-digit values are preceded by a zero.
m	The one-digit or two-digit minute.
mm	The two-digit minute. Single-digit values are preceded by a zero.
M	The one-digit or two-digit month number.

String fragment	Description
MM	The two-digit month number. Single-digit values are preceded by a zero.
MMM	The three-character month abbreviation.
MMMM	The full month name.
t	The one-letter A.M. and P.M. abbreviation (that is, "AM" is displayed as "A").
tt	The two-letter A.M. and P.M. abbreviation (that is, "AM" is displayed as "AM").
X	The callback field. The control uses the other valid format characters and queries the application to fill in the "X" portion of the string. The application must be prepared to handle the DTN_WMKEYDOWN, DTN_FORMAT, and DTN_FORMATQUERY notification messages. Multiple "X" characters can be used in a series to signify unique callback fields.
y	The year is displayed as the last two digits, but with no leading zero for years less than 10.
yy	The last two digits of the year. For example, 1998 would be displayed as "98."
yyy	The full year. For example, 1998 would be displayed as "1998."

You can add body text to the format string. For example, if you want the control to display the current date with the format "Today is: 04:22:31 Tuesday Mar 23, 1998," use the following format string: Today is: 'hh': 'm': 's ddddMMMdd', 'yyy'. The window procedure for a command bar automatically sets the size of the command bar and positions it along the top of the parent window client area. It also destroys the command bar when its parent window is destroyed. Body text must be enclosed in single quotation marks.

Note that segments of nonformat characters in the preceding example are delimited by single quotation marks. Failure to surround body text in this way will result in unpredictable display by the date and time picker control.

Customizing Output with Callback Fields

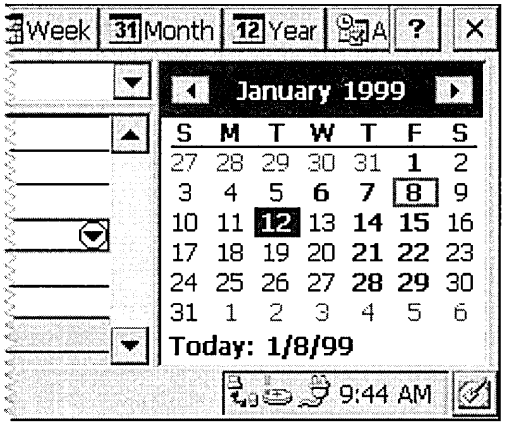
In addition to the standard format characters that define date and time picker fields, you can customize your output by specifying certain parts of a format string as callback fields. To declare a callback field, include one or more ASCII Code 88 "X" characters anywhere in the body of the format string. Like other date and time picker control fields, callback fields are displayed in left-to-right order, based on their location in the format string.

You can create unique callback fields by repeating the "X" character. Thus, the following format string contains two callback fields: 'XXddddMMMdd', 'yyyXXX'. Remember, because callback fields are treated as valid fields, your application must be prepared to handle DTN_WMKEYDOWN notification messages.

When the date and time picker control parses the format string and encounters a callback field, it sends DTN_FORMAT and DTN_FORMATQUERY notification messages. The owner of the control must respond to these notifications to ensure that the custom information is properly displayed.

Creating a Month Calendar Control

A *month calendar control* is a child window that displays a monthly calendar. The calendar can display one or more months at a time. The following screen shot shows a month calendar control.



When a user taps the name of a month with the stylus, a pop-up menu appears that lists all the months of the year. A user can select a month by tapping its name on the menu. A user who is using the date and time picker control can use ALT+DOWN ARROW to activate the month calendar control. A user can scroll the displayed months forward or backward either by tapping the left arrow or right arrow at the top of the control or by pressing the PAGE UP or PAGE DOWN keys on the keyboard. When a user taps the year displayed at the top of the calendar next to the month, an up-down control appears. A user can use this control to change the year. A user can also use CTRL+PAGE UP or CTRL+PAGE DOWN to scroll from one year to another. A user can press keys on the keyboard to navigate; the arrow keys scroll between days, the HOME key moves to the beginning of a month, and the END key moves to the end of a month. Unless the calendar has the MCS_NOTODAY style, a user can return to the current day by tapping the **Today** label at the bottom of the month calendar control.

▶ **To create a month calendar control**

1. Specify MONTHCAL_CLASS in the *lpClassName* parameter of the **CreateWindowEx** function.

   This class is registered when the common control DLL is loaded. You can use the **InitCommonControls** function to ensure that this DLL is loaded. To register the date and time picker class using the **InitCommonControlsEx** function, specify the ICC_DATE_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

2. Specify a date and time picker style in the *dwStyle* parameter of the **CreateWindowEx** function.

   A month calendar control that uses the MCS_DAYSTATE style supports day states. The control uses day state data to determine how it draws specific days within the control. Day state data is expressed as a 32-bit data type, **MONTHDAYSTATE**. Each bit in a **MONTHDAYSTATE** bit field, from 1 through 31, represents the state of a day in a month. If a bit is on, the corresponding day will be displayed in bold; otherwise, it will be displayed with no emphasis. An application can explicitly set day state data by sending the MCM_SETDAYSTATE message or by using the corresponding macro, **MonthCal_SetDayState**. Additionally, month calendar controls that use the MCS_DAYSTATE style send MCN_GETDAYSTATE notification messages to request day state data.

Setting the Time

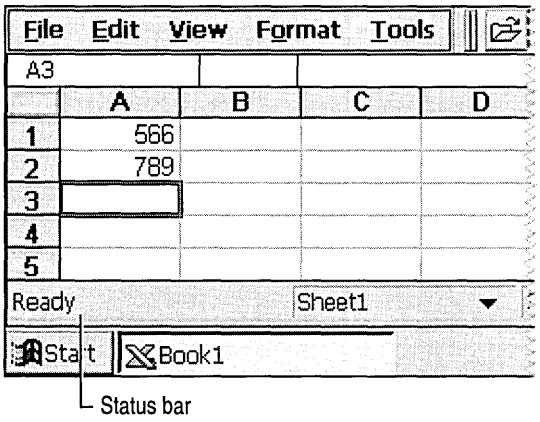
Because the month calendar control is created, it will insert the current time into its "today" date and time. Later, when a time is set programmatically, the control will either copy the time fields as they are or validate them first, and then, if invalid, store the current default time. The following table shows messages that set a date, and the manner in which those messages affect time fields.

Message	Description
MCM_SETCURSEL	The control copies the time fields as they are, without validation or modification.
MCM_SETRANGE	The control validates the time fields of the structures passed in. If valid, the time fields are copied without modification. If invalid, the control copies the time fields from the "today" date and time.
MCM_SETSELRANGE	The control validates the time fields of the structures passed in. If valid, the time fields are copied without modification. If invalid, the control retains the time fields from the current selection ranges.
MCM_SETTODAY	The control copies the time fields as they are, without validation or modification.

When a date is retrieved from the month calendar control, the time fields will be copied from the stored times without modification. Handling of the time fields by the control is provided as a convenience. The control does not examine or modify the time fields as a result of any operations other than those previously listed.

Creating a Status Bar

A *status bar*, also known as a status window, is a horizontal window positioned at the bottom of a parent window. It displays status information defined by the application. The following screen shot shows a status bar.



You create a status bar by calling the **CreateStatusWindow** function. This class is registered when the common control DLL is loaded. You can use the **InitCommonControls** function to ensure that this DLL is loaded. To register the status bar class using the **InitCommonControlsEx** function, specify the ICC_BAR_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.

Because status bars are windows, you can create a status bar by calling **CreateWindow** or **CreateWindowEx** and specifying the window class **STATUSCLASSNAME**.

The window procedure for the status bar control automatically sets the initial size and position of the window. The width is the same as that of the parent window's client area. The height is based on the width of the window's borders and on the metrics of the font currently selected into the status bar's device context.

The window procedure automatically adjusts the size of the status bar when it receives a WM_SIZE message. Typically, when the size of the parent window changes, the parent sends a WM_SIZE message to the status bar.

An application can set the minimum height of a status bar drawing area by sending the window an SB_SETMINHEIGHT message that specifies the minimum height in pixels. The drawing area does not include the window borders.

You retrieve the widths of the borders of a status bar by sending the window an SB_GETBORDERS message. The message includes the address of a three-element array that receives the widths.

Creating a Multiple-Part Status Bar

A status bar can have many different parts, each displaying a different line of text. You divide a status bar into parts by sending the window an SB_SETPARTS message, which specifies the number of parts to create and the address of an integer array. The array contains one element for each part, and each element specifies the client coordinate of the right edge of a part.

A status bar can have a maximum of 255 parts, although applications typically use fewer. You retrieve a count of the parts in a status bar, as well as the coordinate of the right edge of each part, by sending the window an SB_GETPARTS message.

A simple mode status bar is useful for displaying Help text for menu items while a user scrolls through the menu. You put a status bar in simple mode by sending it an SB_SIMPLE message. A simple mode status bar displays only one part. When the text of the window is set, the window is invalidated but is not redrawn until the next WM_PAINT message. Waiting for the message reduces screen flicker by minimizing the number of times the window is redrawn.

The string that a status bar displays while in simple mode is maintained separately from the strings that it displays while it is not in simple mode. This means that you can put the window in simple mode, set its text, and switch out of simple mode without the original text being changed.

Windows CE supports a status bar notification, SBN_SIMPLEMODECHANGE, that a status bar sends when the simple mode changes as a result of receiving an SB_SIMPLE message.

Adding Status Bar Text

You set the text of any part of a status bar by sending the SB_SETTEXT message, specifying the zero-based index of a part, an address of the string to draw in the part, and the technique for drawing the string. The drawing technique determines whether the text has a border and, if it does, the style of the border. It also determines whether the parent window is responsible for drawing the text.

By default, text is left-aligned within the specified part of a status bar. You can embed tab characters, such as **\ t**, in the text to center it or right-align it. Text to the right of a single tab character is centered, and text to the right of a second tab character is right-aligned.

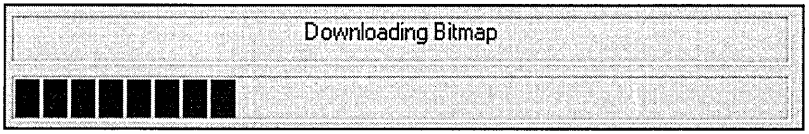
To retrieve text from a status bar, use the SB_GETTEXTLENGTH and SB_GETTEXT messages.

If your application uses a status bar that has only one part, you can perform text operations by using the WM_SETTEXT, WM_GETTEXT, and WM_GETTEXTLENGTH messages. These messages deal only with the part that has an index of zero, enabling you to treat the status bar much like a static text control.

To display a line of status information without creating a status bar, use the **DrawStatusText** function. The function uses the same techniques to draw the status information as it uses to draw the window procedure for the status bar, but it does not automatically set the size and position of the status information. When calling the **DrawStatusText** function, you must specify the size and position of the status information as well as the device context of the window in which to draw it.

Creating a Progress Bar

A *progress bar* is 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. The following screen shot shows a progress bar.



▶ **To create a progress bar**

1. Specify PROGRESS_CLASS in the *lpClassName* parameter of the **CreateWindowEx** function.

   This class is registered when the common control DLL is loaded. You can use the **InitCommonControls** function to ensure that this DLL is loaded. To register the progress bar class using the **InitCommonControlsEx** function, specify the ICC_PROGRESS_CLASS flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.
2. Specify a progress bar style in the *dwStyle* parameter of the **CreateWindowEx** function.

Setting the Range and Current Position

A progress bar's range represents the entire duration of the operation, and the current position represents the progress that the application has made toward completing the operation. The window procedure uses the range and the current position to determine the percentage of the progress bar to fill with the highlight color as well as to determine what text, if any, to display within the progress bar.

If you do not set the range values, the system sets the minimum value to zero and the maximum value to 100. You can adjust the range to convenient integers by using the PBM_SETRANGE message.

A progress bar provides several messages that you can use to set the current position. The PBM_SETPOS message sets the position to a specified value. The PBM_DELTAPOS message advances the position by adding a specified value to the current position. The PBM_SETSTEP message enables you to specify a step increment for a progress bar. Subsequently, when you send the PBM_STEPIT message to the progress bar, the current position advances by the specified increment. The default step increment is 10.

Note The range values in a progress bar are considered signed integers. Any number greater than 0x7FFFFFFF is interpreted as a negative number.

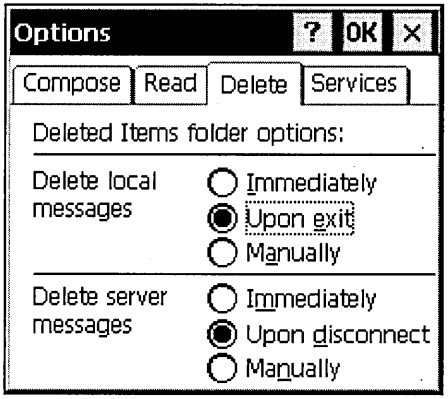
Creating a Property Sheet

A *property sheet* is a system-defined dialog box that you use to view or modify object attributes or properties. A property sheet includes a frame, a title bar, and three buttons: **OK**, **Cancel** (**X**), and **Help** (**?**), located atop the window. To use property sheets, you must include the Prsht.h header file in your application.

A property sheet contains and manages one or more related dialog boxes, called property pages. Each page in a property sheet is an application-defined modeless dialog box that manages the controls that enable a user to view and edit the properties of an object. A property sheet must contain at least one property page, but cannot contain more than the value of **MAXPROPPAGES** as defined in the header files.

Users access property sheets by using an ALT+Tap action. A property sheet sends a notification message to the dialog box procedure for a page when the page becomes active or inactive and when a user taps the **OK**, **Cancel** (**X**), or **Help** (**?**) button. The notifications are sent in the form of WM_NOTIFY messages. The *lParam* parameter of the WM_NOTIFY messages points to an **NMHDR** structure, which includes the window handle of the property sheet dialog box. Some notification messages require that a property sheet page return either TRUE or FALSE in response to the WM_NOTIFY message. To respond, the page must use the **SetWindowLong** function to set the DWL_MSGRESULT value for the page dialog box to either TRUE or FALSE.

Each page has a corresponding label, which the property sheet displays in the tab that it creates for the page. Because all property sheet pages expect you to use a roman font, and not bold, you must ensure that the font is not bold by specifying the DS_3DLOOK style in the dialog box template. The following screen shot shows a Windows CE property sheet.



Before creating a property sheet, you must define one or more pages.

▶ **To define a property sheet page**

1. Create a **PROPSHEETPAGE** structure that contains data about a property sheet icon, label, dialog box template, dialog box procedure, and other attributes.
2. Call the **CreatePropertySheet** function and pass it a pointer to the **PROPSHEETPAGE** structure. The function returns a **HPROPSHEETPAGE** handle to the property page.

Once you have defined one or more property sheet pages, you can create a property sheet. One way to create a property sheet is to specify the address of a **PROPSHEETHEADER** structure in a call to the **PropertySheet** function. The structure defines the icon and title for the property sheet and also includes a pointer to an array of **HPROPSHEETPAGE** handles. When **PropertySheet** creates the property sheet, it includes the pages identified in the array. The order of the array determines the order of the pages in the property sheet.

Another method to create a property sheet is to specify an array of **PROPSHEETHEADER** structures instead of an array of **HPROPSHEETPAGE** handles. In this case, **PropertySheet** creates handles for the pages before adding them to the property sheet.

The **PropertySheet** function automatically sets the size and initial position of a property sheet. The position is based on the position of the owner window, and the size is based on the largest page specified in the array of pages when the property sheet is created.

After creating a property sheet, you can add and remove pages by using the PSM_ADDPAGE message. Note that the size of the property sheet cannot change after it has been created, so the new page must be no larger than the largest page currently in the property sheet. To remove a page, use the PSM_REMOVEPAGE message. When you define a page, you can specify the address of the **PropSheetPageProc** callback function that the property sheet calls when it creates or removes the page. Using **PropSheetPageProc** enables you to initialize individual property sheet pages.

To destroy a page that was created by the **CreatePropertySheetPage** function, but was not added to the property sheet, use the **DestroyPropertySheetPage** function. Destroying a property sheet automatically destroys all pages that have been added. The system destroys the pages in reverse order from that specified in the array used to create the pages.

You specify the title of a property sheet in the **PROPSHEETHEADER** structure that you used to create the property sheet. If the *dwFlags* member includes the PSH_PROPTITLE value, the property sheet adds the prefix "Properties" to the specified title string. Use the PSM_SETTITLE message to change the title after a property sheet has been created.

By default, a property sheet uses the name string specified in the dialog box template as the label for the property page sheet page. You can override the name string by including the PSP_USETITLE value as the *dwFlags* member of the **PROPSHEETPAGE** structure that defines the page. When PSP_USETITLE is specified, the *pszTitle* member must contain the address of the label string for the page.

Working with Active and Inactive Property Sheet Pages

A property sheet can have only one active page at a time. The active sheet is at the top of the overlapping stack of pages. A user activates a page by selecting its tab; an application uses the PSM_SETCURSEL message to activate a page. Before the subsequent active page is visible, the property sheet sends it the PSN_SETACTIVE notification message. The page should respond by initializing its control windows.

The property sheet determines whether to enable or disable the **Help** button for an active page by checking for the PSP_HASHELP style. If the page has this style, it supports the **Help** button. If the PSP_HASHELP style is not present, it disables the button. When a user taps the **Help** button, the active page receives the PSN_HELP notification message. The page should respond by displaying Help information.

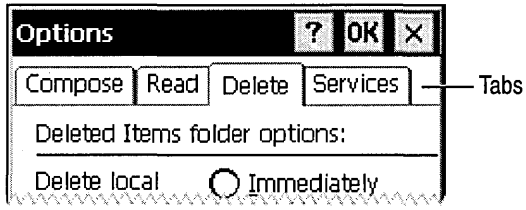
When a user taps **OK**, the property sheet sends the PSN_KILLACTIVE notification message to the active page, giving it an opportunity to validate a user's changes. If the page determines that the changes are valid, it should call the **SetWindowLong** function to set the DWL_MSGRESULT value for the page to FALSE. In this case, the property sheet sends the PSN_APPLY notification message to each page, directing it to apply the new properties to the corresponding item. If the page determines that a user's changes are not valid, it should set DWL_MSGRESULT to TRUE and display a dialog box informing a user of the problem. The page remains active until it sets DWL_MSGRESULT to FALSE in response to a PSN_KILLACTIVE message.

The property sheet sends the PSN_RESET notification message to all pages when a user taps the **Cancel** button, indicating that it is about to destroy the property sheet.

Note To set the position of a property sheet window in an application, use the **SetWindowPos** function rather than the **MoveWindow** function. Call **SetWindowPos** in the dialog box procedure of the property page that will open first when a user activates a property sheet.

Creating a Tab Control

A *tab control* is analogous to a set of dividers in a notebook or labels in a file cabinet. In a property sheet, a user selects a tab to move from one property sheet page to another. The following screen shot shows a Windows CE tab control.



You send messages to a tab control to add tabs and otherwise affect the appearance and behavior of the control. Each message has a corresponding macro, which you can use instead of sending the message explicitly. Though you cannot disable an individual tab in a tab control, you can disable a tab control in a property sheet by disabling the corresponding page.

Each tab in a tab control consists of a label and application-defined data. This data is specified by a **TCITEM** structure. You can add tabs to a tab control, get the number of tabs, retrieve and set the contents of a tab, and delete tabs. Tabs are identified by their zero-based index.

Windows CE supports two extended tab control styles. The first style uses the TCM_SETEXTENDEDSTYLE message or its corresponding macro, **TabCtrl_SetExtendedStyle**, to set the extended style. It uses the TCM_GETEXTENDEDSTYLE message or its corresponding macro, **TabCtrl_GetExtendedStyle**, to retrieve the extended style.

Note Because an extended tab control style is not the same as an extended window style, you cannot pass an extended tab control style to **CreateWindowEx** when you create a tab control.

The second extended tab control style, TCS_EX_FLATSEPARATORS, draws a separator between tab items in tab controls that have the TCS_BUTTONS or TCS_FLATBUTTONS style. When you create a tab control with the TCS_FLATBUTTONS style, this extended style is set by default.

▶ **To create a tab control**

1. Specify the WC_TABCONTROL class in the *lpClassName* parameter of the **CreateWindowEx** function.

   Windows CE registers this class when it loads the common control DLL. You can use the **InitCommonControls** function to ensure that this DLL is loaded. To register the tab control class using the **InitCommonControlsEx** function, specify the ICC_ TAB_CLASSES flag as the *dwICC* member of the **INITCOMMONCONTROLSEX** structure you pass in the *lpInitCtrls* parameter.
2. Specify a tab control style in the *dwStyle* parameter of the **CreateWindowEx** function.

You can add tabs to the control using the TCM_INSERTITEM message, which specifies the position of the tab and the address of its **TCITEM** structure. You can retrieve and set the contents of an existing tab by using the TCM_GETITEM and TCM_SETITEM messages. For each tab, you can specify an icon, a label, or both. You can also specify application-defined data to associate with the tab.

You can associate application-defined data with each tab. For example, you might save information about each page with its corresponding tab. By default, a tab control allocates four extra bytes per tab for application-defined data. You can change the number of extra bytes per tab by using the TCM_SETITEMEXTRA message. You can use this message only when the tab control is empty.

The *lParam* member of the **TCITEM** structure specifies application-defined data. If you use more than four bytes of application-defined data, you need to define your own structure and use it instead of **TCITEM**. You can retrieve and set application-defined data the same way you retrieve and set other information about a tab: Use the TCM_GETITEM and TCM_SETITEM messages.

Note Windows CE does not support vertical text. If you create vertical tabs and want to use vertical text, you have to create a text bitmap and rotate it. Then, you can add the bitmap to an image list and attach it to the tab by specifying its image list index in the *iImage* member of the **TCITEM** or **TCITEMHEADER** structure.

Handling Tab Control Messages

When a user selects a tab, a tab control sends notification messages to its parent window in the form of WM_NOTIFY messages. The tab control sends the TCN_SELCHANGING notification message before the selection changes, and it sends the TCN_SELCHANGE notification message after the selection changes.

You can process TCN_SELCHANGING to save the state of the outgoing page. You can return TRUE to prevent the selection from changing. For example, you might not want to switch away from a child dialog box in which a control has an invalid setting.

To display the incoming page in the display area, you must process TCN_SELCHANGE. Though processing might include changing the information displayed in a child window, it will more likely entail destroying or hiding the outgoing child window or dialog box and creating or showing the incoming child window or dialog box.

You can retrieve and set the current table selection by using the TCM_GETCURSEL and TCM_SETCURSEL messages.

Adding a Tab Control Image List

Each tab can have an icon associated with it. An index specifies the icon into the image list for the tab control. When you create a tab control, it has no image list associated with it. You can use the **ImageList_Create** function to create an image list. You can assign it to a tab control by using the TCM_SETIMAGELIST message.

You can add an image to a tab control's image list just as you would add one to any other image list. To ensure that each tab remains associated with its assigned image, remove images by using the TCM_REMOVEIMAGE message instead of the **ImageList_Remove** function.

Because destroying a tab control does not destroy an associated image list, you must destroy the image list separately. Retaining the image list might be useful if you want to assign the same image list to multiple tab controls.

To retrieve the handle of the image list currently associated with a tab control, you can use the TCM_GETIMAGELIST message.

Setting Tab Size and Position

The tab control display area is the area of the control in which an application displays the current page. An application creates a child window or dialog box to display the current page, and then it sets the window size and position to fit the display area. You can use the TCM_ADJUSTRECT message to calculate a tab control's display area based on the dimensions of a specified rectangle or to calculate the dimensions of a rectangle given the coordinates of a display area.

Each tab in a tab control has a size and a position. You can set the size of tabs, retrieve the bounding rectangle of a tab, or determine which tab is located at a specified position.

For fixed-width and owner-drawn tab controls, you can set the exact width and height of tabs by using the TCM_SETITEMSIZE message. In other tab controls, you calculate each tab's size based on the icon and label for the tab. The tab control includes space for a border and an additional margin. You can set the thickness of the margin by using the TCM_SETPADDING message.

You use messages and styles to learn about tabs. You can determine the current bounding rectangle for a tab by using the TCM_GETITEMRECT message. You can determine which tab, if any, is at a specified location by using the TCM_HITTEST message. In a tab control with the TCS_MULTILINE style, you can determine the current number of rows of tabs by using the TCM_GETROWCOUNT message.

Using the Custom Draw Service

Windows CE supports the custom draw service. The custom draw service is not a common control; it is a service that makes it easy to customize the appearance of a common control. You can use it to change a common control's color or font or to partially or completely draw the control.

A common control that supports the custom draw service provides this service by sending an NM_CUSTOMDRAW notification at specific times during drawing operations. The *lParam* of the NM_CUSTOMDRAW notification is a reference to an **NMCUSTOMDRAW** structure. If the control is a list view, it uses the **NMLVCUSTOMDRAW** structure; if the control is a tree view, it uses the **NMTVCUSTOMDRAW** structure. This structure contains data that the application can use to determine how to draw the control. The following common controls can provide the custom draw service:

- Command bands
- Header controls
- List views
- Toolbars
- Trackbars
- Tree views

Handling Paint Cycles, Drawing Stages, and Notification Messages

Common controls paint and erase themselves based on messages received from the OS or other applications. The process of a control painting or erasing itself is called a *paint cycle*. Controls that support custom draw send NM_CUSTOMDRAW notification messages periodically throughout each paint cycle. This notification message is accompanied by an **NMCUSTOMDRAW** structure or another structure that contains an **NMCUSTOMDRAW** structure as its first member.

In addition to other data, the **NMCUSTOMDRAW** structure informs the parent window about what stage of the paint cycle the control is in. This is referred to as the draw stage and is represented by the value in the structure's *dwDrawStage* member. A control informs its parent about four basic, or global, draw stages. The following table shows the flag values, defined in the Commctrl.h header file, that represent these stages in the structure.

Global draw stage value	Description
CDDS_PREPAINT	Before the paint cycle begins
CDDS_POSTPAINT	After the paint cycle is complete
CDDS_PREERASE	Before the erase cycle begins
CDDS_POSTERASE	After the erase cycle is complete

Each of the preceding values can be combined with the CDDS_ITEM flag to specify draw stages for items. The following table shows item-specific values contained in the Commctrl.h header file.

Item-specific draw stage value	Description
CDDS_ITEMPREPAINT	Before an item is drawn
CDDS_ITEMPOSTPAINT	After an item has been drawn
CDDS_ITEMPREERASE	Before an item is erased
CDDS_ITEMPOSTERASE	After an item has been erased

You must process the **NM_CUSTOMDRAW** notification message and then return a specific value that informs the control what it must do.

The key to harnessing custom draw features is in responding to the NM_CUSTOMDRAW notification messages that a control sends. The return values your application sends in response to these notifications determine the control behavior for that paint cycle.

Responding to the Prepaint Notification

At the beginning of each paint cycle, the control sends the NM_CUSTOMDRAW notification message, which specifies the CDDS_PREPAINT value in the *dwDrawStage* member of the accompanying **NMCUSTOMDRAW** structure. The value that your application returns to this first notification dictates how and when the control sends subsequent custom draw notifications for the rest of that paint cycle. The following table shows the flags that your application can return in response to the first notification.

Return value	Effect
CDRF_DODEFAULT	The control draws itself. It does not send additional NM_CUSTOMDRAW messages for this paint cycle. This flag cannot be used with any other flag.
CDRF_NOTIFYITEDRAW	The control notifies the parent of any item-specific drawing operations. It sends NM_CUSTOMDRAW notification messages before and after it draws items.

If your application returns CDRF_NOTIFYITEMDRAW to the initial prepaint custom draw notification, the control sends notifications for each item it draws during that paint cycle. These item-specific notifications have the CDDS_ITEMPREPAINT value in the *dwDrawStage* member of the accompanying **NMCUSTOMDRAW** structure. Your application can request that the control send another notification when it is done drawing the item by returning CDRF_NOTIFYPOSTPAINT to these item-specific notifications. Otherwise, your application can return CDRF_DODEFAULT, and the control will not notify the parent window until it starts to draw the next item.

If your application draws the item, it should return CDRF_SKIPDEFAULT. This enables the control to skip items that it need not draw, which conserves resources. Remember that returning this value means that the control will not draw any portion of the item, so your application must draw any item images.

Changing Fonts and Colors

Your application can use the custom draw service to change an item font. To do this, select the HFONT you want into the device context specified by the *hdc* member of the **NMCUSTOMDRAW** structure associated with that notification. Because the font you select might have different metrics from the default font, be sure that you include the CDRF_NEWFONT bit in the return value for the notification message. For more information, see the code examples in the "Sample Custom Draw Function" later in this chapter.

The font that your application specifies is used to display that item when it is not selected. Custom draw does not enable you to change the font attributes for selected items.

Sample Custom Draw Function

Upon receiving the prepaint notification CDDS_PREPAINT, the function requests item-specific notifications by returning CDRF_NOTIFYITEMDRAW. When it receives the subsequent item-specific notifications, it selects a previously created font into the provided device context and specifies new colors before returning CDRF_NEWFONT.

The following code example shows how an application-defined function processes custom draw notification messages sent by a child list view control.

```
LRESULT DoNotify (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
  LPNMLISTVIEW pnm = (LPNMLISTVIEW)lParam;

  switch (pnm->hdr.code)
  {
    case NM_CUSTOMDRAW:
    {
      LPNMLVCUSTOMDRAW  lplvcd = (LPNMLVCUSTOMDRAW)lParam;

      if (lplvcd->nmcd.dwDrawStage == CDDS_PREPAINT)
        return CDRF_NOTIFYITEMDRAW;

      if (lplvcd->nmcd.dwDrawStage == CDDS_ITEMPREPAINT)
      {
        if (!(lplvcd->nmcd.dwItemSpec % 3))
          SelectObject (lplvcd->nmcd.hdc, g_hNewFont);
        else
          return CDRF_DODEFAULT;

        lplvcd->clrText = RGB(150, 75, 150);
        lplvcd->clrTextBk = RGB(255,255,255);

        return CDRF_NEWFONT;
      }
    }

    default:
      break;
  }

  return 0;
}
```

CHAPTER 5

Working with Graphics

In Microsoft Windows CE, the graphics device interface (GDI) controls the display of text and graphics. GDI provides several functions and structures you can use to generate graphic output for displays, printers, and other devices. Using GDI functions, you can draw lines, curves, closed figures, text, and bitmapped images. The color and style of the items you draw depends on the drawing objects you create. GDI provides three drawing objects you can use to create graphics: pens to draw lines and curves, brushes to fill the interiors of closed figures, and fonts to write text.

The Windows CE GDI is designed for devices with limited system resources. Therefore, it does not include many of the special graphic functions found in Windows-based desktop platforms. As a consequence, the Windows CE GDI is a powerful, full-color graphics display system with a small footprint.

Applications direct output to a specified device by creating a *device context* for the device. The device context is a GDI-managed structure containing information about the device. An application creates a device context by calling device context functions. GDI returns a device context handle used to identify the device.

Applications can direct output to a physical device, such as a display or printer, or to a logical device, such as a memory device.

A device context also contains attributes that determine how GDI functions interact with a device. These attributes eliminate the need to specify every piece of information Windows CE requires to display an object on a device. If you want to change an attribute, you can use attribute functions to change current device settings and operating modes. Operating modes include text and background colors and the mixing mode that specifies how colors in a pen or brush combine with colors already on a display surface.

Getting a Handle to a Device Context

Windows CE provides several methods for obtaining a device context handle, depending on whether the device is a display, printer, or memory device. You use a display device context to draw in the client area of a screen. You use a memory device context to store bitmapped images in memory rather than sending it to an output device. A memory device context enables Windows CE to treat a portion of memory as a virtual device. You use a printer device context to send output to a printer.

It is important when using a device context that you release or delete it when it is no longer in use. Releasing a device context frees the device for use by other applications.

Note Failure to delete objects no longer in use can significantly affect performance.

Obtaining a Display Device Context

To get a handle to a display device context, call the **BeginPaint** or **GetDC** function and supply a handle to a window. Windows CE returns a handle to a display device context with default objects, attributes, and graphic modes. Newly created device contexts start with default brush, palette, font, pen, and region objects. You can begin drawing using these defaults, or you can choose a new object, change the attributes of an existing object, or choose a new mode.

You can examine a default object's attributes by calling the **GetCurrentObject** and **GetObject** functions. The **GetCurrentObject** function returns a handle identifying the current pen, brush, palette, bitmap, or font, and the **GetObject** function initializes a structure containing the object attributes.

The following table shows the object-specific creation functions you can call to replace a default object.

Graphics object	Creation functions
Bitmap	**CreateBitmap**, **CreateCompatibleBitmap**, **CreateDIBSection**
Brush	**CreateDIBPatternBrushPt**, **CreatePatternBrush**, **CreateSolidBrush**
Palette	**CreatePalette**
Font	**CreateFontIndirect**
Pen	**CreatePen**, **CreatePenIndirect**

Each of these functions returns a handle identifying the new object. After you retrieve a handle, you can call the **SelectObject** function to select the new object into the device context. However, you should save the **SelectObject** return value because it is the handle to the default object. When you finish using the new object, use **SelectObject** to restore the default object, and delete the new object with the **DeleteObject** function.

When you have finished drawing in the display area, you must release the device context by calling the **EndPaint** or **ReleaseDC** function. If you called **BeginPaint** to create the device context, then call **EndPaint** to release it. If you called **GetDC** to create the device context, then call **ReleaseDC** to release it. Generally, you call **BeginPaint** and **EndPaint** while processing WM_PAINT messages in your window procedure. Otherwise, call **GetDC** and **ReleaseDC** to obtain and release a device context.

The following code example shows how to call **GetDC** and **ReleaseDC** to obtain and release a device context and how to call **SelectObject** to get a new object.

```
HDC hDC;                // Handle to a display device context
HBRUSH hBrush,          // Handle to the new brush object
       hOldBrush;       // Handle to the old brush object

// Retrieve the handle to the display device context.
if (!(hDC = GetDC (hwnd)))
  return;

// Create a solid brush and select it into the device context.
hBrush = CreateSolidBrush (RGB(0, 255, 255));
hOldBrush = SelectObject (hDC, hBrush);

// Draw a rectangle.
Rectangle (hDC, 0, 0, 100, 200);

// Select the old brush back into the device context.
SelectObject (hDC, hOldBrush);

// Delete the new brush object.
DeleteObject (hBrush);

// Release the device context.
ReleaseDC (hwnd, hDC);
```

Obtaining a Memory and Printer Device Context

You can create a memory device context for a device by calling the **CreateCompatibleDC** function and supplying a handle to the device context. Memory device contexts are also called compatible device contexts because they are created to be compatible with a particular device. Windows CE creates a temporary 1 x 1 pixel monochrome bitmap and selects it into the device context after calling **CreateCompatibleDC**. Before you can draw with this device context, you must call the **SelectObject** function to select a bitmap with the appropriate width, height, and color depth into the device context. Once the new bitmap is selected into the memory device context, you can use the device context to store images. For more information about image storage, see "Creating Bitmaps" later in this chapter.

You obtain a handle to a printer device context by calling the **CreateDC** function. Call the **DeleteDC** function to delete the printer device context when finished printing.

Note You must delete, rather than release, a printer or memory device context; the **ReleaseDC** function fails if you try to use it to release a printer or memory device context.

Modifying a Device Context

Once you have created a device context, call **GetDeviceCaps** to retrieve device data. **GetDeviceCaps** provides data about a device's color format and raster capabilities, as well as its shape, text, and line drawing capabilities.

Before modifying a device context, save the current device context settings. Call the **SaveDC** function to record the condition of your device context's graphics objects and graphic modes to a special GDI stack. Call this function to save your application's original state. Call **RestoreDC** to return the device context to this original state.

To modify the appearance of a device context, you can use graphics mode functions. Graphics modes control general display characteristics, such as how colors are mixed. Windows CE supports the *background graphics mode* and *drawing mode*. A background graphics mode defines how background colors are mixed with window or screen colors for text and bitmap operations. A drawing mode defines how foreground colors are mixed with window or screen colors for pen, brush, bitmap, and text operations.

Windows CE initializes a device context with default graphics modes. You can get the current background mix mode with the **GetBkMode** function and set it with the **SetBkMode** function. In Windows CE, the background mix mode affects the appearance of text and certain pen types. You can set the foreground mix mode with the **SetROP2** function. The foreground mix mode controls how the brush or pen colors and the image colors combine. **SetROP2** returns the mix mode for the last foreground mix mode.

You can change the viewpoint origin from its default starting point in the upper-left corner of the screen with the **SetViewportOrgEx** function.

Note Windows CE does not support multiple mapping modes. The only mapping mode is MM_TEXT, which maps logical coordinates to the physical coordinates in a 1:1 ratio from left to right and top to bottom.

Creating Bitmaps

A *bitmap* is an array of bits that, when mapped to a rectangular pixel array on an output device, creates an image. Use bitmaps to create, modify, and store images.

Windows CE supports two types of bitmaps, *device-dependent bitmaps* (DDBs) and *device-independent bitmaps* (DIBs). A device-dependent bitmap does not have its own color table and therefore can be properly displayed only by a device with the same display memory organization as the one on which it was created. A device-independent bitmap, on the other hand, generally has its own color table and therefore can be displayed on multiple devices. It is recommended that you use DIBs in Windows CE–based applications.

▶ **To create a device-independent bitmap**

1. Call the **CreateDIBSection** function.

   CreateDIBSection creates a **DIBSection**, which contains all the information necessary for displaying the DIB.

2. Call the **SelectObject** function to select the **DIBSection** into the device context.

3. Select the **DIBSection** again and call **DeleteObject** to delete the **DIBSection** when finished.

The **BITMAPINFO** structure defines the dimensions and color information for a DIB. This structure consists of a **BITMAPINFOHEADER** structure and an array of two or more **RGBQUAD** structures. The **BITMAPINFOHEADER** structure contains information about the dimensions and color format of a DIB. Each **RGBQUAD** structure defines one bitmap color. The **BITMAPINFO** structure must include a color table if the images are palettized with formats of 1, 2, 4, or 8 bits per pixel (bpp). For a 16 bpp or 32 bpp non-palettized image, the color table must be three entries long; the entries must specify the value of the red, green, and blue (RGB) bitmasks. Because GDI ignores the color table for 24-bpp bitmaps, you should store the image pixels in RGB format.

▶ **To create a device-dependent bitmap**

1. Call **CreateCompatibleDC** to create a memory device context.

   This function creates a device context compatible with the specified device. The device context contains a single-bit array that serves as a placeholder for a bitmap.

2. Call **CreateBitmap** or **CreateCompatibleBitmap** to create the bitmap. If calling **CreateCompatibleBitmap**, be sure that you specify a screen device context rather than a memory device context; otherwise, you will get a device context to a 1-bpp device.

3. Call **SelectObject** to select the bitmap into the device context.

   Windows CE then replaces the single-bit array with an array large enough to store color data for the specified rectangle of pixels.

When you draw using the handle returned by **CreateCompatibleDC**, the output does not appear on a device's drawing surface; rather, it is stored in memory. To copy the image stored in memory to a display device, call the **BitBlt** function. **BitBlt** copies the bitmap data from the bitmap in the source device context into the bitmap in the target device context. In this case, the source device context is the memory device context, and the target device context is the display device context. Thus, when **BitBlt** completes the transfer, the image appears on the screen. By reversing the source and target device contexts, you can call **BitBlt** to transfer images from the screen into memory.

The following code example shows how to create a memory device context, how to use a **CreateCompatibleBitmap** to create a bitmap, and how to use **BitBlt** to copy bitmap data from the source device context to the target device context.

```
VOID BitmapDemo (HWND hwnd)
{

  HDC hDC,                  // Handle to the display device context
      hDCMem;               // Handle to the memory device context
  HBITMAP hBitmap,          // Handle to the new bitmap
          hOldBitmap;       // Handle to the old bitmap
  static int iCoordinate[200][4];
  int i, j,
      iXSrc, iYSrc,         // x and y coordinates of the source
                            // Rectangle's upper-left corner
      iXDest, iYDest,       // x and y coordinates of the destination
                            // Rectangle's upper-left corner
      iWidth, iHeight;      // Width and height of the bitmap

  // Retrieve the handle to a display device context for the client
  // area of the window (hwnd).
  if (!(hDC = GetDC (hwnd)))
    return;

  // Create a memory device context compatible with the device.
  hDCMem = CreateCompatibleDC (hDC);

  // Retrieve the width and height of windows display elements.
  iWidth = GetSystemMetrics (SM_CXSCREEN) / 10;
  iHeight = GetSystemMetrics (SM_CYSCREEN) / 10;

  // Create a bitmap compatible with the device associated with the
  // device context.
  hBitmap = CreateCompatibleBitmap (hDC, iWidth, iHeight);

  // Select the new bitmap object into the memory device context.
  hOldBitmap = SelectObject (hDCMem, hBitmap);

  for (i = 0; i < 2; i++)
  {
    for (j = 0; j < 200; j++)
    {
      if (i == 0)
      {
        iCoordinate[j][0] = iXDest = iWidth * (rand () % 10);
        iCoordinate[j][1] = iYDest = iHeight * (rand () % 10);
        iCoordinate[j][2] = iXSrc = iWidth * (rand () % 10);
        iCoordinate[j][3] = iYSrc = iHeight * (rand () % 10);
      }
```

```
        else
        {
          iXDest = iCoordinate[200 - 1 - j][0];
          iYDest = iCoordinate[200 - 1 - j][1];
          iXSrc = iCoordinate[200 - 1 - j][2];
          iYSrc = iCoordinate[200 - 1 - j][3];
        }

        // Transfer pixels from the source rectangle to the destination
        // rectangle.
        BitBlt (hDCMem, 0, 0, iWidth, iHeight, hDC,  iXDest, iYDest,
                SRCCOPY);
        BitBlt (hDC,  iXDest, iYDest, iWidth, iHeight, hDC,  iXSrc, iYSrc,
                SRCCOPY);
      }
    }

    // Select the old bitmap back into the device context.
    SelectObject (hDC, hOldBitmap);

    // Delete the bitmap object and free all resources associated with it.
    DeleteObject (hBitmap);

    // Delete the memory device context and the display device context.
    DeleteDC (hDCMem);
    DeleteDC (hDC);

    return;
}
```

Bit block transfer (blit) functions, such as **BitBlt,** can be used to modify as well as transfer bitmaps. These functions modify a destination bitmap by combining it with a pen, a brush, and, in some cases, a source bitmap, in a format specified by a raster operation (ROP) code. Each ROP code specifies a unique logical pattern for combining graphics objects. For example, the **SRCCOPY** ROP simply copies a source bitmap to a destination bitmap, while the **MERGECOPY** ROP merges the colors of a source rectangle with a specified pattern.

The following table shows the ROP code types.

ROP type	Description
ROP2	Combines a pen or brush with a destination bitmap in one of 16 possible combinations.
ROP3	Combines a brush, a source bitmap, and a destination bitmap in one of 256 possible combinations.
ROP4	Uses a monochrome mask bitmap to combine a foreground ROP3 and a background ROP3. The mask uses zeros and ones to indicate the areas where each ROP3 will be used.

When the source and destination bitmaps are different sizes, you can call the **StretchBlt** function to perform a blit between the two bitmaps. **StrechBlt** copies a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap to fit the destination rectangle.

Additionally, you can call the **PatBlt** function to paint a selected rectangle using a selected brush and an ROP3 code. You can also call the **TransparentImage** to transfer all portions of a bitmap except for those drawn in a specified transparent color. This function is especially useful for transferring non-rectangular images such as icons.

Note Windows CE supports arbitrary bit pixel formats, which enable you to use blit functions among bitmaps with different pixel depths.

Working with Colors

Some display devices and printers display only monochrome images; others use hundreds, thousands, or millions of colors. You should design your applications to display properly on devices with a variety of color capabilities.

The color range available to a display device is determined primarily by the pixel depth it supports. Pixel depth is measured in bits per pixel. Each bit can have a value of 1 or 0. A pixel depth of 1 bpp accepts only two values. While these values are usually black and white, you can use any two colors available on your device. A pixel depth of 2 bpp has four possible color values or all possible combinations of 0 and 1 with two bits. In general, the number of possible colors is equal to 2 raised to the power of the pixel depth. Windows CE supports pixel depths of 1, 2, 4, 8, 16, 24, and 32 bpp.

You can call the **GetDeviceCaps** function, which specifies the **NUMCOLORS** value, to discover the number of colors a device supports. Usually, this count corresponds to a physical property of the output device, such as the number of inks in the printer or the number of distinct color signals the display adapter can transmit to the monitor. For modes greater than 8 bpp, use the return value of **GetDeviceCaps** with the **BITSPIXEL** value. The number of supported colors for that mode is 2 raised to the power of the return value.

Windows and applications use parameters and variables having the **COLORREF** type to pass and store color values. You can extract the individual values of the red, green, and blue components of a color value by using the **GetRValue**, **GetGValue**, and **GetBValue** macros, respectively. Use the **RGB** or **PALLETERGB** macros to create a color value from individual RGB component values.

If you request a color that the display device cannot generate, Windows CE approximates it with a color that the device can generate. For example, if you attempt to create a red pen for a black-and-white printer, you will receive a black pen instead—Windows CE uses black as the approximation for red.

You can discover how Windows CE approximates a specified color by calling the **GetNearestColor** function. The function takes a color value and returns the color value of the closest matching color the device can generate.

Windows handles colors in bitmaps differently from colors in pens, brushes, and text. Compatible bitmaps, created by calling the **CreateBitmap** or **CreateCompatibleBitmap** function, retain color information in a device-dependent format. Specifically, device-dependent bitmaps use the color values of the device on which they were created.

The **DIBSection** structure representing a DIB retains color information as either color values or color palette indexes. If color values are used, the colors might be approximated as necessary. Although Windows CE does not approximate colors identified by indexes, the colors in the bitmap could change if the palette changes.

Note An offscreen DIB section should have the same color table as the screen; otherwise, GDI will have to perform a time-consuming color translating blit when the DIB section is transferred to the screen. For grayscale devices, the color table should be 0x000000, 0x808080, 0xc0c0c0, and 0xFFFFFF. For color devices, the application should first query the standard palette to determine its color display capabilities, and then build a matching color table.

Working with Palettes

A *palette* is a collection that contains the colors that can be displayed on an output device. Palettes are used by devices that can display only a subset of their potential colors at any specified time.

Windows CE has no standard color palette and creates a default palette each time you create a device context. Windows CE bases this palette on the device capabilities. Most devices have 256 colors. Display devices, for example, often use the 16 standard video graphics adapter (VGA) colors and 4 other Windows CE–defined colors. Printer devices might use other default colors. If you specify a pen or text color not in the default palette, Windows CE approximates the display color with the closest color in the palette. When displaying bitmaps, Windows CE assigns colors to a bitmap based on the bitmap's associated color table. If an image has no color table, Windows CE uses the color palette in the currently selected device context.

You cannot change the entries in the default palette. However, you can create your own logical palette and select the palette into a device context in place of the default palette. You can use logical palettes to define and use colors that meet your specific needs. Windows CE enables you to create multiple logical palettes. You can attach each logical palette to a unique device context or you can switch between multiple logical palettes in a single device context.

Windows CE supports both palettized and non-palettized color display devices. Palettized devices have a color palette coded directly into their display card. Non-palettized devices use pixel bit values in the video memory to directly define colors in terms of their RGB values. You can use the **GetDeviceCaps** function to determine if a device supports color palettes.

▶ **To create a logical color palette**

1. Assign values to the members of a **LOGPALETTE** structure and pass a pointer to the structure to the **CreatePalette** function.

   The function returns a handle to a logical palette with the values specified in the **LOGPALETE** structure.
2. Call the **SelectPalette** function to select the palette into the current device context.
3. Call the **RealizePalette** function to make the system palette the same as the palette in the current device context.

Your logical palette should have just enough entries to represent the colors you need. You can call the **GetDeviceCaps** function with the **SIZEPALETTE** index to retrieve the maximum palette size associated with a device.

When working with palettes, you can change the colors in an existing logical palette as well as retrieve color values. The following table shows how to modify the color palette.

To	Call
Change the colors in an existing logical palette.	**SetPaletteEntries**
Update the display after changing or creating a palette.	**RealizePalette**
Retrieve the color values for a logical palette.	**GetPaletteEntries**
Retrieve the value in a specified logical palette that most closely matches a specified color value.	**GetNearestPaletteIndex**
Delete a logical palette. Be sure that the logical palette is not selected into a device context when you delete it.	**DeleteObject**

Note The **GetSystemPaletteEntries** and **RealizePalette** functions will fail if the device associated with the selected device index does not have a palette that you can set. Call **GetDeviceCaps** to determine if the device has a palette that you can set.

Windows CE does not arbitrate between the palettes of the background and foreground applications. The application running in the foreground controls the system palette. Applications that use colors other than standard Windows colors might not display properly when they run in the background. Windows CE does not perform any color matching operations between the foreground and background applications; therefore, background applications cannot successfully call **RealizePalette**.

The following code example shows how to create color palettes.

```
HPALETTE CreateScalePalette (HDC hDC, int iColor)
{
  HPALETTE hPalette = NULL;   // Handle of the palette to be created
  LPLOGPALETTE lpMem = NULL;  // Buffer for the LOGPALETTE structure
                              // which defines the palette
  int index,                  // An integer
     iReserved,               // Number of reserved entries in the
                              // system palette
     iRasterCaps;             // Raster capabilities of the display
                              // device context
  // Retrieve the raster capabilities of the display device context.
  // Check if it is capable of specifying a palette-based device, then
  // determine the number of entries in the logical color palette.

  iRasterCaps = GetDeviceCaps (hDC, RASTERCAPS);
  iRasterCaps = (iRasterCaps & RC_PALETTE) ? TRUE : FALSE;

  if (iRasterCaps)
  {
    iReserved = GetDeviceCaps (hDC, NUMRESERVED);
    iPalSize = GetDeviceCaps (hDC, SIZEPALETTE) - iReserved;
  }
  else
    iPalSize = GetDeviceCaps (hDC, NUMCOLORS);

  // If there cannot be any entries in the logical color palette, exit.
  if (iPalSize <= 0)
  {
    MessageBox (g_hwndMain,
                TEXT("Palette can't be created, there can't be any")
                TEXT("entries in it."),
                TEXT("Info"),
```

```
                MB_OK);
    goto exit;
}

// Allocate a buffer for the LOGPALETTE structure.
if (!(lpMem = (LOGPALETTE *) LocalAlloc (LMEM_FIXED,
                sizeof(LOGPALETTE) + sizeof(PALETTEENTRY) * iPalSize)))
    goto exit;

lpMem->palNumEntries = (WORD) iPalSize;
lpMem->palVersion = (WORD) 0x0300;

switch(iColor)
{
    case 0:             // Red color component only
        for (index = 0; index < iPalSize; index++)
        {
            lpMem->palPalEntry[index].peRed   = (BYTE) index;
            lpMem->palPalEntry[index].peGreen = 0;
            lpMem->palPalEntry[index].peBlue  = 0;
            lpMem->palPalEntry[index].peFlags = 0;
        }
        break;

    case 1:             // Green color component only
        for (index = 0; index < iPalSize; index++)
        {
            lpMem->palPalEntry[index].peRed   = 0;
            lpMem->palPalEntry[index].peGreen = (BYTE) index;
            lpMem->palPalEntry[index].peBlue  = 0;
            lpMem->palPalEntry[index].peFlags = 0;
        }
        break;

    case 2:             // Blue color component only
        for (index = 0; index < iPalSize; index++)
        {
            lpMem->palPalEntry[index].peRed   = 0;
            lpMem->palPalEntry[index].peGreen = 0;
            lpMem->palPalEntry[index].peBlue  = (BYTE) index;
            lpMem->palPalEntry[index].peFlags = 0;
        }
        break;
```

```
    case 3:             // Grayscale palette
    default:
      for (index = 0; index < iPalSize; index++)
      {
        lpMem->palPalEntry[index].peRed   = (BYTE) index;
        lpMem->palPalEntry[index].peGreen = (BYTE) index;
        lpMem->palPalEntry[index].peBlue  = (BYTE) index;
        lpMem->palPalEntry[index].peFlags = 0;
      }
      break;
  }

  // Create the palette.
  hPalette = CreatePalette (lpMem);

  // Free the memory object lpMem.
  LocalFree ((HLOCAL) lpMem);

exit:
  return hPalette;
}
```

Working with Pens

In Windows CE, a *pen* is a graphics object for drawing lines. Drawing applications use pens to draw freehand lines and straight lines. Computer-aided design (CAD) applications use pens to draw visible lines, section lines, center lines, and so on. Word processing and desktop publishing applications use pens to draw borders and rules. Spreadsheet applications use pens to designate trends in graphs and to outline bar graphs and pie charts.

Windows CE stock pens include the **BLACK_PEN** and the **WHITE_PEN**, which draw a solid, 1-pixel-wide line in their respective color, and the **NULL_PEN**, which does not draw. You obtain the stock pens with the **GetStockObject** function.

You call the **CreatePen** or **CreatePenIndirect** function to create a custom pen with a unique color, width, or pen style.

The following table shows the pen styles supported by Windows CE.

Pen style	Description
PS_SOLID	Draws a solid line
PS_DASH	Draws a dashed line
PS_NULL	Does not draw a line

Windows CE supports wide pens and dashed pens, but does not support wide pens, dashed pens, dotted pens, inside frame pens, geometric pens, or pen endcap styles.

You can create a pen with a unique color by storing the RGB value that specifies the color that you want in a **COLORREF** structure and passing this structure address to the **CreatePen** or **CreatePenIndirect** function. In the case of **CreatePenIndirect**, the **COLORREF** pointer is incorporated into the **LOGPEN** structure, which is used by **CreatePenIndirect**.

Note The wide pen requires significant GDI computation. To improve the performance of a handwriting application, use a standard size pen.

The following code example shows how to use pen functions.

```
#define NUMPT  200

HDC hDC;            // Handle to the display device context
HPEN hPen,          // Handle to the new pen object
     hOldPen;       // Handle to the old pen object
RECT rect;          // A RECT structure that contains the window's
                    // client area coordinates
int index,
    iCBHeight;      // Command bar height
POINT ptAxis[2],    // Two dimensional POINT structure array
     ptSine[NUMPT]; // 200 dimensional POINT structure array
static COLORREF g_crColor[] = {
                        0x000000FF,0x0000FF00,0x00FF0000,0x0000FFFF,
                        0x00FF00FF,0x00FFFF00,0x00FFFFFF,0x00000080,
                        0x00008000,0x00800000,0x00008080,0x00800080,
                        0x00808000,0x00808080,0x000000FF,0x0000FF00,
                        0x00FF0000,0x0000FFFF,0x00FF00FF,0x00FFFF00};

// Retrieve a handle to a display device context for the client
// area of the window (hwnd).
if (!(hDC = GetDC (hwnd)))
  return;

// Retrieve the coordinates of the window's client area.
GetClientRect (hwnd, &rect);
```

```
// Retrieve the height of the command bar in pixels.
iCBHeight = CommandBar_Height (g_hwndCB);

// Assign the axis points coordinates in pixels.
ptAxis[0].x = 0;
ptAxis[0].y = iCBHeight + (rect.bottom - iCBHeight) / 2;
ptAxis[1].x = rect.right - 1;
ptAxis[1].y = ptAxis[0].y;

// Assign the sine wave points coordinates in pixels.
for (index = 0; index < NUMPT; ++index)
{
  ptSine[index].x = index * rect.right / NUMPT;
  ptSine[index].y = (long) (iCBHeight + \
                            (rect.bottom - iCBHeight) / 2 * \
                            (1 - sin (8 * 3.14159 * index / NUMPT)));
}

// Create a dash pen object and select it.
hPen = CreatePen (PS_DASH, 1, g_crColor[5]);
hOldPen = SelectObject (hDC, hPen);

// Draw a straight line connecting the two points.
Polyline (hDC, ptAxis, 2);

// Select the old pen back into the device context.
SelectObject (hDC, hOldPen);

// Delete the pen object and free all resources associated with it.
DeleteObject (hPen);

// Create a solid pen object and select it.
hPen = CreatePen (PS_SOLID, 3, g_crColor[5]);
hOldPen = SelectObject (hDC, hPen);

// Draw a sine wave shaped polyline.
Polyline (hDC, ptSine, NUMPT);

// Select the old pen back into the device context.
SelectObject (hDC, hOldPen);

// Delete the pen object and free all resources associated with it.
DeleteObject (hPen);

// Release the device context.
ReleaseDC (hwnd, hDC);

return;
```

Working with Brushes

In Windows CE, a *brush* is a graphic object for painting the interior of closed shapes. Drawing applications use brushes to paint shapes; word-processing applications use brushes to paint rules; CAD applications use brushes to paint the interiors of cross-section views; and spreadsheet applications use brushes to paint graphs.

When you call one of the functions that create a brush, such as **CreatePatternBrush**, it returns a handle to a logical brush. When you select the logical brush into the device context with the **SelectObject** function, the device driver for the corresponding device creates the physical brush used for painting.

When you call a painting function, GDI maps a pixel in the brush bitmap to the window origin of the client area. The window origin is the upper-left corner of the window client area. The coordinates of the mapped pixel are called the *brush origin*. The default brush origin is the upper-left corner of the brush bitmap, at the coordinates (0, 0). You can call the **SetBrushOrgEx** function to change the location of the brush origin by a specified number of pixels. To make the changes effective, you must call the **SelectObject** function to select the modified brush.

Windows CE supports three types of logical brushes: stock brushes, solid brushes, and pattern brushes.

The types of stock brushes include the white brush, black brush, gray brush, light gray brush, dark gray brush, and the null brush, which does not paint. Call the **GetStockObject** function to select one of the stock brushes.

Windows CE maintains 21 stock brushes whose colors are used in window elements such as menus, scroll bars, and buttons. You can obtain a handle to a system stock brush with the **GetSysColorBrush** function. Furthermore, you can retrieve the color window element with the **GetSysColor** function, and set a color corresponding to a window element with the **SetSysColors** function.

A solid brush contains 64 pixels of the same color in an 8 x 8 pixel square. You can call the **CreateSolidBrush** function to create a solid brush of a specified color. To paint with your solid brush, call **SelectObject** to select it into a specified device context.

You can create a pattern brush from an application-defined bitmap or a device-independent bitmap. To create a logical pattern brush, you must create a bitmap and then call the **CreatePatternBrush** or **CreateDIBPatternBrushPt** function, supplying a handle that identifies the bitmap or DIB.

Windows CE does not support hatched brushes. However, you can achieve the effect of a hatched brush by calling the **CreateDIBPatternBrushPt** function to create a pattern brush with the hatch pattern that you want.

The following code example shows how to use brush functions.

```
HDC hDC;            // Handle to the display device context
HRGN hRgn;          // Handle to a region object
HBRUSH hBrush;      // Handle to a brush object
RECT rect;          // A RECT structure that contains the window's
                    // client area coordinates
static COLORREF g_crColor[] = {
                        0x000000FF,0x0000FF00,0x00FF0000,0x0000FFFF,
                        0x00FF00FF,0x00FFFF00,0x00FFFFFF,0x00000080,
                        0x00008000,0x00800000,0x00008080,0x00800080,
                        0x00808000,0x00808080,0x000000FF,0x0000FF00,
                        0x00FF0000,0x0000FFFF,0x00FF00FF,0x00FFFF00};

// Retrieve the handle to a display device context for the client
// area of the window (hwnd).
if (!(hDC = GetDC (hwnd)))
  return;

// Retrieve the coordinates of the window's client area.
GetClientRect (hwnd, &rect);

// Create a rectangular region.
hRgn = CreateRectRgn (0, 0, rect.right, rect.bottom);

// Create a solid brush.
hBrush = CreateSolidBrush (g_crColor[0]);

// Fill the region out with the created brush.
FillRgn (hDC, hRgn, hBrush);

// Delete the rectangular region.
DeleteObject (hRgn);

// Delete the brush object and free all resources associated with it.
DeleteObject (hBrush);

// Release the device context.
ReleaseDC (hwnd, hDC);

return;
```

Printing

Windows CE does not send printing commands directly to output devices. Rather, it passes all output information to device drivers, which in turn send the information to display devices and printers. Windows CE has a small footprint in part because it does not need to maintain hard-coded routines for interfacing with multiple output devices.

Most applications strive for "what you see is what you get" (WYSIWYG) output. Ideally, WYSIWYG means that text drawn with a specified font and size on the screen has a similar appearance when printed. However, it is almost impossible to obtain true WYSIWYG output, partly because of differences between video and printer technologies.

To obtain a WYSIWYG effect when drawing text, call the **CreateFont** function and specify the typeface name and logical size of the font you would like to draw with, and then call the **SelectObject** function to select the font into a printer device context. Windows CE will select a physical font that is the closest possible match to the specified logical font.

▶ **To start a print job**

1. Call the **SetAbortProc** function to establish an abort procedure.

   The abort procedure should include a modeless dialog box that enables a user to cancel a print job.
2. Initialize the necessary variables registered in your **AbortProc** function.
3. Display a modeless **Cancel** dialog box.
4. Call the **StartDoc** function to start the print job.

Once you start the print job, you can define individual pages in the document by calling the **StartPage** and **EndPage** functions and embedding the appropriate calls to GDI drawing functions within this bracket. After you define the last page, you can close the document and end the print job with the **EndDoc** function.

Windows CE does not have a print manager. It will not spool or print more than a single copy of a document.

Note The display driver does all the rendering in Windows CE and scales the output to the printer resolution. If you intend to print text, you should use a system with TrueType fonts because raster fonts cannot be scaled to different printer resolutions without significantly compromising text quality.

Working with Regions

In Windows CE, a *region* is a rectangle that can be filled, painted, framed, and tested to see if it contains a particular point.

You create a region by calling **CreateRectRgn** or **CreateRectRgnIndirect**. These functions return a handle identifying the new region. When using the **CreateRectRgn** and **CreateRectRgnIndirect** functions, use values for regions that can be represented by 16-bit integers because that is how region data is stored in Windows CE. Once you have a handle to a region, you can select the region into a device context with the **SelectObject** function.

You can perform a variety of operations on a region. You can paint or invert its interior, draw a frame around it, retrieve its dimensions, and test whether a particular point lies within it. The following table shows what tasks you can perform on regions.

To	Call
Determine if two regions are equal in size and shape	**EqualRgn**
Paint the interior of a region with a specified brush	**FillRgn**
Retrieve the dimensions of a region's bounding rectangle	**GetRgnBox**
Move a region a specified number of logical units	**OffsetRgn**
Retrieve data describing a region	**GetRegionData**
Determine if a point is inside a specified region	**PtInRegion**

You can also combine or compare a region with another region by calling the **CombineRgn** function. The following table shows how you can call the **CombineRgn** function to combine two regions together.

Value	Description
RGN_AND	The intersecting parts of two original regions define a new region.
RGN_COPY	A copy of the first of the two original regions defines a new region.
RGN_DIFF	The part of the first region that does not intersect the second defines a new region.
RGN_OR	The two original regions define a new region.
RGN_XOR	Those parts of the two original regions that do not overlap define a new region.

Windows CE does not support the **InvertRgn** or **InvertRect** functions. You can achieve the effect of **InvertRect** by calling the **PatBlt** function with an ROP code of DSTINVERT.

Clipping Regions

You can use clipping regions to restrict your output to a specified subregion of the client area. To use a clipping region, you must select it into the device context associated with the display device.

Clipping is used in Windows CE in a variety of ways. Word processing and spreadsheet applications clip keyboard input to keep it from appearing in the margins of a page or spreadsheet. Computer-aided design and drawing applications clip graphics output to keep it from overwriting the edges of a drawing or picture.

Some device contexts provide a predefined or default clipping region. For example, the device context created by the **BeginPaint** function contains a predefined rectangular clipping region that corresponds to the invalid rectangle to be repainted. However, the device contexts created by the **CreateDC** and **GetDC** functions contain empty clipping regions; clipping is done only to keep graphics output in the window client area.

You can perform a variety of operations on clipping regions. Some of these operations require a handle identifying the region and some do not. For example, you can perform the following operations directly on a device context clipping region:

- Determine if part of the client area intersects a region by calling the **RectVisible** function.
- Exclude a rectangular part of the client area from the current clipping region by calling the **ExcludeClipRect** function.
- Combine a rectangular part of the client area with the current clipping region by calling the **IntersectClipRect** function.

After obtaining a handle identifying the clipping region, you can perform any operation common with regions, such as the following operations:

- Combine a copy of the current clipping region with a second region by calling the **CombineRgn** function.
- Compare a copy of the current clipping region to a second region by calling the **EqualRgn** function.
- Determine if a point lies within the interior of a copy of the current clipping region by calling the **PtInRegion** function.

Creating Shapes and Lines

Windows CE enables you to draw lines and a variety of filled shapes. A line is a set of highlighted pixels on a raster display or a set of dots on a printed page identified by two points, a starting point and an ending point. In Windows CE, the pixel located at the starting point is always included in the line, and the pixel located at the ending point is always excluded.

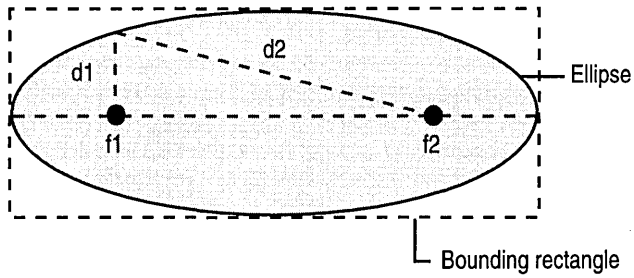
You can draw a series of connected line segments by calling the **Polyline** function and supplying an array of points that specify the ending point of each line segment.

Note Windows CE does not support the **LineTo** or the **MoveToEx** function. However, you can call the **Polyline** function in Windows CE to achieve the same results that you would get in Windows-based desktop platforms if you called the **MoveToEx** function and then made one or more calls to the **LineTo** function.

Filled shapes are geometric shapes that Windows CE outlines with the current pen and fills with the current brush. Windows CE supports four filled shapes: ellipse, polygon, rectangle, and round rectangle, which is a rectangle with rounded corners.

A Windows CE–based application uses filled shapes in a variety of ways. Spreadsheet applications, for example, use filled shapes to construct charts and graphs; drawing applications enable users to draw figures and illustrations using filled shapes.

An ellipse is a closed curve defined by two fixed points—*f1* and *f2*—such that the sum of the distances—*d1* + *d2*—from any point on the curve to the two fixed points is constant. The following illustration shows an ellipse drawn by using the **Ellipse** function.



When calling **Ellipse**, supply the coordinates of the upper-left and lower-right corners of the ellipse *bounding rectangle*. A bounding rectangle is the smallest rectangle that completely surrounds the ellipse.

A polygon is a filled shape with straight sides. Windows CE uses the currently selected pen to draw the sides of the polygon and the current brush to fill it. Windows CE fills all enclosed regions within the polygon with the current brush.

Note Windows CE does not support multiple fill modes. When it fills a polygon, it fills all subareas created by intersecting lines within the polygon.

A rectangle is a four-sided polygon whose opposing sides are parallel and equal in length and whose interior angles are 90 degrees. Although you can call the **Polygon** function to draw a rectangle if you supply it with all four sides, it is easier to call the **Rectangle** function. This function requires only the coordinates of the upper-left and the lower-right corners.

You can call the **RoundRect** function to draw a rectangle with rounded corners. Supply this function with the coordinates of the lower-left and upper-right corners of the rectangle and the width and height of the ellipse used to round each corner.

You can call the **FillRect** function to paint the interior of a rectangle. You can call the **FillRgn** function to fill a region using the specified brush.

Because Windows CE does not support paths, many line-drawing functions available on Windows-based desktop platforms are not available in Windows CE. Windows CE does not support functions to draw an arc, beizer curve, chord, pie, polypolygon, or polypolyline. However, you can approximate these shapes using existing Windows CE drawing functions. For example, you can create an arc by calling the **Ellipse** function with an appropriately defined clipping region.

Note The **Ellipse** and **RoundRect** functions require significant GDI computation. To increase your application performance, use these functions sparingly.

The following code example shows how to create shapes and lines using the **Rectangle**, **Ellipse**, **Polygon**, and **RoundRect** functions.

```
VOID DrawRandomObjects (HWND hwnd)
{
  HDC hDC;                  // Handle to the display device context
  RECT rect;                // A RECT structure that contains the
                            // window's client area coordinates
  POINT pt[4];              // Four dimensional POINT structure array
  HBRUSH hBrush,            // Handle to the new brush object
         hOldBrush;         // Handle to the old brush object
  TCHAR szDebug[80];        // A debug message string

  int x1, y1, x2, y2, x3, y3, x4, y4,
                            // The coordinates of four points
      iRed, iGreen, iBlue,  // Indicate the Red, Green, Blue component
                            // Color of the brush
      iObject;              // An integer indicates the type of objects

  // Retrieves the handle to the display device context.
  if (!(hDC = GetDC (hwnd)))
    return;

  // Retrieve the coordinates of a window's client area.
  GetClientRect (hwnd, &rect);

  // Avoid divide by zero errors when the window is small.
  if (rect.right == 0)
    rect.right++;
  if (rect.bottom == 0)
    rect.bottom++;

  // Generate three random numbers.
  iRed = rand() % 255;
```

```
    iGreen = rand() % 255;
    iBlue = rand() % 255;
  // Create a solid brush object and select it into the device context.
    hBrush = CreateSolidBrush (RGB(iRed, iGreen, iBlue));

    if (hOldBrush = SelectObject (hDC, hBrush))
    {
      // Randomly generates four points.
      x1 = rand() % rect.right;
      y1 = rand() % rect.bottom;
      x2 = rand() % rect.right;
      y2 = rand() % rect.bottom;
      x3 = rand() % rect.right;
      y3 = rand() % rect.bottom;
      x4 = rand() % rect.right;
      y4 = rand() % rect.bottom;

      // Randomly generate an integer to indicate the type of objects.
      iObject = rand() % 4;

      switch (iObject)
      {
        case 0:
          wsprintf (szDebug, TEXT("Rectangle(%d ,%d, %d, %d)\n"),
                    x1, y1, x2, y2);

          // Draws a rectangle.
          Rectangle (hDC, x1, y1, x2, y2);

          break;

        case 1:
          wsprintf (szDebug, TEXT("Ellipse(%d, %d, %d, %d)\n"),
                    x1, y1, x2, y2);

          // Draws an ellipse.
          Ellipse (hDC, x1, y1, x2, y2);

          break;

        case 2:
          wsprintf (szDebug, TEXT("RoundRect (%d, %d, %d, %d, %d, %d)\n"),
                    x1, y1, x2, y2, x3, y3);

          // Draws a rectangle with rounded corners.
          RoundRect (hDC, x1, y1, x2, y2, x3, y3);

          break;
```

```
        case 3:
          pt[0].x = x1;
          pt[0].y = y1;
          pt[1].x = x2;
          pt[1].y = y2;
          pt[2].x = x3;
          pt[2].y = y3;
          pt[3].x = x4;
          pt[3].y = y4;

          wsprintf (szDebug,
                    TEXT("Chord(%d, %d, %d, %d, %d, %d, %d, %d)\n"),
                    x1, y1, x2, y2, x3, y3, x4, y4);

          // Draws a polygon.
          Polygon(hDC, pt, 4);

          break;

        default:
          break;
      }

      // Select the old brush into the device context.
      SelectObject (hDC, hOldBrush);

      // Delete the brush object and free all resources associated with
      //it.
      DeleteObject (hBrush);
    }

    ReleaseDC (hwnd, hDC);
    return;
}
```

Creating Text and Fonts

In Windows CE, a font is a collection of glyphs that share a common design. A font is characterized by its typeface, style, and size. The font typeface determines the specific characteristics of the glyphs, such as the relative width of the thick and thin strokes used in any specified character. The style determines the font weight and slant. Font weights can range from thin to black. Slants can be roman (upright) or italic. The size of a font is the distance from the bottom of a lowercase "g" to the top of an adjacent uppercase "M," measured in points. A point is approximately one seventy-second of an inch.

In Windows CE, fonts are grouped into families, which share common stroke width characteristics. Fonts within a family are distinguished by size and style. The following table shows the font families.

Font family name	Description
Decorative	Specifies a novelty font, for example, Old English.
Dontcare	Specifies a generic family name. This name is used when information about a font does not exist or does not matter.
Modern	Specifies a monospace font with or without serifs. Monospace fonts are usually modern; examples include Pica, Elite, and Courier New.
Roman	Specifies a proportional font with serifs, for example, Times New Roman.
Script	Specifies a font designed to look like handwriting; examples include Script and Cursive.
Swiss	Specifies a proportional font without serifs, for example, Arial.

These family names correspond to constants found in the Wingdi.h header file: FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, and FF_SWISS. Use these constants to create, select, or retrieve font information.

Working with TrueType and Raster Fonts

Windows CE supports raster and TrueType font technologies, but accepts only one type to be used on a specified system. The choice of TrueType or raster font type is made when the system is designed and cannot be changed by an application.

The differences between raster and TrueType fonts have to do with the way the glyph for each character or symbol is stored in the respective font resource file. A raster font glyph is a tiny bitmap that represents a single character size. Because the bitmaps for each glyph in a raster font are designed for a specific resolution on a particular device, raster fonts are generally considered device-dependent.

A TrueType font glyph contains outlines and hints. Windows CE uses these hints to adjust the outlines used to draw the glyphs. These hints and the respective adjustments are based on the amount of scaling used to reduce or increase the size of the glyph. Because TrueType characters can be scaled up or down and still retain their original appearance, they are considered device-independent.

A font's glyphs are stored in a font resource file. A font resource file for a raster font is stored in a .fot file. TrueType fonts have two files, a short .fot header file and a .ttf file that contains the actual data. When installing a TrueType font, you do not have to install the .fot file, only the .ttf file.

Enabling Font Linking

Windows CE provides font linking capability, making it possible to link one or more fonts, called *linked fonts*, to another font, called the *base font*. Once you link fonts, you can use the base font to display code points that do not exist in the base font, but do exist in one of the linked fonts. For example, linking a Hangeul font and a Japanese font to a Latin font gives you the ability to display both Korean and Japanese languages in the Latin font using the Unicode text API.

Note Font linking can only add glyphs to a base font; it is not possible to override or replace glyphs in the base font.

If font linking is enabled on your device, you can examine the registry by enumerating the subkeys of the registry key **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\FontLink\SystemLink** to determine the mappings of linked fonts to base fonts. You can add additional linkings by creating additional subkeys. The following code example shows how to add an additional linking.

```
"base font face name" = "path and file to link to," "face name of the
font to link"
```

Creating End User Defined Characters

Although Windows CE defines thousands of characters, you might need to define your own set of characters. Use an end user defined character (EUDC) any time you need to define a character or glyph for a device. Always associate an EUDC with a double-byte character set (DBCS) and a TrueType font. When you create an EUDC, choose a reserved DCBS value. Applications use DBCS values to identify the EUDC. Windows CE uses DBCS values to locate the shape and style information in the corresponding TrueType font. The shape and style information specifies how to draw the EUDC.

▶ **To create an EUDC**

1. Choose a character value in the specified range or ranges of reserved characters.
2. Use an EUDC editor to create the shape and style of the character.
3. Add the shape and style information to the TrueType font in the entry that corresponds to the selected character value.

▶ **To associate an EUDC font with another font**

1. Copy the EUDC font to a folder.

   The EUDC font has a .tte extension.
2. Call **EnableEUDC** (FALSE).
3. Modify the **HKEY_CURRENT_USER\EUDC** registry key.
4. Create a subkey under **HKEY_CURRENT_USER\EUDC**.
5. In the subkey that you created in step 4, enter the font path that contains the EUDCs.

   For example, enter Tahoma=\windows\test03.tte in the subkey to link the Tahoma font with the test03.tte font located in the \Windows directory.
6. Call **EnableEUDC** (TRUE).

Note Before creating EUDC entries in the registry, enumerate the existing EUDC settings to ensure that you do not overwrite entries defined for the Windows CE–based device.

Installing and Using Fonts

Call the **AddFontResource** function to load a font from a font resource file. When you finish using an installed font, call the **RemoveFontResource** function to remove the font. Whenever you add or delete a font resource, you should call the **SendMessage** function to send a WM_FONTCHANGE message to all top-level windows in the system. This message notifies other applications that an application has modified the internal font table by adding or removing a font. You do not need to call **AddFontResources** to create or realize system fonts.

There are two stages to selecting a font. In the first stage, you specify the ideal font you would like to use. This theoretical font is called a *logical font*. In the second stage, an internal algorithm finds the *physical font* that is the closest match to your specified logical font. A physical font is a font stored on the device or in the operating system. This process is called *font mapping*.

▶ **To use a font**

1. Call the **EnumFontFamilies** function to build a list of the available fonts. This is especially useful when you want to determine available fonts from a specified font family or typeface.
2. Use the values returned by the font enumeration function to initialize the members of a **LOGFONT** structure.
3. Create the logical font by calling the **CreateFontIndirect** function and passing it a pointer to the initialized **LOGFONT** structure.
4. Select the logical font into the current device context with the **SelectObject** function.

When you call **SelectObject**, Windows CE loads the physical font that most closely matches the logical font specified in the **LOGFONT** structure.

When initializing the members of the **LOGFONT** structure, be sure that the *lfCharSet* member specifies a particular character set. This member is used in the font mapping process and the results will be inconsistent if this member is not initialized correctly. If you specify a typeface name in the *lfFaceName* member of the **LOGFONT** structure, be sure that the **lfCharSet** value contains an appropriate value.

Windows CE keeps a table containing all the fonts available for application use. When you call **CreateFontIndirect**, Windows CE chooses a font from this table.

Windows CE provides six standard logical fonts. You can use the **GetStockObject** function to obtain a standard font. The following table shows the standard font values.

Value	Description
ANSI_FIXED_FONT	Specifies a monospace font based on the Windows character set, usually represented by a Courier font.
ANSI_VAR_FONT	Specifies a proportional font based on the Windows character set, usually represented by the MS Sans Serif font.
DEVICE_DEFAULT_FONT	Specifies the preferred font for the specified device, usually represented by the System font for display devices.
OEM_FIXED_FONT	Specifies a monospace font based on an OEM character set.
SYSTEM_FONT	Specifies the System font. This is a proportional font based on the Windows character set and is used by the operating system to display window titles, menu names, and text in dialog boxes. The System font is always available. Other fonts are available only if installed.

Enumerating Fonts

You can enumerate the available fonts by calling the **EnumFonts** or **EnumFontFamilies** function. These functions send information about the available fonts to a callback function that the application supplies. The callback function stores the information in the **LOGFONT** structure and in either the **NEWTEXTMETRIC** structure for TrueType fonts or the **TEXTMETRIC** structure for raster fonts. By using the information returned from these functions, you can limit the user's choices to available fonts only.

The **EnumFontFamilies** function is similar to the **EnumFonts** function, but includes some extra features for use with TrueType fonts. The **EnumFontFamilies** function enumerates all the styles associated with a specified typeface, and not simply the bold and italic attributes. For example, when the system includes a TrueType font called Courier New Extra-Bold, **EnumFontFamilies** lists it with the other Courier New fonts.

Note Despite its name, **EnumFontFamilies** actually enumerates the styles associated with a specified typeface—for example, Arial—rather than a font family, such as Swiss.

If you do not supply a typeface name, the **EnumFonts** and **EnumFontFamilies** functions supply information about one font in each available typeface. To enumerate all the fonts in a device context, you can specify NULL for the typeface name, compile a list of the available typefaces, and then enumerate each font in each typeface.

Formatting Text

Windows CE provides a complete set of functions to format and draw text in an application client area and on a page of printer paper.

The default text color for a display device context is black; the default background color is white; and the default background mode is OPAQUE. Call the **SetTextColor** and **GetTextColor** functions to respectively set and retrieve the color of text drawn in the client area of a window or printed by a color printer. Call the **SetBkColor** and **GetBkColor** functions to respectively set or retrieve the background color. Call the **SetBkMode** and the **GetBkMode** functions to respectively set or retrieve the background mode. The background mode specifies the logical method for combining the selected background color with the current video display colors.

You can call the **GetTextExtentPoint32** function to compute the advance width and height of a string of text. You can call the **GetTextMetrics** function to retrieve a font's logical dimensions. You can call the **GetDeviceCaps** function to determine the dimensions of an output device.

Drawing Text

After you have selected a font, set your text-formatting options, and computed the necessary character width and height values for a string of text, you can draw characters and symbols using either the **DrawText** or **ExtTextOut** function. When you call one of these functions, the operating system passes the call to the graphics engine, which in turn passes the call to the appropriate device driver.

In most cases **ExtTextOut** is faster than **DrawText**. However, there are some instances when **DrawText** is more efficient, as in the case where you need to draw multiple lines of text within the borders of a rectangular region. **DrawText** does not work with rotated text.

CHAPTER 6

Working with Sound

Adding sound to an application can make it more efficient to use. By using sounds to draw attention at critical points, you can help users avoid mistakes and notify them when a time-consuming operation concludes. This chapter describes two different ways to play sound in an application. The first part describes how to use the **PlaySound** function. **PlaySound** provides all the necessary capabilities for playing waveform-audio sounds on a Windows CE–based device. The latter part of this chapter describes how to use the Waveform Audio application programming interface (API). This interface gives an application exact control over waveform audio input/output (I/O) devices.

Using the PlaySound Function

You can use the **PlaySound** function to play waveform audio sound files, as long as the sound can be stored in available device memory. The following list describes three ways to play sound with **PlaySound**:

- As a file name
- As a system alert, using the alias stored in the registry
- As a resource identifier

You can also use the **sndPlaySound** function to play waveform audio files. However, **sndPlaySound** offers a subset of the **PlaySound** feature set Windows CE maintains **sndPlaySound** for backward compatibility.

Using the PlaySound Function with Waveform Audio Files

In Windows CE, most waveform audio files use the .wav file name extension. The following code example shows how to play the \Sounds\Bells.wav file.

```
PlaySound (TEXT("\\SOUNDS\\BELLS.WAV"), NULL, SND_SYNC);
```

If the specified file does not exist or the file cannot be stored in available device memory, **PlaySound** plays the default system sound. If you have not defined a default system sound, **PlaySound** fails without playing a sound. The following code example shows how to specify that the default sound should not be played.

```
PlaySound (TEXT("\\SOUNDS\\BELLS.WAV"), NULL, SND_SYNC | SND_NODEFAULT);
```

The example uses the SND_NODEFAULT flag in the *fdwSound* parameter to prohibit the application from playing the default sound.

Using PlaySound with Registry-Specified Sounds

The **PlaySound** function also plays sounds referred to by a key name in the registry. Using this feature, you can let users assign selected or custom sounds to system alerts and warnings. Sounds associated with system alerts and warnings are called *sound events*.

▶ **To play a sound event**

- Call **PlaySound** with *pszSound* pointing to a string containing the name of the registry entry that identifies the sound.

  The following code example shows how to use **PlaySound** to play a sound event.

  ```
  PlaySound (TEXT("MouseClick"), NULL, SND_SYNC);
  ```

Note **PlaySound** plays the default system sound if the registry entry specified in *pszSound* does not exist or if the sound file cannot be stored in available device memory.

The **sndPlaySound** function searches the registry for a key name matching *lpszSound* before attempting to load a file with the *pszSound* name. The **PlaySound** function accepts flags that specify the location of the sound.

Using PlaySound with a Resource Identifier

To play a sound stored as a resource, use the **PlaySound** function. Although you can use **sndPlaySound** to play a resource sound, you must find, load, lock, unlock, and free the resource; in contrast, **PlaySound** completes these tasks in a single line of code.

▶ **To include a .wav file as a resource in an application**

- Add the following entry to the application resource script (.rc) file.

```
soundName WAVE \sounds\bells.wav
```

The name *soundName* is a placeholder for a name you supply to refer to the wave resource sound. Wave resources are loaded and accessed like other application-defined Windows resources.

The following code example shows how to use the **PlaySound** function to play the wave resource sound.

```
PlaySound (TEXT("soundName"), hInst, SND_RESOURCE | SND_ASYNC);
```

In contrast, the following code example shows how to use the **sndPlaySound** function to play a wave resource sound.

```
BOOL PlayResource (LPTSTR lpName)
{
    BOOL bRtn;
    LPTSTR lpRes;
    HANDLE hResInfo, hRes;

    // Find the WAVE resource.
    hResInfo = FindResource (hInst, lpName, "WAVE");

    if (hResInfo == NULL)
      return FALSE;

    // Load the WAVE resource.
    hRes = LoadResource (hInst, hResInfo);

    if (hRes == NULL)
      return FALSE;

    // Lock the WAVE resource and play it.
    lpRes = LockResource (hRes);

    if (lpRes != NULL)
  {
      bRtn = sndPlaySound (lpRes, SND_MEMORY | SND_SYNC |
SND_NODEFAULT);
      UnlockResource (hRes);
    }
    else
      bRtn = 0;

    // Free the WAVE resource and return success or failure.
    FreeResource (hRes);
```

```
    return bRtn;
}
```

To play a wave resource sound by using **sndPlaySound**, pass the function a pointer to a string containing the resource name, as shown in the following code example.

```
PlayResource (TEXT("soundName"));
```

Using the Waveform Audio Interface

This section describes the Waveform Audio API interface. An application uses this interface to gain the greatest possible control over audio I/O devices. Specifically, this section discusses the following features of the Waveform Audio API:

- Querying and opening waveform audio I/O devices
- Writing waveform-audio data
- Allocating audio data blocks
- Playing waveform-audio files
- Handling errors with audio functions
- Deallocating waveform audio data blocks
- Closing waveform audio output devices

Querying and Opening Waveform Audio I/O Devices

To correctly record or play a sound, first determine what drivers your Windows CE–based device has available for audio I/O, and then open those drivers for recording or playback.

Querying Audio I/O Devices

The following table shows functions that retrieve the number of audio devices on a Windows CE–based device.

Function	Description
waveInGetNumDevs	Retrieves the number of waveform audio input devices present in a system
waveOutGetNumDevs	Retrieves the number of waveform audio output devices present in a system

After you determine how many devices are present in a system, you can query the capabilities of each device.

The following table shows the functions and structures that retrieve this information.

Function	Description	Returned structure
waveInGetDevCaps	Retrieves the capabilities of a specified waveform audio input device	**WAVEINCAPS**
waveOutGetDevCaps	Retrieves the capabilities of a specified waveform audio output device	**WAVEOUTCAPS**

The **waveInGetDevCaps** and **waveOutGetDevCaps** fill the *dwFormats* member of the **WAVEINCAPS** and **WAVEOUTCAPS** structures with flags describing the standard supported sound formats for a specified audio I/O device. Waveform audio I/O devices also support nonstandard capabilities. You can also use the **waveInOpen** or the **waveOutOpen** function to determine if a waveform audio I/O device supports a specific format.

▶ To determine if a waveform audio I/O device supports a standard or nonstandard format

1. Specify the format you want to query in the **WAVEFORMATEX** structure.
2. Pass this structure into **waveInOpen** or **waveOutOpen** with the *pwfx* parameter.
3. Call **waveInOpen** or **waveOutOpen** with the WAVE_FORMAT_QUERY flag set.

   This call does not open the device. Instead, the function returns a message declaring whether or not the audio device supports the specified format.

Note While **WAVEFORMATEX** supersedes **PCMWAVEFORMAT** and **WAVEFORMAT**, Windows CE maintains both structures for backward compatibility.

The following code example shows how to use the **waveOutOpen** function with the WAVE_FORMAT_QUERY flag to determine whether a waveform audio device supports a specified format.

```
MMRESULT IsFormatSupported (LPWAVEFORMATEX pwfx, UINT uDeviceID)
{
    return waveOutOpen (NULL,                 // ptr can be NULL for query
                        uDeviceID,            // The device identifier
                        pwfx,                 // Defines the requested
                                              // format
                        NULL,                 // No callback
                        NULL,                 // No instance data
                        WAVE_FORMAT_QUERY); // Query only, do not open
}
```

The preceding example determines whether the specified waveform-audio output device supports a specified waveform-audio format. It returns MMSYSERROR_NOERROR if the device supports the format, WAVERROR_BADFORMAT if the device does not support the format, and one of the other MMSYSERROR codes for other errors.

This technique for determining nonstandard format support also applies to waveform audio input devices. The only difference is that the function will use **waveInOpen** instead of **waveOutOpen** to query for format support.

To determine if any waveform audio I/O devices on a system support a particular waveform audio data format, use the technique illustrated in the previous example. However, you must specify the WAVE_MAPPER constant in the *uDeviceID* parameter.

Opening Waveform Audio Output Devices

In addition to querying the capabilities of a device, you can use the **waveOutOpen** or **waveInOpen** functions to open a waveform audio I/O device for recording or playback. These functions open the device that is associated with the specified device identifier and return a pointer to an open device handle.

Both **waveOutOpen** and **waveInOpen** choose the device best able to play the specified data format. The Windows CE operating system (OS) identifies waveform audio I/O devices by using a device identifier. The OS determines the device identifier implicitly from the number of devices present in a system. Device identifiers range from zero through the number of devices present minus one. For example, the valid device identifiers for a system with two waveform audio output devices are 0 and 1.

In addition to the device number, the **waveOutOpen** and **waveInOpen** functions require a pointer to a memory location. The functions fill the memory location with a device handle. Use this device handle to identify the open waveform audio I/O device when calling other audio functions. The following list describes the differences between a device identifier and a device handle:

- The OS determines a device identifier implicitly from the number of devices present on a system. The system obtains this number by using the **waveInGetNumDevs** or **waveOutGetNumDevs** function.
- The OS returns a device handle after opening a device driver with the **waveInOpen** or **waveOutOpen** function.

Allocating Audio Data Blocks

Once you have determined the capabilities of your Windows CE–based device, you can allocate memory for your audio data blocks. Using the **WAVEHDR** structure, allocate memory for the **waveInAddBuffer** and **waveOutWrite** functions to play sound. The following table shows functions that prepare headers.

Function	Description
waveInPrepareHeader	Prepares a waveform-audio input data block
waveOutPrepareHeader	Prepares a waveform-audio output data block

Before passing an audio data block to a device driver with **waveInAddBuffer** or **waveOutWrite**, call **waveInPrepareHeader** or **waveOutPrepareHeader** on the data block. Then pass the data block to **waveInAddBuffer** or **waveOutWrite**.

Playing Waveform Audio Files

After successfully opening a waveform-audio output device driver and preparing the header file, you can begin playing a sound. Windows provides the **waveOutWrite** and **waveInAddBuffer** functions for sending data blocks to a waveform audio output device.

Use the **WAVEHDR** structure to specify the waveform-audio data block you are sending by using the **waveOutWrite** or **waveInAddBuffer** function. This structure contains a pointer to a locked data block, the length of the data block, and some flags. After you send a data block to an output device, wait until the device driver is finished with the data block before freeing it. If you are sending multiple data blocks, monitor the completion of data blocks to know when to send additional blocks.

Note Unless the waveform audio input and output data is contained in a single data block, an application must continually supply the device driver with data blocks until recording or playback is complete.

Retrieving the Current Playback Position

Monitor the current playback position within the data block by using the **waveOutGetPosition** or the **waveInGetPosition** function and the **MMTIME** structure. Use the **MMTIME** structure to represent time in one or more different formats, including milliseconds, samples, a MIDI song pointer, or the Society of Motion Picture and Television Engineers (SMPTE) time formats. The *wType* member specifies the format used to represent time.

▶ **To retrieve the current playback position**

1. Set the *wType* member in the **MMTIME** structure to the preferred time format.
2. Call **waveOutGetPosition** or **waveInGetPosition**.

   While calling this function, pass the **MMTIME** structure in the *pmmt* parameter.
3. Check the *wType* member after the call to see whether the requested time format is supported.

   If the device does not support the time format, the device driver specifies the time in an alternate time format and changes *wType* to the selected time format.

Most waveform audio devices support samples as the preferred time format. Thus, a waveform audio device usually describes its current position in a .wav file as the number of samples from the beginning of the waveform-audio file.

Stopping, Pausing, and Restarting a Waveform Audio I/O Device

While recording or playing waveform audio, you might want to stop, pause, or restart the audio I/O device. The following table shows the functions that control these capabilities.

Function	Description
waveInStop	Stops recording on a waveform audio input device
waveOutPause	Pauses playback on a waveform audio output device
waveInReset	Stops recording on a waveform audio input device and marks all pending data blocks as done
waveOutReset	Stops playback on a waveform audio output device and marks all pending data blocks as done
waveInStart	Begins recording on a waveform audio input device
waveOutRestart	Resumes playback on a paused waveform audio output device

Generally, the waveform audio I/O device begins recording or playing as soon as the **waveOutWrite** or **waveInAddBuffer** function sends the first waveform-audio data block.

▶ **To delay activation of waveform-audio output**

1. Call **waveOutPause** to pause the waveform-audio output device.
2. Call **waveOutWrite**.

   This sends the first data block to the waveform-audio output device. However, the device is paused by the previous call to **waveOutPause**.
3. Call waveOutRestart to begin playback.

Once the device begins playback, it might not respond to **waveOutPause** immediately. Depending on the device driver, the device might finish playing the current data block before pausing.

You can also delay activation of waveform-audio input. Call **waveInAddBuffer** as needed to allocate space for the audio input. The device will not begin recording until you call **waveInStart**.

Once the device stops recording or playback in response to a **waveOutReset** or **waveInReset** command, you cannot restart playback with **waveOutRestart** or **waveInStart**. Instead, resume recording or playback by sending the next data block with **waveOutWrite** or **waveInAddBuffer** and **waveInStart**.

Looping Playback

You can control looping playback on a waveform-audio output device with the **WAVEHDR** structure passed into **waveOutWrite**.

▶ **To loop playback**

1. Set the WHDR_BEGINLOOP flag in the *dwFlags* member of the **WAVEHDR** structure.

   This flag defines the beginning data block of a message loop.
2. Set the number of loops in the *dwLoops* member of the same **WAVEHDR** structure.
3. Pass this and the rest of your data blocks to **waveOutWrite**.
4. Pass the final data block to **waveOutWrite** with the WHDR_ENDLOOP flag set in the **WAVEHDR** structure.

   This causes your device to begin looping.
5. Use **waveOutBreakLoop** to break out of the loop prematurely.

Note Loop a single block by setting the WHDR_BEGINLOOP and WHDR_ENDLOOP flags in the same **WAVEHDR** structure.

Changing the Volume of Waveform Audio Playback

The following table shows Windows CE functions that you can use to query and set the volume level of waveform audio output devices.

Function	Description
waveOutGetVolume	Returns the current volume level of the specified waveform-audio output device
waveOutSetVolume	Sets the volume level of the specified waveform audio output device

Volume is specified in a **DWORD** value. In a stereo audio format, the upper 16 bits specify the relative volume of the right channel. The lower 16 bits specify the relative volume of the left channel. For devices that do not support separate volume control for the different channels, Windows CE uses the lower 16 bits to specify the volume level and ignores the upper 16 bits.

Windows CE uses volume ranges from 0x0 (silence) through 0xFFFF (maximum volume) and interprets these ranges logarithmically. The perceived volume increase is the same when increasing the volume level from 0x5000 through 0x6000 as it is from 0x4000 through 0x5000.

Changing the Pitch and Playback Rate

Some waveform audio I/O devices can vary the pitch and the playback rate of waveform audio data. Like other device capabilities, you can query your waveform audio I/O device with the **waveOutGetDevCaps** function to determine if your device supports either of these capabilities.

The following table describes the functions that you can use to query and set waveform audio pitch and playback rates.

Function	Description
waveOutGetPitch	Retrieves the pitch for the specified waveform audio output device
waveOutGetPlaybackRate	Retrieves the playback rate for the specified waveform audio output device
waveOutSetPitch	Sets the pitch for the specified waveform audio device
waveOutSetPlaybackRate	Sets the playback rate for the specified waveform audio output device

These functions change the pitch and playback rates by a factor specified with a fixed-point number packed into a **DWORD** value. The upper 16 bits specify the integer part of the number; the lower 16 bits specify the fractional part.

The following table shows three examples of this style of packaging.

Value	Packaged value
1.5	0x00018000L
0.75	0x0000C000L
1.0	0x00010000L

A value of 1.0 means the pitch or playback rate is unchanged.

Although changing the pitch and changing the playback rate seem similar for the user, they are implemented quite differently. Because the device driver controls the playback rate, altering the playback rate does not require any specialized hardware. However, the driver does not change the sample rate. Instead, the driver skips or synthesizes samples. For example, the driver could skip every other sample if an application changed the playback rate by a factor of two. In contrast, changing the pitch requires specialized hardware. When an application alters the pitch, the application does not specifically alter the playback or sample rate.

Handling Errors with Audio Functions

The waveform audio functions return a nonzero value when an error occurs. Windows CE provides functions that convert these error values into textual descriptions of the errors. The application must still examine the error values to determine how to proceed, but it can use a textual description in a dialog box to describe an error to the user. The following table shows the functions used to retrieve textual descriptions of audio error values.

Function	Description
waveInGetErrorText	Retrieves a textual description of a specified waveform audio input error
waveOutGetErrorText	Retrieves a textual description of a specified waveform audio output error

The only audio functions that do not return error values are **waveInGetNumDevs** and **waveOutGetNumDevs**. These functions return 0 if no devices are present in a system or if they encounter any errors.

Using Windows Messages to Manage Waveform Audio Playback

You can send a variety of Windows CE messages to a windows procedure function to manage waveform audio playback. The following table shows these messages.

Message	Description
MM_WIM_CLOSE	Sent when the **waveOutClose** or the **waveInClose** function closes a device
MM_WIM_DATA	Sent when the device driver finishes with a data block that was sent by **waveOutWrite** or **waveInAddBuffer**
MM_WIM_OPEN	Sent when **waveOutOpen** or **waveInOpen** opens a device

MM_WIM_DATA is the most useful message in this table. When MM_WIM_DATA signals a completed data block, you can clean up and free that data block. Unless you need to allocate memory or initialize variables, you probably do not need to process the MM_WIM_OPEN or MM_WIM_CLOSE messages.

Like other windows messages, these window messages have a *wParam* and an *lParam* parameter associated with them. The *wParam* always specifies a handle to the open waveform audio output device. For the MM_WIM_DATA message, *lParam* specifies a pointer to a **WAVEHDR** structure. This structure identifies a completed data block. The MM_WIM_CLOSE and MM_WIM_OPEN messages do not use *lParam*.

The following code example shows how to process the MM_WIM_DATA message.

```
// WndProc--Main window procedure
LRESULT FAR PASCAL WndProc (HWND hWnd, UINT msg, WPARAM wParam,
                            LPARAM lParam)
{
  switch (msg)
  {
    case MM_WIM_DATA:
      // A waveform audio data block has been played and can now be
      // freed.
      waveOutUnprepareHeader ((HWAVEOUT)wParam, (LPWAVEHDR)lParam,
                              sizeof (WAVEHDR));
      // Free hData memory
      waveOutClose ((HWAVEOUT)wParam);

      break;
  }

  return DefWindowProc (hWnd, msg, wParam, lParam);
}
```

The preceding example assumes that the application does not play multiple data blocks, so it can close the output device after playing a single data block.

Deallocating Memory Blocks

An application must be able to determine when a device driver is finished with the data block so that the application can unprepare and free the memory associated with the data block and header structure. The following list describes several ways to determine when a device driver is finished with a data block:

- By specifying a callback function to receive a message that is sent by the driver when it is finished with a data block
- By using an event callback
- By specifying a window or thread to receive a message that is sent by the driver when it is finished with a data block
- By polling the WHDR_DONE bit in the *dwFlags* member of the **WAVEHDR** structure that is sent with each data block

▶ **To use a callback function for driver messages**

1. Specify the CALLBACK_FUNCTION flag in the *fdwOpen* parameter of the **waveInOpen** or **waveOutOpen** function.
2. Specify the address of the callback in the *dwCallback* parameter of the **waveInOpen** or **waveOutOpen** function.

Messages sent to a callback function are similar to messages sent to a window, except that they have two **DWORD** parameters instead of one **UINT** and one **DWORD** parameter.

The following list describes techniques for passing instance data from an application to a callback function:

- Pass the instance data by using the *dwInstance* parameter of the function that opens the device driver
- Pass the instance data by using the *dwUser* member of the **WAVEHDR** structure that identifies an audio data block that was sent to a device driver

---

Note If you need more than 32 bits of instance data, pass a pointer to a structure containing the additional information.

▶ **To use an event callback**

1. Retrieve the handle of an event from **CreateEvent.**
2. Specify CALLBACK_EVENT for the *fdwOpen* parameter in your call to **waveOutOpen.**
3. Use **waveOutPrepareHeader** or **waveInPrepareHeader** to prepare the data block.
4. Call **ResetEvent** with the event handle retrieved by **CreateEvent.**

   This action creates a non-signaled event.
5. Send the waveform audio data block to the driver.
6. Call **WaitForSingleObject**, specifying as parameters the event handle and a time-out value of INFINITE.

   This call should be inside a loop that checks whether the WHDR_DONE bit is set in the *dwFlags* member of the **WAVEHDR** structure.

Because event callbacks do not receive specific close, done, or open notifications, an application might have to check the status of the process that it is waiting for after the event occurs. It is possible for a number of tasks to be completed by the time **WaitForSingleObject** returns.

▶ **To use a window callback function**

- Call **waveInOpen** or **waveOutOpen** with the *fdwOpen* parameter set to CALLBACK_WINDOW and a window handle passed in the *dwCallback* parameter.

▶ **To use a callback thread**

- Call **waveInOpen** or **waveOutOPen** with the *fdwOpen* parameter set to CALLBACK_THREAD and a thread handle passed in the *dwCallback* parameter.

Note Messages sent to the window or thread callback are specific to the audio I/O device type being used.

In addition to using a callback function, you can poll the *dwFlags* member of a **WAVEHDR** structure to determine when an audio I/O device is finished with a data block. Sometimes it is better to poll *dwFlags* than to wait for another mechanism to receive messages from the drivers. For example, after calling the **waveOutReset** or **waveInReset** function to release pending data blocks, you can immediately poll *dwFlags* for WHDR_DONE to be sure that the data blocks have been released.

Once you determine that the function has released the data block, call the **waveInUnprepareHeader** or **waveOutUnprepareHeader** function to unprepare the header file. After you have unprepared the header file, you can deallocate the memory normally.

Closing Waveform Audio Output Devices

After you have finished playing a waveform-audio file and deallocating the associated header files, call the **waveOutClose** or **waveInClose** function to close the output device. If you call these functions while an application is still using a waveform-audio file, the close operation fails and the function returns an error value indicating that the device is not closed. If you do not want to wait for recording or playback to end before closing the device, call the **waveOutReset** or **waveInReset** function.

CHAPTER 7

Receiving User Input

User input is the means by which a user communicates with a Microsoft Windows CE–based device. The OEM determines the specific combination of input devices that are supported by your platform. Different platforms support different input devices. For example, some platforms, such as the Palm-size PC, support a touch screen rather than a keyboard for text entry. Other platforms might include handwriting recognition software in place of or in addition to a keyboard. The Windows CE operating system (OS) supports the following types of user input:

- Keyboard
- Mouse
- Touch screen and stylus
- Input panel
- Handwriting recognition

Receiving Keyboard Input

The keyboard is an important means of user input on many Windows CE–based devices. Windows CE maintains a device-independent keyboard model that enables it to support a variety of keyboards. The OEM usually determines the keyboard layout for a specified Windows CE–based device.

At the lowest level, each key on the keyboard generates a *scan code* when pressed and released. The scan code is a hardware-dependent number that identifies the key. Unlike Windows-based desktop platforms, Windows CE has no standard set of keyboard scan codes. Your application should rely only on supported scan codes for the target platform.

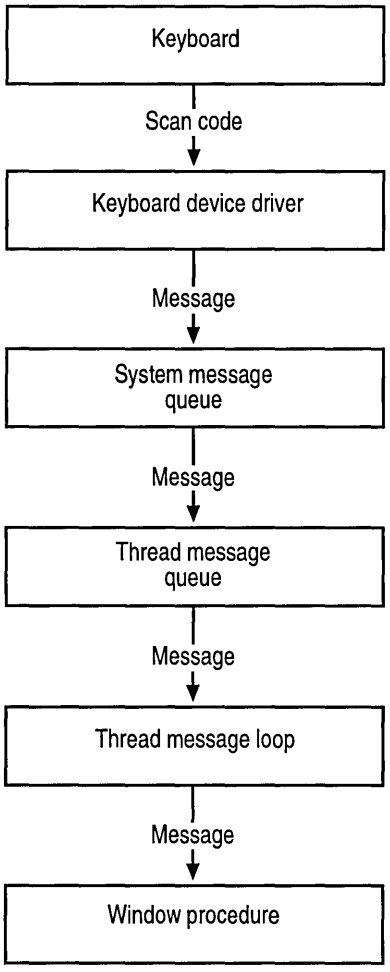
The keyboard driver translates or maps each scan code to a *virtual key code*. The virtual key code is a hardware-independent number that identifies the key. Because keyboard layouts vary from language to language, Windows CE offers only the core set of virtual key codes found on all keyboards. This core set includes Latin letters, numbers, and a few critical keys, such as the function and arrow keys. Keys not included in the core set also have virtual key code assignments, but their values vary from one keyboard layout to the next. You should depend only on the virtual key codes in the core set.

In addition to mapping, the keyboard driver determines which characters the virtual key generates. A single virtual key generates different characters depending on the state of other keys, such as the SHIFT and CAPS LOCK keys. Do not confuse virtual key codes with characters. Although many of the virtual key codes have the same numeric value as one of the characters that the key generates, the virtual key code and the character are two different elements. For example, the same virtual key generates the uppercase "A" character and the lowercase "a" character.

After translating the scan code into a virtual key code, the device driver creates a keyboard message containing scan code, virtual key code, and character data, and then the device driver places the message in the system message queue. The system removes the message from the system message queue and posts it to the message queue of the appropriate thread. Each thread maintains its own *active window* and *focus window*. The active window is a top-level window. The focus window is either the active window or one of its descendants. The active window of this thread is considered the *foreground window*.

The device driver places keyboard messages in the message queue of the foreground thread. The thread message loop pulls the message from the queue and sends it to the window procedure of the thread focus window. If the focus window is NULL, the window procedure of the active window receives the message.

The following illustration shows the keyboard input model.



Working with Threads

There are a number of ways a thread can become the foreground thread. If an application calls the **SetForegroundWindow** function and specifies a top-level window, the thread that owns the window becomes the foreground thread and the window becomes its active window. This function also moves the window to the top of the z-order. You can use **SetForegroundWindow** on any top-level window.

In most cases, if the user chooses a window, the system will place that window in the foreground. The thread that created the window becomes the foreground thread. If the foreground window is hidden or destroyed, the system designates another window as the foreground window. In that case, the new foreground window thread becomes the foreground thread. Call the **GetForegroundWindow** function to get the current foreground window.

In general, an application thread does not need to set the foreground window explicitly. This is usually done by the system when the user selects and closes windows. Call the **SetActiveWindow** function to activate a window. If the calling thread is the foreground thread, the new active window automatically becomes the foreground window. When the activation changes, the system sends a WM_ACTIVATE message to both the deactivated and activated windows. A thread can call the **GetActiveWindow** function to access its active window.

An application thread calls the **SetFocus** function to move the focus between its windows. When the focus changes, the system sends a WM_KILLFOCUS message to the window losing the focus. It sends a WM_SETFOCUS message to the window gaining the focus.

The system ensures that the focus window is always the active window or a descendant of the active window. If the focus changes to a window with a different top-level ancestor, the system first changes the activation, and then it changes the focus.

Processing Keyboard Messages

A window receives keyboard input in the form of keystroke messages and character messages. Keystroke messages control window behavior and character messages determine the text that is displayed in a window.

Windows CE generates a WM_KEYDOWN or WM_SYSKEYDOWN message when a user presses a key. If the user holds a key down long enough to start the keyboard repeat feature, the system generates repeated WM_KEYDOWN or WM_SYSKEYDOWN messages. When the user releases a key, the system generates a WM_KEYUP or WM_SYSKEYUP message.

The system makes a distinction between system keystrokes and nonsystem keystrokes. System keystrokes produce system keystroke messages, WM_SYSKEYDOWN and WM_SYSKEYUP. Non-system keystrokes produce nonsystem keystroke messages, WM_KEYDOWN and WM_KEYUP.

System keystroke messages are generated when the user types a key in combination with the ALT key or when the user types a key and the focus is NULL. If the focus is NULL, the keyboard event is delivered to the active window. These messages have the WM_SYS prefix in the message name. System keystroke messages are primarily used by the system rather than by an application. The system uses them to provide its built-in keyboard interface to menus and to enable the user to control which window is active. If a window procedure processes system keyboard messages, it should pass the message to the **DefWindowProc** function. Otherwise, all system operations involving the ALT key are disabled whenever that window has the keyboard focus.

The window procedure of the window that has the keyboard focus receives all keystroke messages. However, an application that responds to keyboard input typically processes WM_KEYDOWN messages only.

When the window procedure receives the WM_KEYDOWN message, it should examine the virtual-key code that accompanies the message to determine how to process the keystroke. The virtual-key code is contained in the message's *wParam* parameter.

The *lParam* parameter of a keystroke message contains additional data about the keystroke that generated the message. The following table shows the additional keystroke data required by the *lParam* parameter.

Data	Description
Repeat count	Specifies the number of times the keystroke was repeated as a result of a user holding down the key.
Scan code	Gives the hardware-dependent key scan code.
Context code	The value is 1 if the ALT key was pressed and 0 if the pressed key was released.
Previous key state	The value is 1 if the pressed key was previously down and 0 if the pressed key was previously up. The value is 1 for WM_KEYDOWN and WM_SYSKEYDOWN keystroke messages that were generated by the automatic repeat feature.
Transition state	The value is 1 if the key was released or 0 if the key was pressed.

Typically, an application processes only keystrokes that are generated by non-character keys. The following code example shows the window procedure framework that a typical application uses to receive and process keystroke messages.

```
case WM_KEYDOWN:
      switch (wParam)
      {
        case VK_HOME:
          // Insert code here to process the HOME key
          // ...
          break;

        case VK_END:
          // Insert code here to process the END key
          // ...
          break;

        case VK_INSERT:
          // Insert code here to process the INS key
          // ...
          break;

        case VK_F2:
          // Insert code here to process the F2 key
          // ...
          break;

        case VK_LEFT:
          // Insert code here to process the LEFT ARROW key
          // ...
          break;

        case VK_RIGHT:
          // Insert code here to process the RIGHT ARROW key
          // ...
          break;

        case VK_UP:
          // Insert code here to process the UP ARROW key
          // ...
          break;

        case VK_DOWN:
          // Insert code here to process the DOWN ARROW key
          // ...
          break;
```

```
        case VK_DELETE:
          // Insert code here to process the DEL key
          // ...
          break;

        default:
          // Insert code here to process other non-character keystrokes
          // ...
          break;
    }
```

Processing Character Messages

When a user enters a character, Windows CE does not automatically generate a character message. Instead, it generates a keystroke message. To translate a keystroke message into a corresponding character message, the application message loop must call the **TranslateMessage** function. This function examines the virtual-key code of a keystroke message and, if the code corresponds to a character, places a character message into the message queue. The character message is removed and dispatched on the next iteration of the message loop.

The message loop should call the **TranslateMessage** function to translate every message, not just keystroke messages. Although **TranslateMessage** has no effect on other types of messages, using it ensures that keyboard input is translated correctly. The following code example shows how to include the **TranslateMessage** function in a typical thread message loop.

```
while (GetMessage(&msg, (HWND) NULL, 0, 0))
{
   if (TranslateAccelerator(hwndMain, haccl, &msg) == 0)
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
}
```

Windows CE includes four character messages: WM_CHAR, WM_SYSCHAR, WM_DEADCHAR, and WM_SYSDEADCHAR. A typical window procedure ignores all character messages except WM_CHAR. The WM_CHAR message contains the character code and the flags that provide additional data about the character.

When a window procedure receives the WM_CHAR message, it should examine the character code that accompanies the message to determine how to process the character. The character code is in the message *wParam* parameter.

The following code example shows the window procedure framework that a typical application uses to receive and process character messages.

```
while (GetMessage (&msg, (HWND)NULL, 0, 0))
{
   if (TranslateAccelerator (hwndMain, haccl, &msg) == 0)
   {
      TranslateMessage (&msg);
      DispatchMessage (&msg);
   }
}
```

Creating and Displaying a Caret

A window that receives keyboard input typically displays the characters a user types in the window client area. A window should use a caret to indicate the position in the client area where the next character will appear. The window should create and display the caret when it receives the keyboard focus, and it should hide and destroy the caret when it loses the focus. A window can perform these operations when the WM_SETFOCUS and WM_KILLFOCUS messages are processed.

Use the **CreateCaret**, **ShowCaret**, **DestroyCaret**, and **HideCaret** functions to control the visibility of the caret. Use the **SetCaretPos** function to change the position of the caret as the user types.

Checking Other Keys

While processing a keyboard message, an application sometimes needs to determine the status of a key different from the one that generated the current message. You can use the **GetKeyState** function to determine the state of certain keys. This function returns the key's state at the time the current message was generated. The **GetAsyncKeyState** function returns the state of the key at the time of the call.

The Windows CE version of these functions differs slightly from the desktop counterpart. Unlike the equivalent functions in Windows-based desktop platforms, **GetKeyState** supports only a limited number of keys, and **GetAsyncKeyState** returns the current key state even if a window in another thread has the keyboard focus.

Adding Hot Key Support

A *hot key* is a key combination that generates a WM_HOTKEY message. The message is routed to a particular window, regardless of whether or not that window is the current foreground window or focus window.

You define a hot key by calling the **RegisterHotKey** function and specifying the combination of keys that generates the WM_HOTKEY message, the handle of the window to receive the message, and the hot key identifier. When the user presses the hot key, the system places a WM_HOTKEY message in the message queue of the thread that created the specified window. The *wParam* parameter of the message contains the hot key identifier. Before the application terminates, it should use the **UnregisterHotKey** function to destroy the hot key.

Receiving Stylus Input

In many Windows CE–based applications, the user interacts with an application by using a stylus and a touch screen. The stylus and touch screen provide a direct and intuitive alternative to a mouse.

The stylus generates an input event when the user touches the screen with a stylus or moves the stylus when the tip is touching the screen. To an application, stylus input is a subset of mouse input. When a user presses and releases a stylus on a screen, the application processes these events as a click of the left mouse button. When a user moves the stylus across the screen, the application processes this as a mouse move event.

Stylus input events in a window are posted to the message queue of the thread that created the window. A window receives a stylus message when a stylus event occurs within the window client area. When the user presses the stylus to the screen, the window receives a WM_LBUTTONDOWN message. When the stylus is lifted from the screen, the window receives a WM_LBUTTONUP message. Occasionally, a window receives a WM_LBUTTONDBLCLK message instead of a WM_LBUTTONDOWN message. This occurs under the following conditions:

- The window class was registered with the CS_DBLCLKS class style.
- The stylus touches the screen within a certain distance of the last stylus location.
- The stylus touches the screen within a certain time limit after the stylus last touched the screen.

If a user moves the stylus while pressing it to the screen, Windows CE generates a WM_MOUSEMOVE message.

The following table shows style input messages that are supported by Windows CE.

Message	Description
WM_LBUTTONDBLCLK	The user double-tapped the screen.
WM_LBUTTONDOWN	The user pressed the screen.
WM_LBUTTONUP	The user released the stylus from the screen.
WM_MOUSEMOVE	The user moved the stylus while the tip was pressed to the screen.

The *lParam* parameter of a stylus message indicates the position of the stylus tip. The low-order word is the x-coordinate and the high-order word is the y-coordinate. The coordinates are specified in the client coordinates. In the client-coordinate system, all points are specified relative to the upper-left corner of the client area.

The *wParam* parameter contains flags that indicate the status of the other stylus buttons and the CTRL and SHIFT keys at the stylus event time. Check for these flags when the way you process a stylus event depends on the state of another stylus button or on the CTRL or SHIFT key. The following table shows the flags that you can set in the *wParam* parameter.

Value	Description
MK_CONTROL	The CTRL key is down.
MK_LBUTTON	The stylus is touching the screen.
MK_SHIFT	The SHIFT key is down.

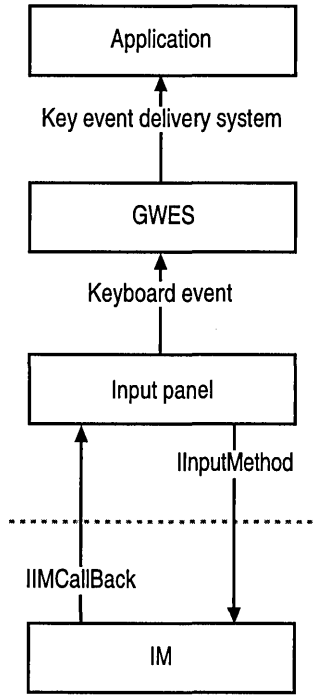
Receiving Input from an Input Panel

Windows CE–based devices that do not have a keyboard require an input method (IM) to simulate keyboard input. For this purpose, Windows CE implements an input panel architecture that functions through a touch screen. This input panel architecture is an IM that enables your application to accommodate input in multiple forms.

The Windows CE IM requires two parts: the software input panel subsystem and IMs. The primary difference between Windows CE IMs and usual keyboard input is that an additional level of interpretation is necessary in Windows CE to convert non-keyboard input into a keyboard event. This conversion is handled by the input panel subsystem in the OS, which also manages software input methods.

The IM can display the input panel window in a specific state by using the **IIMCallback::SetImInfo** method. This method also changes the icon appearing on the **Input Panel** button. To adjust to user-initiated changes in the input panel window, applications can query the **SIPINFO** structure for data on the input panel window size, state, and visible client area.

The input panel creates the IM through the **IInputMethod** interface. Once created, the IM receives user input and passes this data to the input panel through the **IIMCallBack** interface. After the input panel receives data from the IM, the input panel passes the data to the Graphics, Windowing, and Events Subsystem (GWES) module. Usually, this message is a keyboard event. GWES passes the message to your application through the standard keyboard event delivery system. The following illustration shows how an IM, an input panel, GWES, and an application communicate.



If the input panel is altered, the OS sends out a WM_SETTINGCHANGE message to all active applications. The application can modify the input panel and IM through the **SHSipInfo** function.

Note To use input panel technology, your device must include Coresip.lib. This component enables any application to initialize an IM. Call the **SipStatus** function to be sure that Coredsip.lib is present before implementing IMs.

Programming an Input Panel

When a user accesses the input panel, Windows CE creates a dedicated input panel thread. The thread creates an input panel window and initializes the input panel. Then, the thread enters a message loop. Within the message loop, the thread responds to messages and user interface (UI) requests from the input panel window. The thread also calls into the IM object. This enables the IM to create child windows in the input panel window. The content of the input panel window is determined by the current IM.

The input panel thread has a special status with the OS. Any window that the input panel thread creates will not be obscured by other windows. Because some UI elements save and clear themselves when they lose focus, the input panel and its children do not receive the focus, even if a user is currently using the input panel.

The input panel queries the IM for data through the **IInputMethod** interface. The input panel can remove the current IM and replace it with a new IM. The following table shows the IM queries that an input panel sends to an IM, listed in the order received.

Method	Description
Deselect	Destroys its window and performs IM-specific cleanup procedures
Select	Creates the IM window and image list
GetInfo	Requests data regarding the new IM, including property flags and the preferred IM size
ReceiveSipInfo	Sends the IM data about the size, placement, and docked status that the IM should use
RegisterCallback	Provides the IM with a pointer to the **IIMCallback** interface

After the input panel calls these methods, the IM should render the input panel window space and respond to user actions. For more information about programming the IM, see "Programming Input Methods" later in this chapter.

An application that uses the input panel should know the input panel state—whether the panel is visible, whether it is docked, or floating, and its size and position. This data is stored in the **SIPINFO** structure, which is accessed through the **SHSipInfo** function. The following code example shows how to use the **SHSipInfo** function to access and modify the **SIPINFO** structure.

```
BOOL LowerSip( void )
{
  BOOL fRes = FALSE;
  SIPINFO si;

  memset (&si, 0, sizeof (si));
  si.cbSize = sizeof (si);

  if (SHSipInfo (SPI_GETSIPINFO, 0, &si, 0))
  {
    si.fdwFlags &= ~SIPF_ON;
    fRes = SHSipInfo (SPI_SETSIPINFO, 0, &si, 0);
  }

  return fRes;
}
```

When the user changes the input panel state, the OS sends out a WM_SETTINGCHANGE message to all active applications. This message has SPI_SETSIPINFO in its *wParam* parameter. The following code example shows how you can use the **SHSipInfo** function to move the input panel on the screen in response to a WM_SETTINGCHANGE message.

```
WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    SIPINFO si;

    switch (msg)
    {
        case WM_SETTINGCHANGE:
            switch (wParam)
            {
                case SPI_SETSIPINFO:
                    memset (&si, 0, sizeof (si));
                    si.cbSize = sizeof (si);

                    if (SHSipInfo (SPI_GETSIPINFO, 0, &si, 0))
```

```
{
                        MoveWindow (
                    hwnd,
                    si.rcVisibleDesktop.left,
                    si.rcVisibleDesktop.top,
                    si.rcVisibleDesktop.right - si.rcVisibleDesktop.left,
                    si.rcVisibleDesktop.bottom - si.rcVisibleDesktop.top,
                    TRUE);
                      }
                      break;
                }
                break;
    }
  return 0;
}
```

A change in the active IM sends messages to all top-level applications that are registered to receive IM notifications. Typically, the messages go to an application that controls a display, such as a taskbar. The following table shows the messages.

Event	Windows CE message	Application action
IM size or position changes	WM_IM_INFO	Return 0 if the application processes the message
IM has a new icon to associate with its current state	WM_IM_INFO	Return 0 if the application processes the message

Programming Input Methods

An IM is an in-process Component Object Model (COM) server that implements the **IInputMethod** interface. Windows CE provides a default QWERTY keyboard IM to handle alpha-numeric input. The IM manages the space inside the input panel window. Within this window, the IM is responsible for screen output and responding to user input. Typically, an IM creates a child window of the input panel window. This enables the IM window to respond to user input through the input panel window; the IM does not have access to the input panel window's **WindowProc** function unless the IM subclasses that window. The IM usually converts user input into characters and sends those characters to the input panel through the **IMCallback** interface.

In order to use an IM, the user first selects the IM from the input panel dialog box or **Input Panel** button. The input panel dynamically loads the selected IM by calling the **CoCreateInstance** function. When the user selects a different IM, the input panel frees the existing IM by calling **Release** on the interface pointer. The input panel then calls **IInputMethod** methods to notify the IM of events and to request data. The input panel implements and exposes the **IIMCallback** interface. This interface lets the IM call the input panel to send keystrokes to the application through GWES. The following table shows the methods available through the **IIMCallback** interface.

IIMCallback method	Description
SetImInfo	Lets the IM change the **input panel** icon and the visible state of the input panel
SendVirtualKey	Modifies the global key state
SendCharEvents	Sends Unicode characters to the window with the current focus
SendString	Sends strings to the window with the current focus

In response to the input panel calls to **IInputMethod**, an IM creates windows in the contexts of the input panel thread. This way the input panel and the IM belong to the same message loop. For simplicity, all calls to **IIMCallback** should be made in the input panel thread. That is, the IM should call **IIMCallback** methods only in response to a call coming through an **IInputMethod** method.

Additionally, the IM should not create a separate thread to implement a UI. Only the thread that responds to **IInputMethod** methods should create and manipulate windows. Instead, the IM can create process threads to implement a UI. However, these process threads must not call **IIMCallback**, because certain GWES window functions work properly only if created from the same thread that created the window.

If you develop your own IM component, you should have your setup application perform self-registration by calling the **DllRegisterServer** and **DllUnregisterServer** functions. Implement these functions in the IM server dynamic-link library (DLL). Optionally, you can set the registry values directly. The **Input Panel** properties dialog box does not provide any UI elements for the self-registration service.

Input Method Registry Values

Technically, any COM component that implements **IInputMethod** can be selected into the input panel. The **IsSIPInputMethod** subkey is a shortcut that presents a list of IMs to a user without loading and querying each object for the **IInputMethod** interface.

IMs are installed in the system as in-process COM servers by using standard COM registry keys. The **HKEY_CLAS_ROOT\CLSID** key contains subkeys representing COM components. The subkeys are textual representations of class identifiers (CLSID). The **CLSID** subkeys contain an **InprocServer32** subkey with a default value. This value specifies the DLL path that implements the component. The **CLSID** subkeys also contain a **IsSIPInputMethod** subkey with a default value equal to the "1" string. The following table shows examples of IM registry values.

Key	Default value
HKEY_CURRENT_USER\ CLSID\ {4a4a96d7-ae04-11d0-a4f8-00aa00a749b9}	**"MS QWERTY IM"**
HKEY_CURRENT_USER\ CLSID\ {4a4a96d7-ae04-11d0-a4f8-00aa00a749b9}\ InprocServer32	**"\Windows\alphanum.dll"**
HKEY_CURRENT_USER\ CLSID\ {4a4a96d7-ae04-11d0-a4f8-00aa00a749b9}\ IsSIPInputMethod	**"1"**

Handwriting Recognition

As an alternative or companion to a keyboard, you can employ a handwriting pad IM in your application for processing user-drawn numbers, letters, characters, and symbols. The Windows CE handwriting recognition engine currently recognizes all 94 characters of the ASCII character set. In addition, the engine recognizes all of the available glyph characters, known as ideographs, for Chinese, Japanese, and Korean, and produces their corresponding Unicode output.

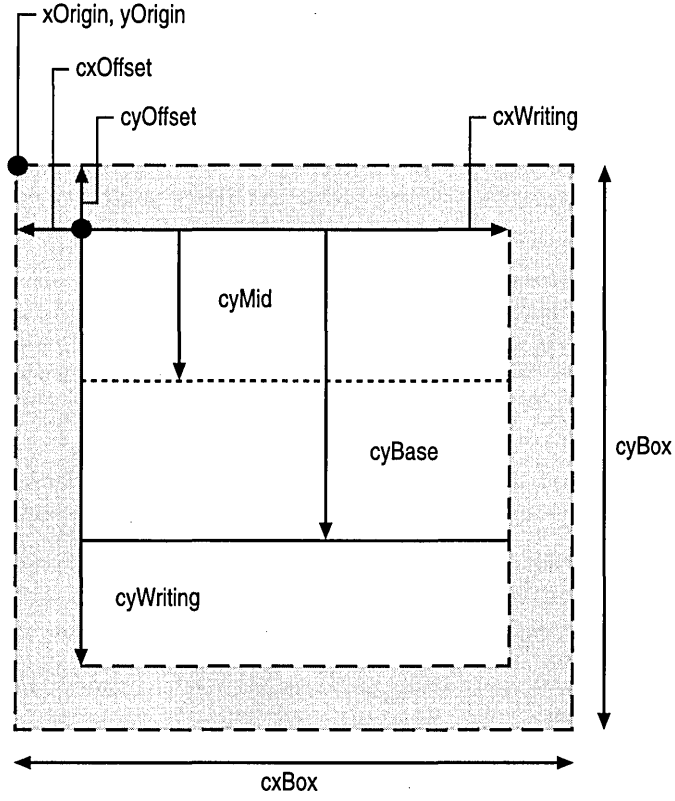
This section describes the implementation of the **Hwx** (pronounced *wix*) functions for processing handwritten characters that are entered by a user on the input panel.

Recognizing a Hand-Drawn Character

The recognition of hand-drawn alpha-numeric characters and glyphs occurs within the boundaries of a box, or multiple of boxes, that you define for the input panel of your target device. The structure of the applicable language determines how multiple boxes are arranged; for example, the boxes could be arranged from left to right and top to bottom for English. By moving a stylus in a predefined pattern within the box, the handwriting recognition engine employed by your application can process and recognize this input, one character or glyph at a time, to produce the corresponding Unicode output.

Setting Up the HWXGUIDE Structure

You define the size and position of the box or boxes used for character entry within the **HWXGUIDE** structure. For interpreting inked input coordinates, the **HwxSetGuide** function provides the recognition engine with the coordinates to the **HWXGUIDE** structure. The following illustration shows the dimensions of a character input box.



The base (*cyBase*) and midline (*cyMid*) in the **HWXGUIDE** structure are used primarily for characters and are not required for glyphs. Placement of a specific box within the target device display is defined by the *xOrigin* and *yOrigin* parameters. Offset parameters establish a buffer space between each box when multiple boxes are used to prevent a user from overwriting ink data from one box into another. In the event that you overwrite ink data, the recognition engine must determine to which box the ink data corresponds.

After initializing the handwriting recognition engine with **HwxConfig**, the entire recognition process begins and ends with the creation and destruction of a handwriting recognition context (HRC) object. Similar to a handle to a window or a handle to a device context, an HRC is a unique 32-bit handle that is used by the recognition engine for handwriting recognition. An HRC is used for input in a single box or multiple boxes. It carries all ink data and setup parameters that are necessary for recognizing a glyph, one character, or multiple characters. Use **HwxCreate** and **HwxDestroy**, respectively, to create and destroy an HRC object.

Processing User Input

To begin each input recognition session with a clean HRC, use the context from a pre-established HRC as a template. Each time you need a new HRC, pass the handle of the previous HRC object, excluding any previous ink data, to the recognition engine. This enables your application to save processing time by not having to recreate common HRC parameters that are used repeatedly in your application, such as **GUIDE** structures and alphabet codes.

The character recognition sequence begins when the input panel IM sends a call to **HwxInput**. This function adds ink to the HRC object to provide the **STROKEINFO** structure with the count of stylus stroke points that are generated. **HwxProcess** passes the HRC to the recognition engine where the stroke data is processed, based on predetermined character recognition parameters. When a user finishes entering stroke data, your application calls **HwxEndInput** to tell the recognition engine that no further ink will be added to the HRC.

To improve the performance of the recognition process, pass the context for the previous character to the recognition engine through **HwxSetContext**. If this function is not called, the recognition engine will assume that no previous context is available. **HwxSetContext** is called before **HwxProcess**. **HwxProcess** processes the ink received by the HRC to perform the recognition. Full character recognition occurs only after **HwxEndInput** is called. Results of the recognition are returned by **HwxGetResults**.

HwxGetResults simplifies the task of recognizing characters and glyphs that are drawn within the boundaries of the defined box by responding with character alternatives on a per-box basis in one call. The following code example sets the results for 10 boxes at a time, with 5 alternatives for each box.

```
HANDLE hMem = GlobalAlloc( GHND, 10 * (sizeof( BOXRESULTS )
        + (5-1) * sizeof( SYV )) );
LPBOXRESULTS rgBoxR = (LPBOXRESULTS)GlobalLock( hMem );
UINT indx = 0;
Do
{
   int iRes = HwxGetResults( hrc, 5, indx, 10, rgBoxR);
       .
       .    // Check for errors and use rgBoxR
       .
   indx += (UINT)iRes;
}
while (iRes == 10);
```

Once all input, processing, recognition, and output has been completed, destroy the active HRC with the **HwxDestroy** function.

Recognition Process

HwxALCValid and **HwxALCPriority** enable the recognition engine to identify a correct character match more rapidly and with a greater accuracy by establishing a prerequisite set of recognition parameters. Use **HwxALCValid** if you know what the user-entered values for a specified input field will be. For example, if your input field is part of a mailing address, such as numbers only for a United States Postal Code, you can use **HwxALCValid** with its *alc* parameter set to ALC_NUMERIC so that only number values are compared by the recognition engine to match the user input.

If the user-entered values for a specified field will not be limited to any one predictable set of alphabet codes, you can use **HwxALCPriority**. For the type of input expected, you can use **HwxALCPriority** to provide the recognition engine with a prioritized list of alphabet codes to use for comparison to user ink input.

When implemented in your application, multi-threaded processing improves the recognition engine's ability to communicate with the input thread, and thus speeds up the recognition process. This is because the recognition of a single character can take several seconds when using an order-independent process. In multi-threaded processing, the thread that gathers ink input can control the thread that is used for recognition. Optimally, this enables the recognition engine to produce partial results for display while the user writes on the writing pad.

Partial Recognition Process

Through a process called partial recognition, the recognition engine uses some or all of the user-entered stroke data to determine the resulting Unicode match. Windows CE uses partial recognition for glyphs in character sets with complex multi-stroke stylus input, such as Japanese, Chinese, and Korean. You can set up the **HwxSetPartial** function to accomplish partial recognition of glyphs similarly to how you set it up for character recognition. The exception is that your application calls the recognition engine multiple times for each glyph recognition. Each time a stroke is processed through the UI, you instruct the recognition engine to process by using the current amount of data.

Though not an absolute requirement, multi-threaded operation is the most efficient means for performing partial recognition. While one thread gathers character input, another can process data. When the two threads communicate with each other, the thread that performs character input can control the thread that performs the character recognition. This gives your application the capability to produce partial results while the user is still writing. The **HwxSetAbort** function enables the gathering thread to tell the processing thread to stop, saving processing time by not using an invalid result. In this basic process, **HwxSetAbort** passes the recognition engine an HRC and a pointer to an address that contains the number of strokes that are currently input by the user. If the contents of the address do not match the number of strokes that the recognition engine is trying to process, the recognition engine stops.

Performing Handwriting Recognition

The process of handwriting recognition requires your application to use the handwriting application programming interface (API) to accomplish the following sequence of events.

▶ **To start and end a handwriting recognition session**

1. Call **HwxConfig** to initialize the recognition engine for your application.

   This occurs only once for any specified application.
2. Create an HRC object by using **HwxCreate**.
3. Define the box or boxes used for processing user input by using **HwxSetGuide**.
4. Define recognition criteria by using **HwxALCValid** and **HwxALCPriority**.
5. Pass the previously-recognized character, if one exists, to the HRC with **HwxSetContext**.
6. Call HwxInput to send ink data to the HRC as the user writes.
7. Use **HwxProcess** to pass the HRC to the recognition engine for processing.

   Note Repeat step 7 for each input stroke when performing partial recognition.
8. Call **HwxResultsAvailable** to obtain the number of recognized characters.
9. Get the recognition engine results with **HwxGetResults**.
10. Invalidate the current HRC with **HwxDestroy** to complete the recognition process.

CHAPTER 8

Designing a User Interface for Windows CE

An application user interface (UI) serves two main purposes: to receive user input and to provide user output. How well your application handles these tasks depends on the target hardware platform capabilities, operating system (OS) configuration, and input/output (I/O) requirements.

Before designing your application, you need to ask some important questions about its interface:

- Will it include graphics?
- How will your application receive user input?
- Will users enter commands with a keyboard, with a touch screen, with voice commands, or with buttons on a console?
- How will you provide feedback to the user?
- Will your device support an LCD screen or audio feedback?

The Windows CE OS supports a range of devices, from the Handheld PC (H/PC) to embedded systems. Its modular feature design enables you to create applications that are suited for a specific platform. Because UI requirements vary, this chapter describes general design considerations for a graphical UI.

A well-designed UI focuses on users and their tasks. Good UI design considers general design principles as well as how graphics, color, and layout influence application usability. Consider the following design concepts when creating a user-focused UI:

- Give the user control

  Enable the user, not the computer or software, to initiate actions. Remember, the goal of the user is not to use the application, but to accomplish a task.

- Use familiar concepts

  To increase familiarity with the interface, enable users to manipulate representations of the tasks they perform. For example, if you provide a desktop-like interface, enable users to drag icons depicting documents to an icon depicting a trash can when deleting a file. For other types of interfaces, be sure that buttons and icons relate to the tasks they perform. For example, display a wrench icon to start an automotive maintenance application.

  Another way to increase user familiarity with the interface is to avoid using modes. Modes, which occur when identical commands or keystrokes perform different actions in different situations, force users to think about how the application works instead of the task at hand. Though modes are best avoided, warning boxes and message boxes are two necessary and appropriate mode types.

- Be consistent

  Consistency makes the interface familiar and predictable, which reduces user errors and improves performance. Consistency is enhanced with components that have a similar appearance and behavior and with actions that have the same result regardless of context. For example, in a desktop environment, scroll bars operate the same way, whether the scroll bar is in a list box or window. To achieve consistency, reuse standard commands in each task.

- Enable interactive discovery

  Empower the user to explore the interface through trial and error, but provide warnings about potential damage to the system or data. To minimize user problems, provide clear error messages and indicate appropriate actions to recover from an error. When possible, make actions reversible or recoverable.

- Provide feedback

  Present the user with timely visual and audio cues to confirm that the software is responding to input.

- Focus on aesthetics

  An attractive interface helps the user select appropriate information and suggests a quality application.

- Design with simplicity

  Simple interfaces, with an uncluttered display, are easy to learn and use. Show the most important controls directly on the interface and hide the rest in menus. Reduce the number of tasks presented in a single window or screen and group related tasks together.

- Support multiple input methods

  Provide multiple methods for performing an operation. To accomplish this, support multiple input devices if possible, and provide keyboard shortcuts or accelerators for specific tasks, if a keyboard is supported.

Designing Windows and Dialog Boxes

Many graphical UI use a desktop metaphor, which simplifies common file operations by presenting them in a familiar context. Depicting files as paper documents, directories as folders, and deleted items within a trash can are examples of the desktop metaphor. Though appropriate for most applications running on an H/PC or similar device, this metaphor might not be appropriate for some embedded systems, such as an automobile navigation application or a point-of-sale device. If the desktop metaphor is not appropriate for your application, use another suitable metaphor.

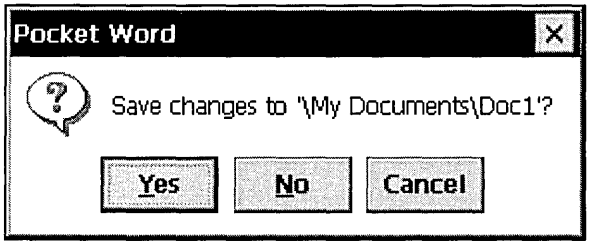
Whatever metaphor you choose, provide a context for your application. If you use the desktop metaphor, it is best to present objects in standard windows and dialog boxes. If you use a different metaphor, you could forgo using windows entirely and present objects only in dialog boxes. If you do use windows, and if your application's windows do not fill the entire screen, design the windows to be a fixed size because Windows CE does not support the resizing of windows by users.

Dialog boxes are secondary windows that contain controls and provide information to a user about actions. Windows CE supports three types of dialog boxes: application-defined dialog boxes, message boxes, and property sheets.

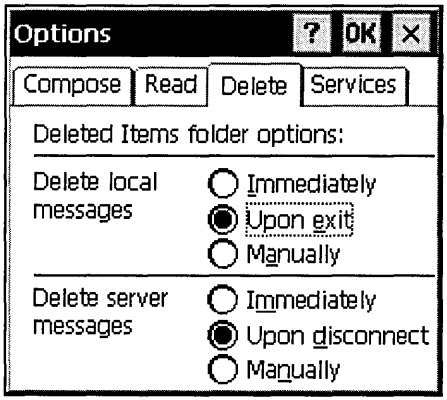
An application-defined dialog box helps users perform tasks specific to an application. It provides a great deal of flexibility by enabling you to place controls directly onto the body of the dialog box. This is especially useful when designing interfaces that do not use a desktop metaphor because you can design an entire application interface by using only application-defined dialog boxes to house controls. When using an application-defined dialog box, include only as many controls as are needed for your application and space them adequately.

An application-defined dialog box can be *modal* or *modeless*. A modal dialog box requires the user to supply information or close the dialog box before enabling the application to continue. A modeless dialog box enables the user to supply information and return to a previous task without closing the dialog box.

A *message box* is a modal dialog box that displays a message and prompts for user input. It typically contains a text message and one or more predefined buttons. The following screen shot shows a modal dialog box.



The following screen shot shows a *property sheet*, which is a collection of tabbed pages that enables a user to view and modify object properties.



In a desktop metaphor, a dialog box typically contains **OK** and **Cancel** commands, which initiate a user request or dismiss the window, respectively. In Windows CE, the **X** button represents both the **Close** and **Cancel** commands. Follow these guidelines for using the **X** and **OK** buttons in dialog boxes:

- When the **OK** and **X** buttons perform the same function, use the **OK** button, because users are more comfortable clicking the **OK** button than the **X** button to confirm an action. Be sure to disable the **X** button.
- Never place an **OK** button in both the command bar and the body of a dialog box because users might find this confusing. However, you can place a **Cancel** button in the body of a dialog box and an **X** button on the title bar.

Designing Menus

Menus are collections of commands, attribute selections, separators, and other selectable elements. All Windows CE menus are implemented as top-level, pop-up windows that do not support buttons. Although Windows CE supports owner-drawn menu items, it handles them as it would other menu items.

Windows CE does not support menu bars. Rather, it combines the features of a menu bar and a toolbar into one control called a *command bar*, which makes efficient use of the screen space available on Windows CE–based devices.

The following screen shot shows a command bar.

Windows CE supports four menu types:

- Pop-up

  A *pop-up menu* is a floating menu that displays commands that are specific to the object selected by the user or to the object's immediate context. A pop-up menu appears at the location on the screen where the user accessed it. It is typically used for common commands that rarely change in content and for items that require a small amount of screen space. It is recommended that you restrict the number of items in a pop-up menu to less than 10.

- Scrolling

  A *scrolling menu* is a menu that adds scroll arrows to enable a user to scroll the menu up and down if a menu exceeds the display area height. Scrolling menus are unique to Windows CE. With scrolling menus, you do not have to limit the size of a menu to the number of items that fit on screen.

- Cascading

  A *cascading menu* is a secondary menu or submenu that appears when a certain option is selected in the parent menu. A triangular arrow next to the parent item in a menu indicates a cascading menu. Windows CE displays cascading menus in alphabetical order. If the height of a cascading menu exceeds the maximum screen height, the menu adopts a multiple-column mode on an H/PC and scrolls on a Palm-size PC to show the remaining menu items. Use a cascading menu to group related menu items or when a choice leads to a short list of related options.

- Pull-down

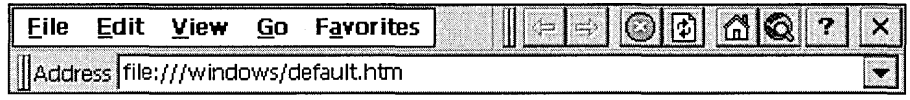
  A *pull-down menu* contains commands that are accessed from a command or menu bar. It is commonly used to display text, but it can also contain graphics, colors, and shading. When creating a pull-down menu, display all possible command choices on the menu. Items that cannot be chosen due to the application state should be dimmed. Use a pull-down menu to provide access to a small number of items with content that rarely changes.

Working with Command Bars

One of the challenges encountered when creating a Windows CE–based application is to design for a small screen. To maximize the screen space available for applications in the client area, the OS supports a new type of control—the *command bar*. The command bar is unique to Windows CE because it combines a menu bar, toolbar, and an optional address bar. Windows CE supports multiple command bars, each containing gripper controls that enable users to hide buttons and menus. Command bars can contain combo boxes, edit boxes, and buttons, as well as other types of controls. They also can include the **Close** (**X**) button, the **Help** (**?**) button, and the **OK** button.

Command bars vary from 240 pixels through 640 pixels in length depending on the screen resolution. It is recommended that you always display a command bar in Windows CE–based applications.

Command bars are composed of bands, separated by gripper controls. Each band can contain up to one child window, which can be a toolbar or any other control. The default is to display a toolbar. Additionally, each band can have its own bitmap, displayed as a background for the toolbar. A user can resize or reposition a band by dragging its gripper bar. If a band has a text label or icon next to its gripper bar, a user can maximize the band and restore it to its previous size by using the pointing device to choose the label or icon. The following screen shot shows a command bar.



A command bar menu includes a list of commands that drops down when a user taps the menu's caption on the command bar. Menu titles on a command bar appear in bold text. If you include a menu bar, always position it as the first (leftmost) element on the command bar. If you provide **File**, **Edit**, **View**, **Insert**, **Format**, **Tools**, and **Window** menus, place them in this order, from left to right. The menu titles appear as bold text surrounded by a rectangular frame.

Windows CE supports ToolTips for command bar and toolbar buttons, but not for menus or combo boxes on a command bar. ToolTips usually display only the title of a button command, but they can also display the shortcut key for the command.

You can place check boxes or radio buttons on the command bar to enable users to toggle between different views. Moving between views can make windows more readable by eliminating unnecessary scrolling. A command bar button can display both text and images. This enables you to include text as part of a button label to provide descriptions, which eliminates the need for ToolTips.

If you choose to place a label next to your edit control on a command bar, you have two choices. You can insert a static text field above or to the left of the control. Alternatively, you can include an edit control label inside the text field as the default text. In this case, you would enclose the label between angle brackets, as in the following example: <name>. Because the user can no longer see the control label when typing text in the field, it is recommended that you use a static text field.

If you provide individual **New**, **Open**, **Save**, and **Print** buttons on a command bar, you must position them in this order, from left to right. If you provide individual **Bold**, **Italic**, and **Underline** buttons, you must also place them in this order, from left to right. Always make buttons at least 23 x 23 pixels. If you plan to support touch interaction in which users use a finger rather than a stylus, make all buttons at least 38 x 38 pixels. However, to conserve space, consider creating a combo-box button instead of three or four separate buttons.

Choosing Controls

Windows CE supplies a set of controls that you can use to build an application. Controls commonly appear in dialog boxes, but they can also appear on toolbars and command bars. Windows CE supports many predefined controls, which can be divided into two categories: *window controls* and *common controls*. Window controls send the WM_COMMAND message and include buttons, check boxes, radio buttons, push buttons, group boxes, combo boxes, edit controls, list boxes, and static controls. Common controls send the WM_NOTIFY message and include all other controls. They are divided into the following sub-categories: foundation controls, file controls, scale controls, informational controls, and miscellaneous controls used for specific Windows CE–based platform features.

Due to the large number of controls available in Windows CE, determining which control to use in a specific situation can be difficult. When choosing a control, you must consider the type of input you are trying to capture, the abilities and limitations of the control, and the characteristics of your platform's screen.

The following table shows all predefined Windows CE controls and their uses.

Control	Description	Use
Check box	A two-part control consisting of a square box and text options. Each option acts as a switch that can be turned on and selected or turned off and cleared. When an item is turned on, a check mark appears within the square box; otherwise, the square box is empty. A user can select more than one option in a group of check boxes.	When setting properties, attributes, or values. When more than one choice can be selected. When ample screen space is available. When options do not change.
Radio button	A two-part control consisting of a small circle and text options. When an option is selected, the circle appears highlighted or filled. Only one option can be selected at one time in a group of radio buttons.	When setting properties, attributes, or values. When only one choice can be selected. When ample screen space is available. When options do not change.
Push button (command button)	A square or rectangle with a text or graphic label inside. When chosen, an application immediately performs the associated action or command.	To perform an action. To display a menu or window. To set a condition or property value. When ample screen space is available.
Group box	A rectangular frame that surrounds a group of controls.	To visually separate and relate groups of controls. To visually relate elements within a window.
Combo box	A control possessing the characteristics of both an edit control and a list box or drop-down list box. Information can be either typed into the edit control field or selected from items displayed in the list box.	When options are large in number and not frequently selected. When the list of options can change. When only one choice can be selected. To capture unlisted data. When users prefer to type information rather than select it in a list. When a form of input is available.
Drop-down list box	A rectangular box with an arrow button on the side. When the arrow button is selected, the box displays a hidden list of items that seems to drop down from a single item. If the list exceeds the box boundaries, scroll bars appear, enabling a user to view the remaining list.	When only one choice can be selected. When screen space is limited. When options are many and not frequently selected.
Edit control	A rectangular box in which information can be entered by the user or in which information is displayed for read-only purposes. Edit controls typically contain captions and can be designated as either single-line or multiple-line.	When options are difficult to categorize and vary in length. When an input method is available. When providing a list of options is not feasible.

Control	Description	Use
List box	A rectangular box containing a list of items from which either a single selection or multiple selections are made. Lists can contain either text or graphics. If the list exceeds the boundaries of the box, scroll bars appear, enabling a user to view the remaining items.	When options are large in number and not frequently selected. When screen space makes radio buttons or check boxes impractical. When the list of options might change. When ample screen space is available.
Scroll bar	A rectangular container consisting of a scroll area, a slider box, and arrows. Scroll bars are typically found on primary and secondary windows.	To view information that uses more than the available space.
Static control	A text field that displays read-only information.	To display a caption. To provide instructional information. To display descriptive information. To display a keyboard hot key for another control.

The following table shows foundation controls, which are used to contain or manage other controls.

Control	Description	Use
Command band	A special kind of rebar control. It has a fixed band at the top containing a toolbar with an optional **Close** (**X**) button, **OK** button, and **Help** (**?**) button, in the right corner. By default, each band in the command bands control contains a command bar. You can override this if you want a band to contain some other type of child window.	To provide easy access to frequently used commands or options. When screen space is limited.
Command bar	A toolbar that combines a menu bar as well as the **Close** (**X**) button, an **OK** button, and optionally, the **Help** (**?**) button. A command bar can contain menus, combo boxes, buttons, and separators. A separator is a blank space that you can use to divide other elements into groups or to reserve space in a command bar.	To provide easy access to frequently used commands or options. When screen space is limited.
Toolbar	A panel that contains a set of controls.	To provide easy access to frequently used commands or options.

Control	Description	Use
Property sheet	A control to define property sheets. It accepts dialog box layout specifications and automatically creates tabbed property pages.	When creating property sheets.
Tab control	A tab control resembles a divider in a notebook and is used to define sections of information within the same window.	To present repetitive, related information. To present options or settings that can be applied to one object.
Rebar	A control that acts as a container for a child window. It contains one or more bands; each band can contain one child window, which can be a toolbar or any other control. Each band can have its own bitmap, which is displayed as a background for the toolbar on that band. A user can resize or reposition a band by dragging its gripper bar. If a band has a text label next to its gripper bar, a user can maximize the band and restore it to its previous size.	When screen space is limited. To hide and show portions of a command bar.

The following table shows file controls, which are used to display files.

Control	Description	Use
Header control	A heading above a column of text or numbers that can be divided into two or more parts for multiple columns. Each part can operate like a command button to support a different function.	To display text and graphics. To aid the user in sorting or sizing columns of information.
Image list	A list box that contains a collection of images that are all the same size, such as bitmaps or icons. Image lists manage images, but do not display them. They are designed to be used with toolbar buttons, list view controls, and tree view controls.	When the displaying of icons or images is appropriate. When you implement a toolbar, treeview control, or list view control.
Tree view	A list box that displays a hierarchical set of labeled items as an indented outline. It includes buttons that enable the outline to be expanded and contracted.	To display a relationship between a set of containers. When ample screen space is available.
List view	A list box that displays a collection of files or folders consisting of an icon and a label. Selection and navigation in this control work similarly to that in a folder window.	When the displaying of icons is appropriate. When ample screen space is available.

Control	Description	Use
Spin box	An edit control with an associated spin button control. A spin box enables the user to select an option by scrolling through a small list or by typing an item in the edit control field.	When options are infrequently selected and small in number. When screen space is limited. When there are too many options to display in a combo box or list box.When only one choice can be selected.
Trackbar (slider)	A bar with tick marks on it and a slider or thumb. The tick marks represent a range of values. When a user drags the slider arm, it moves in the appropriate direction, tick by tick.	To set an attribute. When only one choice can be selected. When a limited range of possible settings exists. When options are incremented. When ample screen space is available.

The following table shows informational controls, which are used to provide information about tools, processes, or time.

Control	Description	Use
Progress bar	A display-only control that consists of a rectangular bar that fills from left to right.	To provide visual feedback concerning completion of a process. When ample screen space is available.
Date and time picker	A control that provides users with an easy way to modify date and time information. Each field in the control displays a time element, such as month, day, hour, or minute.	To modify date and time information. When screen space is limited.
Status bar	An area within a window, typically at the bottom, that displays information. It can contain display-only controls.	To provide information about the current state of what is being viewed in the window. To provide a descriptive message about a selected menu or toolbar.
Month calendar	A child window that displays a monthly calendar. The calendar can display one or more months at a time.	To select date information. When screen space is limited.
ToolTip	A small pop-up window containing information about a control. A ToolTip appears when a stylus is held on a control that supports ToolTips.	To supply information about a control. To reduce screen clutter caused by control captions.

The following table shows miscellaneous controls, which are used for specific Windows CE–based platform features.

Control	Description	Use
HTML viewer	A control that provides the features required to implement Microsoft Pocket Internet Explorer Internet browser or Windows CE Help.	To view Hypertext Markup Language (HTML) text and embedded images.
Rich Ink	A control that captures stylus motions in order to emulate the act of writing or drawing on paper. The control's document view, under the touch screen, serves as electronic paper. In addition to capturing images, Rich Ink also has editing and formatting capabilities.	To accept user input without using a keyboard.
Voice recorder	A control that provides recording and playback features.	To support voice recording and playback.

In addition to predefined controls, Windows CE supports a new custom draw service. The custom draw service is not a predefined control; it is a service that makes it easy to customize a common control's appearance. You can use the custom draw service to change a common control's color or font or to partially or completely draw the control.

In addition to Windows CE controls, you can also create custom controls. When designing custom controls, avoid the following:

- Controls that are difficult to use

  Make controls easy to use. For example, make controls larger, use colors that contrast with the screen background, and remove nearby controls and unnecessary images. Additionally, when you design a control, have a variety of people test its usability.

- Controls that are too close together

  Controls should be spaced so that users do not accidentally select one control while intending to select another.

- Controls that are hard to interpret

  A control should represent its corresponding function. For example, place an image of a scissors on a button control used to "cut" text.

- Controls that are hard to distinguish

  Controls should have easily recognizable differences. When you have several similar controls close together, users can confuse them. Distinguish controls by using a unique size, position, shape, and contrast for each.

- Controls that are hidden

  Controls should be obvious. If you want to hide a control, place it where users expect to find it, such as in a menu. Controls that are occasionally unavailable should be disabled and dimmed, not hidden, unless screen space is severely limited.

- Controls that are not predictable

  Controls that have the same function should operate the same way regardless of where they are placed. If a control uses a different operating principle, design the control so that it will not be confused with controls that operate differently.

Using Color and Grayscale Palettes

Designers often rely on color to make an application visually pleasing. However, using color randomly or excessively can affect usability. To use color effectively, keep the following guidelines in mind when designing a UI:

- Display no more than four colors on a single screen at one time and limit the colors for your entire application to fewer than eight.
- Use color in combination with other emphasis techniques to distinguish areas on the interface and identify important features. Never use color alone to distinguish elements because users might have difficulty distinguishing colors under various lighting conditions. Also use fonts, icons, screen placement, or patterns to distinguish screen elements.
- Avoid spectrally opposite color combinations, such as red and blue or yellow and purple; they can make images appear blurred.
- Design applications primarily for a grayscale display. Many users might not have color displays. Then, when the application is complete, add color.
- Use color contrast for extended viewing; dim colors might not be discernable once a user's eyes adapt to the color.
- Avoid colors lacking contrast as well as colors of equal brightness; they are not easily distinguished.
- Use black, white, and gray to improve resolution.
- Use common color associations, such as red for stop or green for go, to enhance familiarity.

The color design model for Windows CE uses a 16-color Windows palette, based on the Windows-based color scheme, and is measured in bits per pixel (bpp). Windows CE supports pixel formats of 1, 2, 4, 8, 16, 24, and 32 bpp. Your application should determine the color format that is supported by a display device, and then adopt a complimentary display strategy.

Note An 8-bpp display driver can display a 32-bpp device-independent bitmap (DIB) by mapping each color in the DIB color table to a specific color on the device. The palette available in the application displaying the bitmap determines what mapping is used. The application can lose color information if it does not use an appropriate palette or if a bitmap uses more colors than the palette can hold.

The following illustration shows a standard 16-color palette converted to grayscale.

Color:	Red	Green	Blue
White	255	255	255
Teal	0	255	255
Purple	255	0	255
Blue	0	0	255
Light grey	192	192	192
Dark grey	128	128	128
Dark teal	0	128	128
Dark purple	128	0	128
Dark blue	0	0	128
Yellow	255	255	0
Green	0	255	0
Dark yellow	128	128	0
Dark green	0	128	0
Red	255	0	0
Dark red	128	0	0
Black	0	0	0

Some Windows CE–based devices support only a 2-bpp palette, with four grayscale colors: black, white, light gray, and dark gray. On a grayscale display, a single-pixel graphical element, such as a dot or a line, can be difficult to distinguish without an adjacent high contrast color. For example, white and light gray elements can be hard to see unless presented against a black or dark gray background.

Likewise, light colors might be difficult to distinguish. When using light colors, double the thickness of pixels or lines to strengthen them. Light gray works well for creating a shadow effect around large controls on a white background and for anti-aliasing, which adds colored pixels to a graphic to smooth jagged edges. If you use light gray as a background color for your screen, use a white line to visually separate key areas, such a command bar or owner-drawn menu, from other areas of the screen.

Windows CE does not arbitrate between the palettes of the background and foreground applications. Because of this, you should use only the first 10 and last 10 colors included in the stock palette of a display device, which generally are the standard Windows video graphic adapter (VGA) colors.

Creating Icons and Bitmaps

In a graphical UI, icons convey attributes or tasks. An effective icon clearly represents its function and is easy to remember; an ineffective icon reduces the usability of an application by making it appear obscure and unapproachable.

Icons are used in different ways. They can either resemble what they represent—for example, a book that is used to represent a dictionary—or they can represent a characteristic, such as a gas pump to represent a gas station. Icons can also be symbolic representations, which might or might not be clear to the user. An example of this type of icon is the magnifying glass, which is used on Windows-based desktop platforms to signify a search feature.

Icons most often are used on buttons, but they can be used for progress indicators as well. When a Windows CE color icon has a Windows-based desktop platform application equivalent, both icons use the same design and color. However, you must create a 16-color version and a grayscale version of the icon to ensure that it displays correctly on both color and 2-bpp devices.

Note The icon editor in the Microsoft Windows CE Toolkit for Microsoft Visual C++ version 5.0 or later can create icon (.ico) files that retain both 16-color and 2-bpp gray versions of an icon.

In addition to using Windows-based desktop platform application icon equivalents, you can create your own icons by using the standard Windows-based 16-color palette. To add depth to an icon, use highlights and shadows; however, recall that the icons you create must translate correctly to 2 bpp gray if the target device supports both grayscale and color displays.

The following table shows how the 16-color palette translates to 4 grays.

Color	Red	Green	Blue	Gray conversion
Black	0	0	0	Black
White	255	255	255	White
Dark gray	128	128	128	Dark gray
Light gray	192	192	192	Light gray
Dark red	128	0	0	Black
Red	255	0	0	Dark gray
Dark yellow	128	128	0	Dark gray
Yellow	255	255	0	Light gray
Dark green	0	128	0	Black
Green	0	255	0	Dark gray
Dark cyan	0	128	128	Dark gray
Cyan	0	255	255	Light gray
Dark blue	0	0	128	Black
Blue	0	0	255	Dark gray
Dark magenta	128	0	128	Dark gray
Magenta	255	0	255	Light gray

User Input Devices

Input devices enable users to interact with the UI. Windows CE supports several types of user input devices, such as a keyboard, a mouse, a touch screen, a stylus, and voice recognition, though input devices that are available on a specific device might vary. For general design considerations for user input devices, see *The Windows Interface Guidelines for Software Design.*

Providing User Feedback

In addition to receiving user input, a UI provides feedback by displaying *messages*. Messages are communications to the user that are displayed on the screen. They inform the user of system status or prompt the user to complete some action. Effective messages are clear and concise.

When you include an identification number in message text, place it at the end of the message text and not in the title bar or at the beginning of the text where it might decrease the user's ability to quickly read the message.

APPENDIX A

Window and Control Styles

Window and control styles are attributes that are controlled by specific style flags. There are also extended styles that have their own set of flags.

Window and Message Box Styles

The following table shows window styles that are supported by Windows CE.

Basic window styles	Description
WS_CHILD	Specifies a child window. This should not be changed after the window is created.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing occurs within the parent window. This style is used on the parent window. Windows CE–based windows always have the WS_CLIPCHILDREN style.
WS_CLIPSIBLINGS	Excludes the area that is occupied by sibling windows above a window.
WS_DISABLED	Specifies a window that is initially disabled. A disabled window cannot receive input from the user.
WS_EX_NOACTIVATE	Specifies that a window cannot be activated. If a child window has this style, tapping it does not cause its top-level parent to activate. Although a window that has this style will still receive stylus events, neither it nor its child windows can get the focus. This style is supported only by the Windows CE operating system (OS).
WS_EX_NOANIMATION	Prevents a window from showing animated exploding and imploding rectangles and from having a button on the taskbar. This style is supported only by Windows CE.

Basic window styles	Description
WS_EX_NODRAG	Specifies a stationary window that cannot be dragged by its title bar. This style is supported only by Windows CE.
WS_EX_TOPMOST	Creates a window that will be placed and remain above all non-topmost windows. To add or remove this style, use the **SetWindowPos** function.
WS_GROUP	Specifies the first control of a group of controls. This style is used primarily when creating dialog boxes. The group consists of this first control and all controls that are defined after it, up to the next control for which the WS_GROUP style is specified. Because the first control in each group often has the WS_TABSTOP style, a user can move from group to group.
WS_POPUP	Specifies a pop-up window. This style should not be changed after the window is created.
WS_TABSTOP	Specifies a control that can receive the keyboard focus when the user presses the TAB key. This style is used primarily when creating controls in a dialog box. Pressing the TAB key changes the keyboard focus to the next control with the WS_TABSTOP style.
WS_VISIBLE	Specifies a window that is initially visible. This style can be turned on and off to change window visibility.

Non-client area styles	Description
WS_BORDER	Specifies a window with a thin-line border.
WS_CAPTION	Specifies a window with a title bar and border.
WS_DLGFRAME	Specifies a window with a dialog box border style. A window with this style cannot have a title bar.
WS_EX_CAPTIONOKBTN	Includes an **OK** button in the title bar.
WS_EX_CLIENTEDGE	Specifies a window with a border that has a sunken edge.
WS_EX_CONTEXTHELP	Includes a **Help** button (?) in the title bar of the window.
WS_EX_DLGMODALFRAME	Specifies a window with a double border.
WS_EX_OVERLAPPEDWINDOW	Combines the WS_EX_CLIENTEDGE and WS_EX_WINDOWEDGE styles.
WS_EX_STATICEDGE	Specifies a window with a three-dimensional border style. This style should be used for items that do not accept user input.

Non-client area styles	Description
WS_EX_WINDOWEDGE	Specifies a window border with a raised edge.
WS_HSCROLL	Specifies a window with a horizontal scroll bar.
WS_OVERLAPPED	Specifies a window with the WS_BORDER and WS_CAPTION styles.
WS_SYSMENU	Specifies a window with a window menu on its title bar. Use in conjunction with the WS_CAPTION style. Windows CE does not have a system menu, but you can use the WS_SYSMENU style to add the standard **Close (X)** button to a window title bar.
WS_VSCROLL	Specifies a window with a vertical scroll bar.

The following table shows message box styles that are supported by Windows CE.

Button styles	Description
MB_ABORTRETRYIGNORE	Specifies that the message box contains three buttons: **Abort**, **Retry**, and **Ignore**.
MB_DEFBUTTON1	Specifies that the first button is the default button. The first button is always the default unless you specify MB_DEFBUTTON2.
MB_DEFBUTTON2	Specifies that the second button is the default button.
MB_DEFBUTTON3	Specifies that the third button is the default button.
MB_OK	Specifies that the message box contains one button: **OK**.
MB_OKCANCEL	Specifies that the message box contains two buttons: **OK** and **Cancel**.
MB_RETRYCANCEL	Specifies that the message box contains two buttons: **Retry** and **Cancel**.
MB_YESNO	Specifies that the message box contains two buttons: **Yes** and **No**.
MB_YESNOCANCEL	Specifies that the message box contains three buttons: **Yes**, **No**, and **Cancel**.

Icon styles	Description
MB_ICONASTERISK MB_ICONINFORMATION	Includes an icon consisting of a lowercase letter *i* in a circle in the message box.
MB_ICONERROR MB_ICONHAND MB_ICONSTOP	Includes a stop-sign icon in the message box.
MB_ICONEXCLAMATION MB_ICONWARNING	Includes an exclamation-point icon in the message box.
MB_ICONQUESTION	Includes a question-mark icon in the message box.

Window styles	Description
MB_APPLMODAL	Specifies that the user must respond to the message box before continuing work in the window that is identified by the *hWnd* parameter. However, the user can move to the windows of other applications and work in those windows. Depending on the hierarchy of windows in the application, the user might be able to move to other windows within the application. All child windows of the message box's parent window are automatically disabled, but pop-up windows are not. MB_APPLMODAL is the default value. Windows CE does not support MB_SYSTEMMODAL or MB_TASKMODAL.
MB_SETFOREGROUND	Specifies that the message box becomes the foreground window.
MB_TOPMOST	Specifies that the message box is created with the WS_EX_TOPMOST window style.

Control Styles

The following table shows window control styles that are supported by Windows CE.

Check box styles	Description
BS_3STATE	Creates a check box in which the box can be unavailable as well as selected or cleared. Use the unavailable state to show that the state of the check box is not determined.
BS_AUTO3STATE	Creates a three-state check box in which the state cycles through selected, unavailable, and cleared each time the user selects the check box.
BS_AUTOCHECKBOX	Creates a check box in which the check state switches between selected and cleared each time the user selects the check box.
BS_CHECKBOX	Creates a small, empty check box with a label displayed to the right of it. To display the text to the left of the check box, combine this flag with the BS_RIGHTBUTTON style.

Check box styles	Description
BS_LEFT	Left-aligns the text in the button rectangle on the right side of the check box.
BS_RIGHT	Right-aligns text in the button rectangle on the right side of the check box.
BS_RIGHTBUTTON	Positions a check box square on the right side of the button rectangle.
WS_TABSTOP	Turns the control into a tab stop, which enables the user to select the control by tabbing through the controls in a dialog box.

Combo box styles	Description
CBS_AUTOHSCROLL	Automatically scrolls the text in an edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is enabled.
CBS_DISABLENOSCROLL	Shows a disabled vertical scroll bar in the list box when the box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.
CBS_DROPDOWN	Displays only the edit control by default. The user can display the list box by selecting an icon next to the edit control.
CBS_DROPDOWNLIST	Displays a static text field that displays the current selection in the list box.
CBS_LOWERCASE	Converts to lowercase any uppercase characters that are typed into the edit control of a combo box.
CBS_NOINTEGRALHEIGHT	Specifies that the combo box will be exactly the size specified by the application when it created the combo box. Usually, Windows CE sizes a combo box so that it does not display partial items.
CBS_OEMCONVERT	Converts text typed in the combo box edit control from the Windows CE character set to the OEM character set and then back to the Windows CE set. This style is most useful for combo boxes that contain file names. It applies only to combo boxes created with the CBS_DROPDOWN style.
CBS_SORT	Sorts strings that are typed into the list box.
CBS_UPPERCASE	Converts to uppercase any lowercase characters that are typed into the edit control of a combo box.
WS_TABSTOP	Turns control into a tab stop, which enables the user to select the control by tabbing through the controls in a dialog box.

Edit control styles	Description
ES_AUTOHSCROLL	Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to the zero position.
ES_AUTOVSCROLL	Scrolls text up one page when the user presses the ENTER key on the last line.
ES_CENTER	Centers text in a multiline edit control.
ES_COMBOBOX	Indicates that the edit control is part of a combo box
ES_LEFT	Left-aligns text.
ES_LOWERCASE	Converts all characters to lowercase as they are typed into the edit control.
ES_MULTILINE	Designates a multiline edit control. The default is a single-line edit control. When the multiline edit control is in a dialog box, the default response to pressing the ENTER key is to activate the default button. To use the ENTER key as a carriage return, use the ES_WANTRETURN style. When the multiline edit control is not in a dialog box and the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If you do not specify ES_AUTOVSCROLL, the edit control shows as many lines as possible and beeps if the user presses the ENTER key when no more lines can be displayed. If you specify the ES_AUTOHSCROLL style, the multiline edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If you do not specify ES_AUTOHSCROLL, the control automatically wraps words to the beginning of the next line when necessary. A new line is also started if the user presses the ENTER key. The window size determines the position of the word wrap. If the window size changes, the word wrapping position changes and the text is redisplayed. Multiline edit controls can have scroll bars. An edit control with scroll bars processes its own scroll bar messages. Edit controls without scroll bars scroll as described in the previous paragraphs and process any scroll messages that are sent by the parent window.

Edit control styles	Description
ES_NOHIDESEL	Negates the default behavior for an edit control. The default behavior hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. If you specify ES_NOHIDESEL, the selected text is inverted, even if the control does not have the focus.
ES_NUMBER	Accepts into the edit control only digits to be typed.
ES_OEMCONVERT	Converts text typed in the edit control from the Windows CE character set to the OEM character set and then converts it back to the Windows CE set. This style is most useful for edit controls that contain file names.
ES_PASSWORD	Displays an asterisk (*) for each character that is typed into the edit control. You can use the EM_SETPASSWORDCHAR message to change the displayed character.
ES_READONLY	Prevents the user from typing or editing text in the edit control.
ES_RIGHT	Right-aligns text in a multiline edit control.
ES_UPPERCASE	Converts all characters to uppercase as they are typed into the edit control.
ES_WANTRETURN	Specifies that a carriage return be inserted when the user presses the ENTER key while typing text into a multiline edit control in a dialog box. If you do not specify this style, pressing the ENTER key has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control.
WS_TABSTOP	Turns the control into a tab stop, which enables the user to select the control by tabbing through the controls in a dialog box.
WS_TABSTOP	Turns the control into a tab stop, which enables the user to select the control by tabbing through the controls in a dialog box.

List box styles	Description
LBS_DISABLENOSCROLL	Shows a disabled vertical scroll bar for the list box when the box does not contain enough items to scroll. If you do not specify this style, the scroll bar is hidden when the list box does not contain enough items.
LBS_EXTENDEDSEL	Enables the user to select multiple items by using the SHIFT key and the mouse or hot keys.
LBS_MULTICOLUMN	Specifies a multicolumn list box that the user scrolls through horizontally. You set the width of the columns by using the LB_SETCOLUMNWIDTH message.
LBS_MULTIPLESEL	Turns string selection on or off each time a user taps or double-taps a string in the list box. A user can select any number of strings simultaneously.
LBS_NOINTEGRALHEIGHT	Specifies that the list box will be exactly the size specified by the application when it created the list box. Usually, Windows CE sizes a list box so that it does not display partial items.
LBS_NOREDRAW	Ensures that the list box appearance is not automatically updated when changes are made. You can change this style by sending a WM_SETREDRAW message.
LBS_NOSEL	Specifies that the user can view list box strings but cannot select them.
LBS_NOTIFY	Notifies the parent window when the user taps or double-taps a string in the list box.
LBS_SORT	Sorts strings in the list box alphabetically.
LBS_STANDARD	Sorts strings in the list box alphabetically. The parent window receives an input message when the user taps or double-taps a string. The list box has borders on all sides.
LBS_USETABSTOPS	Enables a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog box units. A dialog box unit is equal to one-fourth of the current dialog box base-width unit. Windows CE calculates these units based on the height and width of the current system font.
LBS_WANTKEYBOARDINPUT	Specifies that the owner of the list box receives WM_VKEYTOITEM messages when the user presses a key and the list box has the input focus. This enables an application to perform special processing on the keyboard input.
WS_TABSTOP	Turns the control into a tab stop, which enables the user to select the control by tabbing through the controls in a dialog box.

Push button styles	Description
BS_BOTTOM	Places the text at the bottom of the button rectangle.
BS_CENTER	Centers the text horizontally in the button rectangle.
BS_DEFPUSHBUTTON	Creates a push button with a heavy black border. If the button is in a dialog box, the user can select the button by pressing the ENTER key, even when the button does not have the input focus. This style is useful for enabling the user to quickly select the most likely option, or default.
BS_LEFT	Left-aligns the text in the button rectangle.
BS_NOTIFY	Enables a button to send BN_DBLCLK, BN_KILLFOCUS, and BN_SETFOCUS notification messages to its parent window. Note that the button sends the BN_CLICKED notification message regardless of whether it has this style.
BS_OWNERDRAW	Creates an owner-drawn button. The owner window receives a WM_MEASUREITEM message when the button is created and a WM_DRAWITEM message when a visual aspect of the button has changed.
BS_PUSHBUTTON	Creates a push button that posts a WM_COMMAND message to the owner window when the user clicks the button.
BS_RIGHT	Right-aligns text in the button rectangle.
BS_TOP	Places text at the top of the button rectangle.
BS_VCENTER	Vertically centers text in the button rectangle.
WS_TABSTOP	Turns the control into a tab stop, which enables the user to select the control by tabbing through the controls in a dialog box.

Radio button styles	Description
BS_AUTORADIOBUTTON	Creates a radio button that, when selected by a user, clears all other buttons in the same group.
BS_LEFT	Left-aligns the text in the button rectangle on the right side of the check box.
BS_RADIOBUTTON	Creates a small circle with a label displayed to the right of it. To display the text to the left of the circle, combine this flag with the BS_RIGHTBUTTON style.

Radio button Styles	Description
BS_RIGHT	Right-aligns the text in the button rectangle on the right side of the check box.
BS_RIGHTBUTTON	Positions a check box square on the right side of the button rectangle.
WS_TABSTOP	Turns the control into a tab stop, which enables the user to select the control by tabbing through the controls in a dialog box.

Scroll bar styles	Description
SBS_HORZ	Designates a horizontal scroll bar. If you do not specify the SBS_TOPALIGN style, the scroll bar has the height, width, and position specified by the parameters of the **CreateWindow** function.
SBS_VERT	Designates a vertical scroll bar. If you do not specify the SBS_LEFTALIGN style, the scroll bar has the height, width, and position specified by the parameters of the **CreateWindow** function.

Static control styles	Description
SS_BITMAP	Specifies that a bitmap will be displayed in the static control. The text is the name of a bitmap that is defined elsewhere in the resource file, not a file name. The style ignores the *nWidth* and *nHeight* parameters; the control automatically sizes itself to accommodate the bitmap.
SS_CENTER	Specifies a simple rectangle and centers the error value text in the rectangle. Windows CE formats the text before display. The control automatically wraps words that extend past the end of a line to the beginning of the next centered line.
SS_CENTERIMAGE	Specifies that the midpoint of a static control with the SS_BITMAP style will remain fixed when you resize the control. The four sides are adjusted to accommodate a new bitmap. If the bitmap is smaller than the control's client area, the rest of the client area is filled with the color of the pixel in the upper-left corner of the bitmap.
SS_ICON	Specifies that an icon will be displayed in the static control. The text is the name of an icon defined elsewhere in the resource file, not a file name. The style ignores the *nWidth* and *nHeight* parameters; the icon automatically sizes itself.

Static control styles	Description
SS_LEFT	Specifies a rectangle and left-aligns the text in the rectangle. Windows CE formats the text before display. The control automatically wraps words that extend past the end of a line to the beginning of the next left-aligned line.
SS_LEFTNOWORDWRAP	Specifies a rectangle and left-aligns the text in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.
SS_NOPREFIX	Prevents interpretation of any ampersand (&) characters in the control's text as accelerator prefix characters. An application can combine SS_NOPREFIX with other styles by using the bitwise OR (|) operator. This can be useful when file names or other strings that might contain an ampersand (&) must be displayed within a static control in a dialog box.
SS_NOTIFY	Sends the parent window the STN_CLICKED notification when the user clicks the control.
SS_RIGHT	Specifies a rectangle and right-aligns the specified text in the rectangle. Windows CE formats the text before display. The control automatically wraps words that extend past the end of a line to the beginning of the next right-aligned line.

The following table shows common control styles that are supported by Windows CE.

Basic common control styles	Description
CCS_ADJUSTABLE	Enables a toolbar's built-in customization features, which enable the user to drag a button to a new position or to remove a button by dragging it off the toolbar. In addition, the user can double-click the toolbar to display the **Customize Toolbar** dialog box, which enables the user to add, delete, and rearrange toolbar buttons.
CCS_BOTTOM	Causes the control to position itself at the bottom of the parent window's client area and sets the width of the control to be the same as the parent window's width. Status windows have this style by default.
CCS_LEFT	Causes the control to display vertically on the left side of the parent window.

Basic common control styles	Description
CCS_NODIVIDER	Prevents a 2-pixel highlight from being drawn at the top of the control.
CCS_NOMOVEX	Causes the control to resize and move itself vertically, but not horizontally, in response to a WM_SIZE message. This message does not apply if your control has the CCS_NORESIZE style.
CCS_NOMOVEY	Causes the control to resize and move itself horizontally, but not vertically, in response to a WM_SIZE message. Header windows have this style by default. This style does not apply if your control has the CCS_NORESIZE style.
CCS_NOPARENTALIGN	Prevents the control from automatically moving to the top or bottom of the parent window. Instead, the control keeps its position within the parent window despite changes to the size of the parent. If the application also uses the CCS_TOP or CCS_BOTTOM styles, it adjusts the height to the default, but does not change the position and width of the control.
CCS_NORESIZE	Prevents the control from using the default width and height when setting its initial size or a new size. Instead, the control uses the width and height that is specified in the request for creation or sizing.
CCS_RIGHT	Causes the control to display vertically on the right side of the parent window.
CCS_TOP	Causes the control to position itself at the top of the parent window client area and matches the width of the control to the width of the parent window. Toolbars have this style by default.
CCS_VERT	Causes the control to display vertically.

Date and time picker styles	Description
DTS_APPCANPARSE	Enables the owner to parse user input. When a date time picker (DTP) control has this style, a user can make changes within the client area of the control by pressing the F2 key. The control sends a DTN_USERSTRING notification message when the user is finished editing.
DTS_LONGDATEFORMAT	Displays the date in long format. The default format string for this style is defined by LOCALE_SLONGDATEFORMAT, which produces output like "Friday, April 19, 1998."
DTS_SHORTDATEFORMAT	Displays the date in short format. The default format string for this style is defined by LOCALE_SSHORTDATE, which produces output like "4/19/98."
DTS_SHOWNONE	Enables the control to accept "no date" as a valid selection state. This state can be set with the DTM_SETSYSTEMTIME message or verified with the DTM_GETSYSTEMTIME message.
DTS_TIMEFORMAT	Displays the time. The default format string for this style is defined by LOCALE_STIMEFORMAT, which produces output like "5:31:42 PM." An up-down control is placed to the right of the DTP control to modify time values.
DTS_UPDOWN	Places an up-down control to the right of a DTP control to modify time values. This style can be used instead of the drop-down month calendar, which is the default style.

Header control styles	Description
HDS_BUTTONS	Causes each header item to look and behave like a button. This style is useful if an application carries out a task when the user clicks an item in the header control.
HDS_DRAGDROP	Enables drag-and-drop reordering of header items.
HDS_FULLDRAG	Causes the header control to display column contents even while a user resizes a column.
HDS_HIDDEN	Creates a header control that you can hide by setting its height to zero. This style is useful when you use the control as an information container instead of a visual control.
HDS_HORZ	Creates a horizontal header control.

List view styles	Description
LVS_ALIGNLEFT	Specifies that items are left-aligned in icon view and small icon view.
LVS_ALIGNTOP	Specifies that items are aligned with the top of the list view control in icon view and small icon view.
LVS_AUTOARRANGE	Specifies that icons automatically remain arranged in icon view and small icon view.
LVS_EDITLABELS	Enables item text to be edited in place. The parent window must process the LVN_ENDLABELEDIT notification message.
LVS_EX_CHECKBOXES	Enables items in a list view control to be displayed as check boxes. This style uses item state images to produce the check box effect.
LVS_EX_FULLROWSELECT	Specifies that when an item is selected, the item and all of its subitems are highlighted. This style is available only in conjunction with the LVS_REPORT style.
LVS_EX_GRIDLINES	Displays gridlines around items and subitems. This style is available only in conjunction with the LVS_REPORT style.
LVS_EX_HEADERDRAGDROP	Enables drag-and-drop reordering of columns in a list view control. This style is only available to list view controls that use the LVS_REPORT style.
LVS_EX_SUBITEMIMAGES	Enables images to be displayed for subitems. This style is available only in conjunction with the LVS_REPORT style.
LVS_ICON	Specifies icon view.
LVS_LIST	Specifies list view.
LVS_NOCOLUMNHEADER	Specifies that no column header is displayed in report view, which is the default view.
LVS_NOLABELWRAP	Displays item text on a single line in icon view. By default, item text might wrap in icon view.
LVS_NOSCROLL	Disables scrolling, so all items must be displayed within the client area.
LVS_NOSORTHEADER	Specifies that column headers do not work like buttons. This style is useful if clicking a column header in report view does not carry out any action, such as sorting.

List view styles	Description
LVS_OWNERDATA	Creates a virtual list view control.
LVS_OWNERDRAWFIXED	Enables the owner window to paint items in report view. The list view control sends a WM_DRAWITEM message to paint each item; it does not send separate messages for each subitem. The *itemData* member of the **DRAWITEMSTRUCT** structure contains the item data for the specified list view item.
LVS_REPORT	Specifies report view.
LVS_SHAREIMAGELISTS	Specifies that the control does not destroy the image lists assigned to it when it is destroyed. This style enables the same image lists to be used with multiple list view controls.
LVS_SHOWSELALWAYS	Always shows the selection highlighted, even if the control is not activated.
LVS_SINGLESEL	Enables only one item to be selected at a time. By default, multiple items can be selected.
LVS_SMALLICON	Specifies small icon view.
LVS_SORTASCENDING	Sorts items based on item text in ascending order.
LVS_SORTDESCENDING	Sorts items based on item text in descending order.

Month calendar control styles	Description
MCS_DAYSTATE	Specifies that the month calendar will send MCN_GETDAYSTATE notifications to request information about which days should be displayed in bold.
MCS_MULTISELECT	Enables the user to select a range of dates. By default, the maximum range is one week. You can change the maximum selectable range by using the MCM_SETMAXSELCOUNT message.
MCS_NOTODAY	Creates a month calendar that does not display a Today selection.
MCS_NOTODAYCIRCLE	Creates a month calendar that does not circle the current date.
MCS_WEEKNUMBERS	Displays the week number, from 1 through 52, to the left of each week in the calendar.

Progress bar styles	Description
PBS_SMOOTH	Displays progress status in a smooth scrolling bar instead of the default segmented bar.
PBS_VERTICAL	Displays progress status vertically, from bottom to top.

Rebar styles	Description
CCS_VERT	Causes the control to appear vertically at the left side of the parent window.
RBBS_NOGRIPPER	Creates a rebar band that displays no gripper. This style applies to individual bands, not to the entire rebar. Windows CE is the only Windows-based OS that supports the RBS_NOGRIPPER style for rebar controls.
RBS_AUTOSIZE	Specifies that the layout of a band will automatically change when the size or position of its control changes. When the layout changes, the control sends an RBN_AUTOSIZE notification.
RBS_BANDBORDERS	Displays narrow lines to separate adjacent bands.
RBS_FIXEDORDER	Displays multiple bands in the same order at all times. A user can move bands to different rows, but the band order is static.
RBS_SMARTLABELS	Displays the icon for a band that has an icon only when the band is minimized. If a band has a text label, the label is displayed only when the band is in its restored state or in its maximized state. Windows CE is the only Windows-based OS that supports the RBS_SMARTLABELS style for rebar controls.
RBS_VARHEIGHT	Displays a band at the minimum required height, when possible. Without this style, the command bands control displays all bands at the same height, using the height of the tallest visible band to determine the height of other bands.
RBS_VERTICALGRIPPER	Displays the size grip vertically, instead of horizontally, in a vertical command bands control. This style is ignored for command bands controls that do not have the CCS_VERT style.

Tab control styles	Description
TCS_BOTTOM	Displays the tabs at the bottom of the control. If the TCS_VERTICAL style is also specified, this style is interpreted as TCS_RIGHT.
TCS_BUTTONS	Displays all tabs as buttons with no border drawn around the display area.
TCS_FIXEDWIDTH	Specifies that all tabs are the same width. You can not combine this style with the TCS_RIGHTJUSTIFY style.
TCS_FLATBUTTONS	Changes the appearance of a selected tab to indented while other tabs appear to be on the same plane as the background. This style only applies to tab controls that have the TCS_BUTTONS style.
TCS_FLIP	Flips all tabs from top to bottom or left to right.
TCS_FOCUSNEVER	Creates a tab control that never receives the input focus.
TCS_FOCUSONBUTTONDOWN	Specifies that a given tab, when selected, receives the input focus.
TCS_FORCEICONLEFT	Aligns an icon with the left edge of a fixed-width tab. This style can only be used with the TCS_FIXEDWIDTH style.
TCS_FORCELABELLEFT	Aligns a label with the left edge of a fixed-width tab; that is, it displays the label immediately to the right of the icon instead of centering it. This style can only be used with the TCS_FIXEDWIDTH style, and it implies the TCS_FORCEICONLEFT style.
TCS_MULTILINE	Displays multiple rows of tabs, if necessary, so that all tabs are visible at once.
TCS_MULTISELECT	Specifies that multiple tabs can be selected by holding down CTRL when selecting a tab. This style only applies to tabs that have the TCS_BUTTONS style.
TCS_OWNERDRAWFIXED	Specifies that the parent window is responsible for drawing tabs.
TCS_RAGGEDRIGHT	Leaves a ragged right edge by not stretching a row of tabs to fill the entire width of the control. This style is the default.

Tab control styles	Description
TCS_RIGHT	Displays multiple tabs vertically on the right side of controls that use the TCS_VERTICAL style. If the TCS_VERTICAL style is not specified, this style is interpreted as TCS_BOTTOM.
TCS_RIGHTJUSTIFY	Increases the width of each tab, if necessary, so that each row of tabs fills the entire width of the tab control. This style is valid only when it is used with the TCS_MULTILINE style.
TCS_SCROLLOPPOSITE	Specifies that unused tabs move to the opposite side of the control when a new tab is selected.
TCS_SINGLELINE	Displays only one row of tabs. The user can scroll to see more tabs, if necessary. This style is the default.
TCS_VERTICAL	Displays multiple tabs vertically on the left side of the control. This style is valid only when it is used with the TCS_MULTILINE style. To make tabs appear on the right side of the control, combine this style with the TCS_RIGHT style.

Toolbar styles	Description
TBSTYLE_AUTOSIZE	Calculates a button width based on the text of the button, not on the size of the image.
TBSTYLE_BUTTON	Creates a toolbar button that looks like a standard Windows CE push button.
TBSTYLE_CHECK	Creates a button that toggles between the pressed and not pressed states each time the user clicks it. The button has a different background color when it is in the pressed state.
TBSTYLE_CHECKGROUP	Creates a check button that stays pressed until another button in the group is pressed.
TBSTYLE_CUSTOMERASE	Creates a toolbar that generates NM_CUSTOMDRAW notification messages when it processes WM_ERASEBKGND messages.
TBSTYLE_DROPDOWN	Creates a drop-down list button.
TBSTYLE_FLAT	Creates a flat toolbar, in which both the toolbar and the buttons are transparent. Button text appears under button bitmaps.

Toolbar styles	Description
TBSTYLE_GROUP	Creates a button that stays pressed until another button in the group is pressed.
TBSTYLE_LIST	Places button text to the right of button bitmaps. This style can only be used with the TBSTYLE_FLAT style. In Windows CE, the TBSTYLE_LIST style creates a toolbar with variable width buttons. If you want to use the TBSTYLE_LIST style with fixed width buttons, you can override the default behavior by sending a TB_SETBUTTONSIZE or TB_SETBUTTONWIDTH message.
TBSTYLE_SEP	Creates a separator, which provides a small gap between button groups. A button that has this style does not receive user input.
TBSTYLE_TOOLTIPS	Creates a ToolTip control that an application can use to display descriptive text for the buttons in the toolbar.
TBSTYLE_TRANSPARENT	Creates a transparent toolbar, in which the toolbar is transparent, but the buttons are not. Button text appears under button bitmaps.
TBSTYLE_WRAPABLE	Creates a toolbar that can have multiple rows of buttons. Toolbar buttons can wrap to the next line when the toolbar becomes too narrow to include all buttons on the same line. Wrapping occurs on separation and non-group boundaries.

Tree view styles	Description
TVS_CHECKBOXES	Enables items in a tree view control to be displayed as check boxes. This style uses item state images to produce the check box effect.
TVS_DISABLEDRAGDROP	Prevents the tree view control from sending TVN_BEGINDRAG notification messages.
TVS_EDITLABELS	Enables the user to edit the labels of tree view items.
TVS_HASBUTTONS	Displays plus (+) and minus (–) buttons next to parent items. The user taps the buttons to expand or collapse a parent item's list of child items. To include buttons with items at the root of the tree view, you must also specify the TVS_LINESATROOT style.

Toolbar styles	Description
TVS_HASLINES	Uses lines to show the hierarchy of items.
TVS_LINESATROOT	Uses lines to link items at the root of the tree view control. This value is ignored if the TVS_HASLINES control is not also specified.
TVS_SHOWSELALWAYS	Uses the system highlight colors to draw the selected item.
TVS_SINGLESEL	Specifies that when a new tree view item is selected, the selected item will automatically expand and the previously selected item will collapse.

Up-down control styles	Description
UDS_ALIGNLEFT	Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to the right and its width is decreased to accommodate the width of the up-down control.
UDS_ALIGNRIGHT	Positions the up-down control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the up-down control.
UDS_ARROWKEYS	Causes the up-down control to process the UP ARROW and DOWN ARROW keys on the keyboard.
UDS_AUTOBUDDY	Automatically elects the previous window in the z-order as the up-down control's buddy window. In Windows CE, the window must be an edit control.
UDS_HORZ	Causes the up-down control's arrows to point left and right instead of up and down.
UDS_NOTHOUSANDS	Prevents insertion of a thousands separator between every three decimal positions.
UDS_SETBUDDYINT	Causes the up-down control to set the text of the buddy window, using the WM_SETTEXT message, when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.
UDS_WRAP	Causes the position to wrap if it is incremented or decremented beyond the end or beginning of the range.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

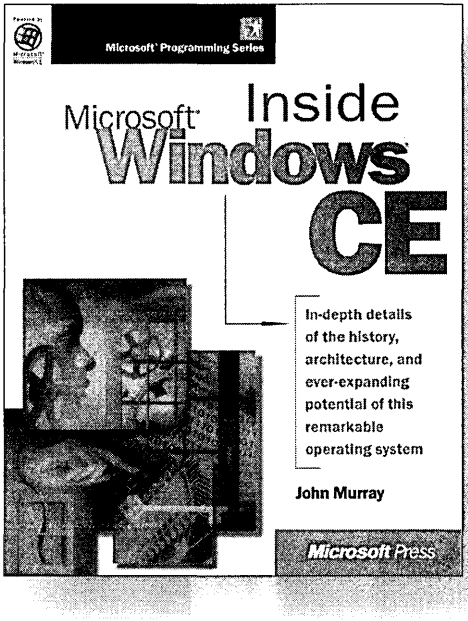
V

W

X

Z

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

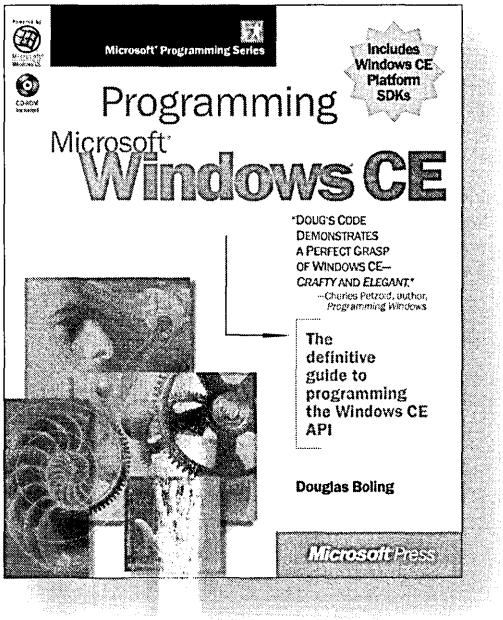
From roadside computing and pocket PCs to smart appliances and rich multimedia home theater, Microsoft® Windows® CE opens dynamic new development vistas for work, home, and everywhere in between. This modular, customizable operating system extends the Windows platform far beyond the desktop to the realm 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.

icrosoft Press® products are available worldwide wherever quality computer books are ıld. For more information, contact your book or computer retailer, software reseller, or cal Microsoft Sales Office, or visit our Web site at mspress.microsoft.com. To locate your ɜarest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in e U.S. (in Canada, call 1-800-268-2222).

ices and availability dates are subject to change.

Microsoft®
mspress.microsoft.com

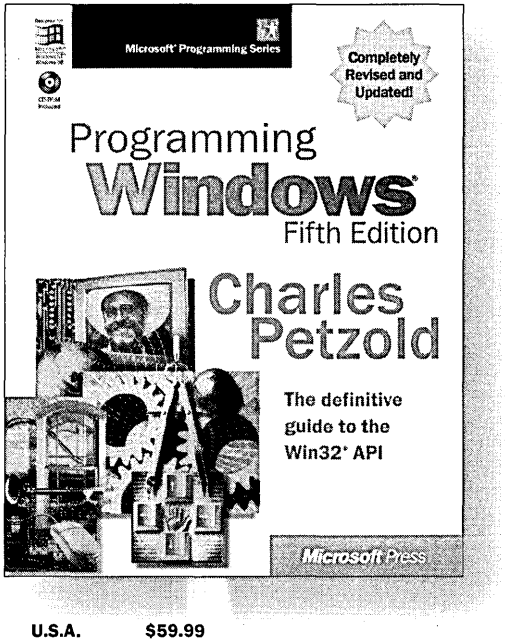
The *definitive guide* to programming the **Windows CE API**



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!

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®
mspress.microsoft.com

Microsoft® Windows® CE User Interface Services Guide

Your official guide to the Windows CE user interface—straight from the source.

Here's authoritative information to help you maximize the functionality of the user interface (UI) on Windows CE devices. This guide delves into the Graphics, Windowing, and Events Subsystem (GWES) interface in Windows CE that allows users to control an application. Understand how to implement the windowing, messaging, and power-management capabilities provided by GWES to optimize the UI for target audiences and devices.

Get the definitive guide to programming the Windows CE API.

Programming Microsoft Windows CE
ISBN: 1-57231-856-2

Part No. 097-0002195

mspress.microsoft.com

Microsoft® Press